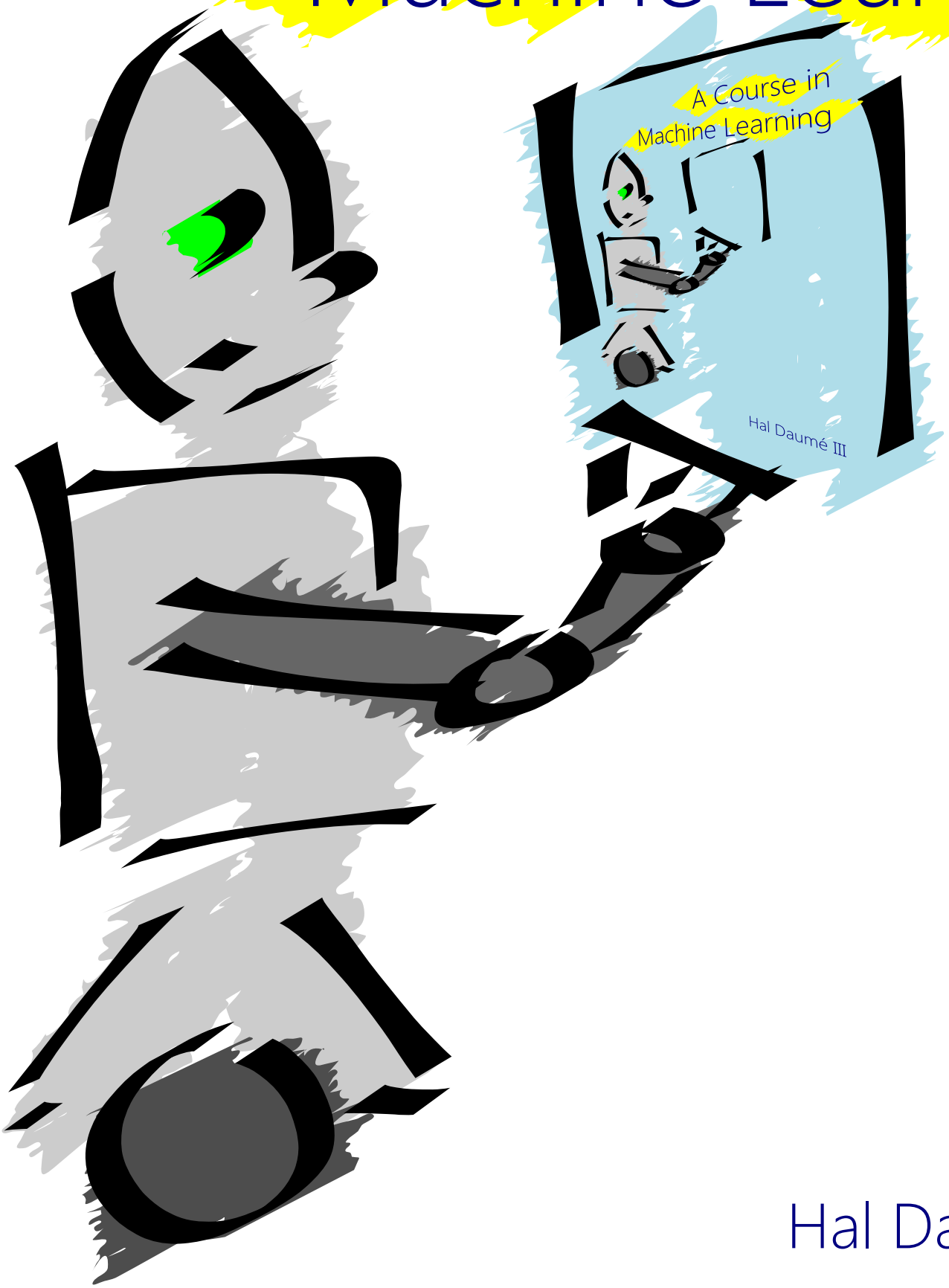
# A Course in Machine Learning




Hal Daumé III

Published by TODO

http://hal3.name/courseml/

**TODO...**.

*First printing, September 2014*

For my students and teachers.

Often the same.

# Table of Contents

# About this Book

Machine learning is a broad and fascinating field. It has been called one of the sexiest fields to work in[1]. It has applications in an incredibly wide variety of application areas, from medicine to advertising, from military to pedestrian. Its importance is likely to grow, as more and more areas turn to it as a way of dealing with the massive amounts of data available.

[1]

## 0.1 How to Use this Book

## 0.2 Why Another Textbook?

The purpose of this book is to provide a *gentle* and *pedagogically organized* introduction to the field. This is in contrast to most existing machine learning texts, which tend to organize things topically, rather than pedagogically (an exception is Mitchell's book[2], but unfortunately that is getting more and more outdated). This makes sense for researchers in the field, but less sense for learners. A second goal of this book is to provide a view of machine learning that focuses on ideas and models, not on math. It is not possible (or even advisable) to avoid math. But math should be there to *aid* understanding, not hinder it. Finally, this book attempts to have minimal dependencies, so that one can fairly easily pick and choose chapters to read. When dependencies exist, they are listed at the start of the chapter, as well as the list of dependencies at the end of this chapter.

[2] ?

The *audience* of this book is anyone who knows differential calculus and discrete math, and can program reasonably well. (A little bit of linear algebra and probability will not hurt.) An undergraduate in their fourth or fifth semester should be fully capable of understanding this material. However, it should also be suitable for first year graduate students, perhaps at a slightly faster pace.

## 0.3 Organization and Auxilary Material

There is an associated web page, http://hal3.name/courseml/, which contains an online copy of this book, as well as associated code and data. It also contains errate. For instructors, there is the ability to get a solutions manual.

This book is suitable for a single-semester undergraduate course, graduate course or two semester course (perhaps the latter supplemented with readings decided upon by the instructor). Here are suggested course plans for the first two courses; a year-long course could be obtained simply by covering the entire book.

## 0.4 Acknowledgements

# 1 | Decision Trees

> The words printed here are concepts.
> You must go through the experiences. – Carl Frederick

**Learning Objectives:**

- Explain the difference between memorization and generalization.
- Define "inductive bias" and recognize the role of inductive bias in learning.
- Take a concrete task and cast it as a learning problem, with a formal notion of input space, features, output space, generating distribution and loss function.
- Illustrate how regularization trades off between underfitting and overfitting.
- Evaluate whether a use of test data is "cheating" or not.

Dependencies: None.

At a basic level, machine learning is about predicting the future based on the past. For instance, you might wish to predict how much a user Alice will like a movie that she hasn't seen, based on her ratings of movies that she has seen. This means making informed guesses about some unobserved property of some object, based on observed properties of that object.

The first question we'll ask is: what does it mean to learn? In order to develop learning machines, we must know what learning actually means, and how to determine success (or failure). You'll see this question answered in a very limited learning setting, which will be progressively loosened and adapted throughout the rest of this book. For concreteness, our focus will be on a very simple model of learning called a **decision tree**.

**Vignette: Alice Decides which Classes to Take**

todo

## 1.1 What Does it Mean to Learn?

Alice has just begun taking a course on machine learning. She knows that at the end of the course, she will be expected to have "learned" all about this topic. A common way of gauging whether or not she has learned is for her teacher, Bob, to give her a exam. She has done well at learning if she does well on the exam.

But what makes a reasonable exam? If Bob spends the entire semester talking about machine learning, and then gives Alice an exam on History of Pottery, then Alice's performance on this exam will *not* be representative of her learning. On the other hand, if the exam only asks questions that Bob has answered exactly during lectures, then this is also a bad test of Alice's learning, especially if it's an "open notes" exam. What is desired is that Alice observes *specific*

examples from the course, and then has to answer new, but related questions on the exam. This tests whether Alice has the ability to **generalize**. Generalization is perhaps the most central concept in machine learning.

As a running concrete example in this book, we will use that of a course recommendation system for undergraduate computer science students. We have a collection of students and a collection of courses. Each student has taken, and evaluated, a subset of the courses. The evaluation is simply a score from $-2$ (terrible) to $+2$ (awesome). The job of the recommender system is to **predict** how much a particular student (say, Alice) will like a particular course (say, Algorithms).

Given historical data from course ratings (i.e., the past) we are trying to predict unseen ratings (i.e., the future). Now, we could be unfair to this system as well. We could ask it whether Alice is likely to enjoy the History of Pottery course. This is unfair because the system has no idea what History of Pottery even is, and has no prior experience with this course. On the other hand, we could ask it how much Alice will like Artificial Intelligence, which she took last year and rated as $+2$ (awesome). We would expect the system to predict that she would really like it, but this isn't demonstrating that the system has learned: it's simply recalling its past experience. In the former case, we're expecting the system to generalize *beyond* its experience, which is unfair. In the latter case, we're not expecting it to generalize at all.

This general set up of predicting the future based on the past is at the core of most machine learning. The objects that our algorithm will make predictions about are **examples**. In the recommender system setting, an example would be some particular Student/Course pair (such as Alice/Algorithms). The desired prediction would be the rating that Alice would give to Algorithms.

To make this concrete, Figure **??** shows the general framework of **induction**. We are given **training data** on which our algorithm is expected to learn. This training data is the examples that Alice observes in her machine learning course, or the historical ratings data for the recommender system. Based on this training data, our learning algorithm induces a function $f$ that will map a new example to a corresponding prediction. For example, our function might guess that $f(\text{Alice/Machine Learning})$ might be high because our training data said that Alice liked Artificial Intelligence. We want our algorithm to be able to make lots of predictions, so we refer to the collection of examples on which we will evaluate our algorithm as the **test set**. The test set is a closely guarded secret: it is the final exam on which our learning algorithm is being tested. If our algorithm gets to peek at it ahead of time, it's going to cheat and do better than it should.

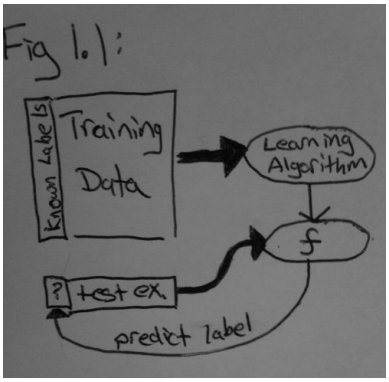



Figure 1.1: The general supervised approach to machine learning: a learning algorithm reads in training data and computes a learned function $f$. This function can then automatically label future text examples.

? Why is it bad if the learning algorithm gets to peek at the test data?

The goal of inductive machine learning is to take some training data and use it to induce a function $f$. This function $f$ will be evaluated on the test data. The machine learning algorithm has succeeded if its performance on the test data is high.

## 1.2 Some Canonical Learning Problems

There are a large number of typical inductive learning problems. The primary difference between them is in what type of *thing* they're trying to predict. Here are some examples:

*Regression:* trying to predict a real value. For instance, predict the value of a stock tomorrow given its past performance. Or predict Alice's score on the machine learning final exam based on her homework scores.

*Binary Classification:* trying to predict a simple yes/no response. For instance, predict whether Alice will enjoy a course or not. Or predict whether a user review of the newest Apple product is positive or negative about the product.

*Multiclass Classification:* trying to put an example into one of a number of classes. For instance, predict whether a news story is about entertainment, sports, politics, religion, etc. Or predict whether a CS course is Systems, Theory, AI or Other.

*Ranking:* trying to put a set of objects in order of relevance. For instance, predicting what order to put web pages in, in response to a user query. Or predict Alice's ranked preferences over courses she hasn't taken.

? For each of these types of canonical machine learning problems, come up with one or two concrete examples.

The reason that it is convenient to break machine learning problems down by the type of object that they're trying to predict has to do with measuring error. Recall that our goal is to build a system that can make "good predictions." This begs the question: what does it mean for a prediction to be "good?" The different types of learning problems differ in how they define goodness. For instance, in regression, predicting a stock price that is off by \$0.05 is perhaps much better than being off by \$200.00. The same does not hold of multiclass classification. There, accidentally predicting "entertainment" instead of "sports" is no better or worse than predicting "politics."

## 1.3 The Decision Tree Model of Learning

The **decision tree** is a classic and natural model of learning. It is closely related to the fundamental computer science notion of "divide and conquer." Although decision trees can be applied to many

learning problems, we will begin with the simplest case: binary classification.

Suppose that your goal is to predict whether some unknown user will enjoy some unknown course. You must simply answer "yes" or "no." In order to make a guess, your're allowed to ask binary questions about the user/course under consideration. For example:

**You:** Is the course under consideration in Systems?

**Me:** Yes

**You:** Has this student taken any other Systems courses?

**Me:** Yes

**You:** Has this student like most previous Systems courses?

**Me:** No

**You:** *I predict this student will not like this course.*

The goal in learning is to figure out what questions to ask, in what order to ask them, and what answer to predict once you have asked enough questions.

The decision tree is so-called because we can write our set of questions and guesses in a tree format, such as that in Figure 1.2. In this figure, the questions are written in the internal tree nodes (rectangles) and the guesses are written in the leaves (ovals). Each non-terminal node has two children: the left child specifies what to do if the answer to the question is "no" and the right child specifies what to do if it is "yes."

In order to learn, I will give you training data. This data consists of a set of user/course examples, paired with the correct answer for these examples (did the given user enjoy the given course?). From this, you must construct your questions. For concreteness, there is a small data set in Table **??** in the Appendix of this book. This training data consists of 20 course rating examples, with course ratings and answers to questions that you might ask about this pair. We will interpret ratings of 0, $+1$ and $+2$ as "liked" and ratings of $-2$ and $-1$ as "hated."

In what follows, we will refer to the questions that you can ask as **features** and the responses to these questions as **feature values**. The rating is called the **label**. An example is just a set of feature values. And our training data is a set of examples, paired with labels.

There are a lot of logically possible trees that you could build, even over just this small number of features (the number is in the millions). It is computationally infeasible to consider all of these to try to choose the "best" one. Instead, we will build our decision tree *greedily.* We will begin by asking:

**If I could only ask one question, what question would I ask?**

You want to find a feature that is *most useful* in helping you guess whether this student will enjoy this course.[1] A useful way to think

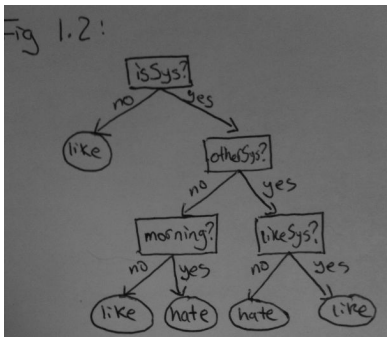



Figure 1.2: A decision tree for a course recommender system, from which the in-text "dialog" is drawn.

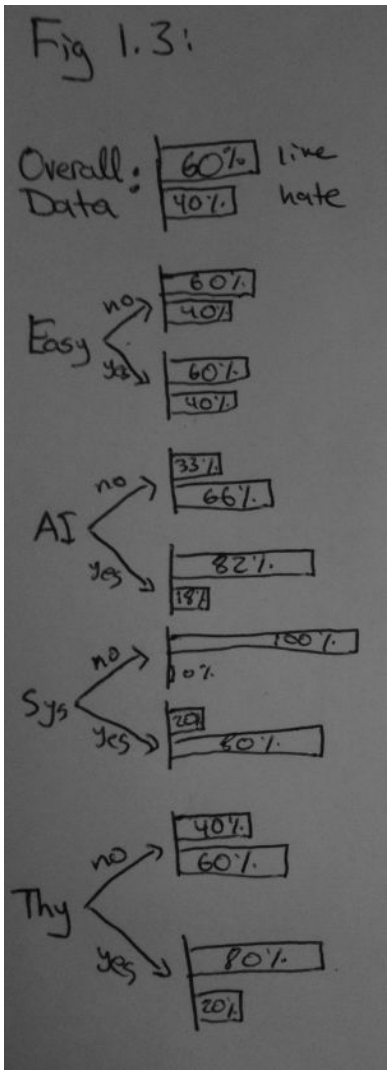



Figure 1.3: A histogram of labels for (a) the entire data set; (b-e) the examples in the data set for each value of the first four features.

[1] A colleague related the story of getting his 8-year old nephew to guess a number between 1 and 100. His nephew's first four questions were: Is it bigger than 20? (YES) Is it even? (YES) Does it have a 7 in it?

about this is to look at the **histogram** of labels for each feature. This is shown for the first four features in Figure 1.3. Each histogram shows the frequency of "like"/"hate" labels for each possible value of an associated feature. From this figure, you can see that asking the first feature is not useful: if the value is "no" then it's hard to guess the label; similarly if the answer is "yes." On the other hand, asking the second feature *is* useful: if the value is "no," you can be pretty confident that this student will like this course; if the answer is "yes," you can be pretty confident that this student will hate this course.

More formally, you will consider each feature in turn. You might consider the feature "Is this a System's course?" This feature has two possible value: no and yes. Some of the training examples have an answer of "no" – let's call that the "NO" set. Some of the training examples have an answer of "yes" – let's call that the "YES" set. For each set (NO and YES) we will build a histogram over the labels. This is the second histogram in Figure 1.3. Now, suppose you were to ask this question on a random example and observe a value of "no." Further suppose that you must *immediately* guess the label for this example. You will guess "like," because that's the more prevalent label in the NO set (actually, it's the *only* label in the NO set). Alternative, if you recieve an answer of "yes," you will guess "hate" because that is more prevalent in the YES set.

So, for this single feature, you know what you *would* guess if you had to. Now you can ask yourself: if I made that guess on the *training data,* how well would I have done? In particular, how many examples would I classify *correctly?* In the NO set (where you guessed "like") you would classify all 10 of them correctly. In the YES set (where you guessed "hate") you would classify 8 (out of 10) of them correctly. So overall you would classify 18 (out of 20) correctly. Thus, we'll say that the *score* of the "Is this a System's course?" question is 18/20.

? How many training examples would you classify correctly for each of the other three features from Figure 1.3?

You will then repeat this computation for each of the available features to us, compute the scores for each of them. When you must choose which feature consider first, you will want to choose the one with the highest score.

But this only lets you choose the *first* feature to ask about. This is the feature that goes at the *root* of the decision tree. How do we choose subsequent features? This is where the notion of divide and conquer comes in. You've already decided on your first feature: "Is this a Systems course?" You can now *partition* the data into two parts: the NO part and the YES part. The NO part is the subset of the data on which value for this feature is "no"; the YES half is the rest. This is the *divide* step.

The *conquer* step is to recurse, and run the *same* routine (choosing

**Algorithm 1** DecisionTreeTrain(*data*, *remaining features*)

```
guess ← most frequent answer in data                    // default answer for this data
if the labels in data are unambiguous then
    return Leaf(guess)                                  // base case: no need to split further
else if remaining features is empty then
    return Leaf(guess)                                  // base case: cannot split further
else                                                    // we need to query more features
    for all f ∈ remaining features do
        NO ← the subset of data on which f=no
        YES ← the subset of data on which f=yes
        score[f] ← # of majority vote answers in NO
                 + # of majority vote answers in YES
                                                        // the accuracy we would get if we only queried on f
    end for
    f ← the feature with maximal score(f)
    NO ← the subset of data on which f=no
    YES ← the subset of data on which f=yes
    left ← DecisionTreeTrain(NO, remaining features \ {f})
    right ← DecisionTreeTrain(YES, remaining features \ {f})
    return Node(f, left, right)
end if
```

**Algorithm 2** DecisionTreeTest(*tree*, *test point*)

```
if tree is of the form Leaf(guess) then
    return guess
else if tree is of the form Node(f, left, right) then
    if f = yes in test point then
        return DecisionTreeTest(left, test point)
    else
        return DecisionTreeTest(right, test point)
    end if
end if
```

the feature with the highest score) on the NO set (to get the left half of the tree) and then separately on the YES set (to get the right half of the tree).

At some point it will become useless to query on additional features. For instance, once you know that this is a Systems course, you *know* that everyone will hate it. So you can immediately predict "hate" without asking any additional questions. Similarly, at some point you might have already queried every available feature and still not whittled down to a single answer. In both cases, you will need to create a leaf node and guess the most prevalent answer in the current piece of the training data that you are looking at.

Putting this all together, we arrive at the algorithm shown in Algorithm 1.3.[2] This function, DecisionTreeTrain takes two arguments: our data, and the set of as-yet unused features. It has two

[2] There are more nuanced algorithms for building decision trees, some of which are discussed in later chapters of this book. They primarily differ in how they compute the *score* funciton.

base cases: either the data is unambiguous, or there are no remaining features. In either case, it returns a LEAF node containing the most likely guess at this point. Otherwise, it loops over all remaining features to find the one with the highest score. It then partitions the data into a NO/YES split based on the best feature. It constructs its left and right subtrees by recursing on itself. In each recursive call, it uses one of the partitions of the data, and removes the just-selected feature from consideration.

? Is the Algorithm in Figure **??** guaranteed to terminate?

The corresponding *prediction* algorithm is shown in Algorithm **??**. This function recurses down the decision tree, following the edges specified by the feature values in some *test point*. When it reaches a leave, it returns the guess associated with that leaf.

TODO: define outlier somewhere!

## 1.4 Formalizing the Learning Problem

As you've seen, there are several issues that we must take into account when formalizing the notion of learning.

- The performance of the learning algorithm should be measured on unseen "test" data.
- The way in which we measure performance should depend on the problem we are trying to solve.
- There should be a strong relationship between the data that our algorithm sees at training time and the data it sees at test time.

In order to accomplish this, let's assume that someone gives us a **loss function**, $\ell(\cdot, \cdot)$, of two arguments. The job of $\ell$ is to tell us how "bad" a system's prediction is in comparison to the truth. In particular, if $y$ is the truth and $\hat{y}$ is the system's prediction, then $\ell(y, \hat{y})$ is a measure of error.

For three of the canonical tasks discussed above, we might use the following loss functions:

*Regression:* **squared loss** $\ell(y, \hat{y}) = (y - \hat{y})^2$
or **absolute loss** $\ell(y, \hat{y}) = |y - \hat{y}|$.

*Binary Classification:* **zero/one loss** $\ell(y, \hat{y}) = \begin{cases} 0 & \text{if } y = \hat{y} \\ 1 & \text{otherwise} \end{cases}$

This notation means that the loss is zero if the prediction is correct and is one otherwise.

*Multiclass Classification:* also zero/one loss.

? Why might it be a bad idea to use zero/one loss to measure performance for a regression problem?

Note that the loss function is something that *you* must decide on based on the goals of learning.

Now that we have defined our loss function, we need to consider where the data (training *and* test) comes from. The model that we

will use is the *probabilistic* model of learning. Namely, there is a probability distribution $\mathcal{D}$ over input/output pairs. This is often called the **data generating distribution**. If we write $\boldsymbol{x}$ for the input (the user/course pair) and $y$ for the output (the rating), then $\mathcal{D}$ is a distribution over $(\boldsymbol{x}, y)$ pairs.

A useful way to think about $\mathcal{D}$ is that it gives *high probability* to reasonable $(\boldsymbol{x}, y)$ pairs, and *low probability* to unreasonable $(\boldsymbol{x}, y)$ pairs. A $(\boldsymbol{x}, y)$ pair can be unreasonable in two ways. First, $\boldsymbol{x}$ might an unusual input. For example, a $\boldsymbol{x}$ related to an "Intro to Java" course might be highly probable; a $\boldsymbol{x}$ related to a "Geometric and Solid Modeling" course might be less probable. Second, $y$ might be an unusual rating for the paired $\boldsymbol{x}$. For instance, if Alice were to take AI 100 times (without remembering that she took it before!), she would give the course a $+2$ almost every time. Perhaps some semesters she might give a slightly lower score, but it would be unlikely to see $\boldsymbol{x} =$Alice/AI paired with $y = -2$.

It is important to remember that we are not making *any* assumptions about what the distribution $\mathcal{D}$ looks like. (For instance, we're not assuming it looks like a Gaussian or some other, common distribution.) We are also not assuming that we know what $\mathcal{D}$ is. In fact, if you know *a priori* what your data generating distribution is, your learning problem becomes significantly easier. Perhaps the hardest think about machine learning is that we *don't* know what $\mathcal{D}$ is: all we get is a random sample from it. This random sample is our training data.

Our learning problem, then, is defined by two quantities:

1. The loss function $\ell$, which captures our notion of what is *important* to learn.
2. The data generating distribution $\mathcal{D}$, which defines what sort of data we expect to see.

? Consider the following prediction task. Given a paragraph written about a course, we have to predict whether the paragraph is a *positive* or *negative* review of the course. (This is the sentiment analysis problem.) What is a reasonable loss function? How would you define the data generating distribution?

We are given access to **training data**, which is a random sample of input/output pairs drawn from $\mathcal{D}$. Based on this training data, we need to **induce** a function $f$ that maps new inputs $\hat{\boldsymbol{x}}$ to corresponding prediction $\hat{y}$. The key property that $f$ should obey is that it should do well (as measured by $\ell$) on future examples that are *also* drawn from $\mathcal{D}$. Formally, it's **expected loss** $\epsilon$ over $\mathcal{D}$ with repsect to $\ell$ should be as small as possible:

$$\epsilon \triangleq \mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}}\big[\ell(y, f(\boldsymbol{x}))\big] = \sum_{(\boldsymbol{x},y)} \mathcal{D}(\boldsymbol{x}, y)\ell(y, f(\boldsymbol{x})) \tag{1.1}$$

The difficulty in minimizing our **expected loss** from Eq (1.1) is that we *don't know what $\mathcal{D}$ is!* All we have access to is some training

**MATH REVIEW | EXPECTATED VALUES**

remind people what expectations are and explain the notation in Eq (1.1).

data sampled from it! Suppose that we denote our training data set by $D$. The training data consists of $N$-many input/output pairs, $(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_N, y_N)$. Given a learned function $f$, we can compute our **training error**, $\hat{\epsilon}$:

$$\hat{\epsilon} \triangleq \frac{1}{N} \sum_{n=1}^{N} \ell(y_n, f(\boldsymbol{x}_n)) \tag{1.2}$$

That is, our training error is simply our *average error* over the training data.

Of course, we can drive $\hat{\epsilon}$ to zero by simply memorizing our training data. But as Alice might find in memorizing past exams, this might not generalize well to a new exam!

This is the fundamental difficulty in machine learning: the thing we have access to is our training error, $\hat{\epsilon}$. But the thing we care about minimizing is our expected error $\epsilon$. In order to get the expected error down, our learned function needs to **generalize** beyond the training data to some future data that it might not have seen yet!

So, putting it all together, we get a formal definition of induction machine learning: **Given (i) a loss function $\ell$ and (ii) a sample $D$ from some unknown distribution $\mathcal{D}$, you must compute a function $f$ that has low expected error $\epsilon$ over $\mathcal{D}$ with respect to $\ell$.**

Figure 1.4:

? Verify by calculation that we can write our training error as $\mathbb{E}_{(\boldsymbol{x},y)\sim D}\big[\ell(y, f(\boldsymbol{x}))\big]$, by thinking of $D$ as a distribution that places probability $1/N$ to each example in $D$ and probabiliy 0 on everything else.

## 1.5 *Inductive Bias: What We Know Before the Data Arrives*

In Figure 1.5 you'll find training data for a binary classification problem. The two labels are "A" and "B" and you can see five examples for each label. Below, in Figure 1.6, you will see some test data. These images are left unlabeled. Go through quickly and, based on the training data, label these images. (Really do it before you read further! I'll wait!)

Figure 1.5: `dt:bird`: bird training images

Most likely you produced one of two labelings: either ABBAAB or ABBABA. Which of these solutions is right?

The answer is that you cannot tell based on the training data. If you give this same example to 100 people, $60-70$ of them come up with the ABBAAB prediction and $30-40$ come up with the ABBABA prediction. Why are they doing this? Presumably because the first group believes that the relevant distinction is between "bird" and

Figure 1.6: `dt:birdtest`: bird test images

"non-bird" while the secong group believes that the relevant distinction is between "fly" and "no-fly."

This preference for one distinction (bird/non-bird) over another (fly/no-fly) is a bias that different human learners have. In the context of machine learning, it is called **inductive bias**: in the absense of data that narrow down the relevant concept, what type of solutions are we more likely to prefer? Two thirds of people seem to have an inductive bias in favor of bird/non-bird, and one third seem to have an inductive bias in favor of fly/no-fly.

Throughout this book you will learn about several approaches to machine learning. The decision tree model is the first such approach. These approaches differ primarily in the sort of inductive bias that they exhibit.

Consider a variant of the decision tree learning algorithm. In this variant, we will not allow the trees to grow beyond some pre-defined maximum depth, $d$. That is, once we have queried on $d$-many features, we cannot query on any more and must just make the best guess we can at that point. This variant is called a **shallow decision tree**.

The key question is: What is the inductive bias of shallow decision trees? Roughly, their bias is that decisions can be made by only looking at a small number of features. For instance, a shallow decision tree would be very good a learning a function like "students only like AI courses." It would be very bad at learning a function like "if this student has liked an odd number of his past courses, he will like the next one; otherwise he will not." This latter is the *parity* function, which requires you to inspect every feature to make a prediction. The inductive bias of a decision tree is that the sorts of things we want to learn to predict are more like the first example and less like the second example.

? It is also possible that the correct classification on the test data is BABAAA. This corresponds to the bias "is the background in focus." Somehow no one seems to come up with this classification rule.

## 1.6 Not Everything is Learnable

Although machine learning works well—perhaps astonishingly well—in many cases, it is important to keep in mind that it is not magical. There are many reasons why a machine learning algorithm might fail on some learning task.

There could be **noise** in the training data. Noise can occur both at the feature level and at the label level. Some features might correspond to measurements taken by sensors. For instance, a robot might use a laser range finder to compute its distance to a wall. However, this sensor might fail and return an incorrect value. In a sentiment classification problem, someone might have a typo in their review of a course. These would lead to noise at the feature level. There might

also be noise at the label level. A student might write a scathingly negative review of a course, but then accidentally click the wrong button for the course rating.

The features available for learning might simply be insufficient. For example, in a medical context, you might wish to diagnose whether a patient has cancer or not. You may be able to collect a large amount of data about this patient, such as gene expressions, X-rays, family histories, etc. But, even knowing all of this information exactly, it might still be impossible to judge for sure whether this patient has cancer or not. As a more contrived example, you might try to classify course reviews as positive or negative. But you may have erred when downloading the data and only gotten the first five characters of each review. If you had the rest of the features you might be able to do well. But with this limited feature set, there's not much you can do.

Some example may not have a single correct answer. You might be building a system for "safe web search," which removes offensive web pages from search results. To build this system, you would collect a set of web pages and ask people to classify them as "offensive" or not. However, what one person considers offensive might be completely reasonable for another person. It is common to consider this as a form of label noise. Nevertheless, since you, as the designer of the learning system, have some control over this problem, it is sometimes helpful to isolate it as a source of difficulty.

Finally, learning might fail because the inductive bias of the learning algorithm is too far away from the concept that is being learned. In the bird/non-bird data, you might think that if you had gotten a few more training examples, you might have been able to tell whether this was intended to be a bird/non-bird classification or a fly/no-fly classification. However, no one I've talked to has ever come up with the "background is in focus" classification. Even with many more training points, this is such an unusual distinction that it may be hard for anyone to figure out it. In this case, the inductive bias of the learner is simply too misaligned with the target classification to learn.

Note that the inductive bias source of error is fundamentally different than the other three sources of error. In the inductive bias case, it is the *particular* learning algorithm that you are using that cannot cope with the data. Maybe if you switched to a different learning algorithm, you would be able to learn well. For instance, Neptunians might have evolved to care greatly about whether backgrounds are in focus, and for them this would be an easy classification to learn. For the other three sources of error, it is not an issue to do with the particular learning algorithm. The error is a fundamental part of the

learning problem.

## 1.7 Underfitting and Overfitting

As with many problems, it is useful to think about the *extreme cases* of learning algorithms. In particular, the extreme cases of decision trees. In one extreme, the tree is "empty" and we do not ask any questions at all. We simply immediate make a prediction. In the other extreme, the tree is "full." That is, every possible question is asked along every branch. In the full tree, there may be leaves with no associated training data. For these we must simply choose arbitrarily whether to say "yes" or "no."

Consider the course recommendation data from Table **??**. Suppose we were to build an "empty" decision tree on this data. Such a decision tree will make the same prediction regardless of its input, because it is not allowed to ask any questions about its input. Since there are more "likes" than "hates" in the training data (12 versus 8), our empty decision tree will simply always predict "likes." The training error, $\hat{\epsilon}$, is $8/20 = 40\%$.

On the other hand, we could build a "full" decision tree. Since each row in this data is unique, we can guarantee that any leaf in a full decision tree will have either 0 or 1 examples assigned to it (20 of the leaves will have one example; the rest will have none). For the leaves corresponding to training points, the full decision tree will always make the correct prediction. Given this, the training error, $\hat{\epsilon}$, is $0/20 = 0\%$.

Of course our goal is *not* to build a model that gets 0% error on the training data. This would be easy! Our goal is a model that will do well on *future, unseen* data. How well might we expect these two models to do on future data? The "empty" tree is likely to do not much better and not much worse on future data. We might expect that it would continue to get around 40% error.

Life is more complicated for the "full" decision tree. Certainly if it is given a test example that is identical to one of the training examples, it will do the right thing (assuming no noise). But for everything else, it will only get about 50% error. This means that even if every other test point happens to be identical to one of the training points, it would only get about 25% error. In practice, this is probably optimistic, and maybe only one in every 10 examples would match a training example, yielding a 35% error.

So, in one case (empty tree) we've achieved about 40% error and in the other case (full tree) we've achieved 35% error. This is not very promising! One would hope to do better! In fact, you might notice that if you simply queried on a *single* feature for this data, you

? Convince yourself (either by proof or by simulation) that even in the case of imbalanced data – for instance data that is on average 80% positive and 20% negative – a predictor that guesses randomly (50/50 positive/negative) will get about 50% error.

would be able to get very low training error, but wouldn't be forced to "guess" randomly.

This example illustrates the key concepts of **underfitting** and **overfitting**. Underfitting is when you had the opportunity to learn something but didn't. A student who hasn't studied much for an upcoming exam will be underfit to the exam, and consequently will not do well. This is also what the empty tree does. Overfitting is when you pay too much attention to idiosyncracies of the training data, and aren't able to generalize well. Often this means that your model is fitting noise, rather than whatever it is supposed to fit. A student who memorizes answers to past exam questions without understanding them has overfit the training data. Like the full tree, this student also will not do well on the exam. A model that is neither overfit nor underfit is the one that is expected to do best in the future.

? Which feature is it, and what is it's training error?

## 1.8 Separation of Training and Test Data

Suppose that, after graduating, you get a job working for a company that provides persolized recommendations for pottery. You go in and implement new algorithms based on what you learned in her machine learning class (you have learned the power of generalization!). All you need to do now is convince your boss that you has done a good job and deserve a raise!

How can you convince your boss that your fancy learning algorithms are really working?

Based on what we've talked about already with underfitting and overfitting, it is not enough to just tell your boss what your training error is. Noise notwithstanding, it is easy to get a training error of zero using a simple database query (or `grep`, if you prefer). Your boss will not fall for that.

The easiest approach is to *set aside* some of your available data as "test data" and use this to evaluate the performance of your learning algorithm. For instance, the pottery recommendation service that you work for might have collected 1000 examples of pottery ratings. You will select 800 of these as **training data** and set aside the final 200 as **test data**. You will run your learning algorithms *only* on the 800 training points. Only once you're done will you apply your learned model to the 200 test points, and report your **test error** on those 200 points to your boss.

The hope in this process is that however well you do on the 200 test points will be indicative of how well you are likely to do in the future. This is analogous to estimating support for a presidential candidate by asking a small (random!) sample of people for their opinions. Statistics (specifically, concentration bounds of which the

"Central limit theorem" is a famous example) tells us that if the sample is large enough, it will be a good representative. The 80/20 split is not magic: it's simply fairly well established. Occasionally people use a 90/10 split instead, especially if they have a *lot* of data.

? If you have more data at your disposal, why might a 90/10 split be preferable to an 80/20 split?

They cardinal rule of machine learning is: never touch your test data. Ever. If that's not clear enough:

Never *ever* touch your test data!

If there is only one thing you learn from this book, let it be that. Do not look at your test data. Even once. Even a tiny peek. Once you do that, it is not test data any more. Yes, perhaps your algorithm hasn't seen it. But you have. And you are likely a better learner than your learning algorithm. Consciously or otherwise, you might make decisions based on whatever you might have seen. Once you look at the test data, your model's performance on it is no longer indicative of it's performance on future unseen data. This is simply because future data is unseen, but your "test" data no longer is.

## 1.9 Models, Parameters and Hyperparameters

The general approach to machine learning, which captures many existing learning algorithms, is the **modeling** approach. The idea is that we come up with some formal model of our data. For instance, we might model the classification decision of a student/course pair as a decision tree. The choice of using a *tree* to represent this model is *our choice.* We also could have used an arithmetic circuit or a polynomial or some other function. The model tells us what sort of things we can learn, and also tells us what our inductive bias is.

For most models, there will be associated parameters. These are the things that we use the data to decide on. Parameters in a decision tree include: the specific questions we asked, the order in which we asked them, and the classification decisions at the leaves. The job of our decision tree learning algorithm DecisionTreeTrain is to take data and figure out a good set of parameters.

Many learning algorithms will have additional knobs that you can adjust. In most cases, these knobs amount to tuning the inductive bias of the algorithm. In the case of the decision tree, an obvious knob that one can tune is the **maximum depth** of the decision tree. That is, we could modify the DecisionTreeTrain function so that it *stops* recursing once it reaches some pre-defined maximum depth. By playing with this depth knob, we can adjust between underfitting (the empty tree, depth= 0) and overfitting (the full tree, depth= $\infty$).

? Go back to the DecisionTreeTrain algorithm and modify it so that it takes a maximum depth parameter. This should require adding two lines of code and modifying three others.

Such a knob is called a **hyperparameter**. It is so called because it

is a parameter that controls other parameters of the model. The exact definition of hyperparameter is hard to pin down: it's one of those things that are easier to identify than define. However, one of the key identifiers for hyperparameters (and the main reason that they cause consternation) is that they cannot be naively adjusted using the training data.

In DecisionTreeTrain, as in most machine learning, the learning algorithm is essentially trying to adjust the parameters of the model so as to minimize training error. This suggests an idea for choosing hyperparameters: choose them so that they minimize training error.

What is wrong with this suggestion? Suppose that you were to treat "maximum depth" as a hyperparameter and tried to tune it on your training data. To do this, maybe you simply build a collection of decision trees, $\text{tree}_0$, $\text{tree}_1$, $\text{tree}_2$, ..., $\text{tree}_{100}$, where $\text{tree}_d$ is a tree of maximum depth $d$. We then computed the training error of each of these trees and chose the "ideal" maximum depth as that which minimizes training error? Which one would it pick?

The answer is that it would pick $d = 100$. Or, in general, it would pick $d$ as large as possible. Why? Because choosing a bigger $d$ will *never hurt* on the training data. By making $d$ larger, you are simply encouraging overfitting. But by evaluating on the training data, overfitting actually looks like a good idea!

An alternative idea would be to tune the maximum depth on test data. This is promising because test data peformance is what we really want to optimize, so tuning this knob on the test data seems like a good idea. That is, it won't accidentally reward overfitting. Of course, it breaks our cardinal rule about test data: that you should never touch your test data. So that idea is immediately off the table.

However, our "test data" wasn't magic. We simply took our 1000 examples, called 800 of them "training" data and called the other 200 "test" data. So instead, let's do the following. Let's take our original 1000 data points, and select 700 of them as training data. From the remainder, take 100 as **development data**[3] and the remaining 200 as test data. The job of the development data is to allow us to tune hyperparameters. The general approach is as follows:

[3] Some people call this "**validation data**" or "**held-out data**."

1. Split your data into 70% training data, 10% development data and 20% test data.
2. For each possible setting of your hyperparameters:
   (a) Train a model using that setting of hyperparameters on the training data.
   (b) Compute this model's error rate on the development data.

3. From the above collection of models, choose the one that achieved the lowest error rate on development data.

4. Evaluate that model on the test data to estimate future test performance.

? In step 3, you could either choose the model (trained on the 70% training data) that did the best on the development data. Or you could choose the hyperparameter settings that did best and *retrain* the model on the 80% union of training and development data. Is either of these options obviously better or worse?

## 1.10 Chapter Summary and Outlook

At this point, you should be able to use decision trees to do machine learning. Someone will give you data. You'll split it into training, development and test portions. Using the training and development data, you'll find a good value for maximum depth that trades off between underfitting and overfitting. You'll then run the resulting decision tree model on the test data to get an estimate of how well you are likely to do in the future.

You might think: why should I read the rest of this book? Aside from the fact that machine learning is just an awesome fun field to learn about, there's a lot left to cover. In the next two chapters, you'll learn about two models that have very different inductive biases than decision trees. You'll also get to see a very useful way of thinking about learning: the geometric view of data. This will guide much of what follows. After that, you'll learn how to solve problems more complicated that simple binary classification. (Machine learning people like binary classification a lot because it's one of the simplest non-trivial problems that we can work on.) After that, things will diverge: you'll learn about ways to think about learning as a formal optimization problem, ways to speed up learning, ways to learn without labeled data (or with very little labeled data) and all sorts of other fun topics.

But throughout, we will focus on the view of machine learning that you've seen here. You select a model (and its associated inductive biases). You use data to find parameters of that model that work well on the training data. You use development data to avoid underfitting and overfitting. And you use test data (which you'll never look at or touch, right?) to estimate future model performance. Then you conquer the world.

## 1.11 Exercises

**Exercise 1.1. TODO...**

# 2 | Geometry and Nearest Neighbors

> Our brains have evolved to get us out of the rain, find where the berries are, and keep us from getting killed. Our brains did not evolve to help us grasp really large numbers or to look at things in a hundred thousand dimensions. – Ronald Graham

**Learning Objectives:**

- Describe a data set as points in a high dimensional space.
- Explain the curse of dimensionality.
- Compute distances between points in high dimensional space.
- Implement a $K$-nearest neighbor model of learning.
- Draw decision boundaries.
- Implement the $K$-means algorithm for clustering.

Dependencies: Chapter 1

You can think of prediction tasks as mapping inputs (course reviews) to outputs (course ratings). As you learned in the previous chapter, decomposing an input into a collection of features (eg., words that occur in the review) forms the useful abstraction for learning. Therefore, inputs are nothing more than lists of feature values. This suggests a **geometric view** of data, where we have one dimension for every feature. In this view, examples are points in a high-dimensional space.

Once we think of a data set as a collection of points in high dimensional space, we can start performing geometric operations on this data. For instance, suppose you need to predict whether Alice will like Algorithms. Perhaps we can try to find another student who is most "similar" to Alice, in terms of favorite courses. Say this student is Jeremy. If Jeremy liked Algorithms, then we might guess that Alice will as well. This is an example of a **nearest neighbor** model of learning. By inspecting this model, we'll see a completely different set of answers to the key learning questions we discovered in Chapter 1.

## 2.1 *From Data to Feature Vectors*

An example, for instance the data in Table **??** from the Appendix, is just a collection of feature values about that example. To a person, these features have meaning. One feature might count how many times the reviewer wrote "excellent" in a course review. Another might count the number of exclamation points. A third might tell us if any text is underlined in the review.

To a machine, the **features** themselves have no meaning. Only the **feature values**, and how they vary across examples, mean something to the machine. From this perspective, you can think about an example as being reprsented by a **feature vector** consisting of one "dimension" for each feature, where each dimenion is simply some real value.

Consider a review that said "excellent" three times, had one excla-

mation point and no underlined text. This could be represented by the feature vector $\langle 3, 1, 0 \rangle$. An almost identical review that happened to have underlined text would have the feature vector $\langle 3, 1, 1 \rangle$.

Note, here, that we have imposed the convention that for **binary features** (yes/no features), the corresponding feature values are 0 and 1, respectively. This was an arbitrary choice. We could have made them 0.92 and $-16.1$ if we wanted. But 0/1 is convenient and helps us interpret the feature values. When we discuss practical issues in Chapter 4, you will see other reasons why 0/1 is a good choice.

Figure 2.1 shows the data from Table **??** in three views. These three views are constructed by considering two features at a time in different pairs. In all cases, the plusses denote positive examples and the minuses denote negative examples. In some cases, the points fall on top of each other, which is why you cannot see 20 unique points in all figures.

The mapping from feature values to vectors is straighforward in the case of real valued feature (trivial) and binary features (mapped to zero or one). It is less clear what do do with **categorical features**. For example, if our goal is to identify whether an object in an image is a tomato, blueberry, cucumber or cockroach, we might want to know its color: is it RED, BLUE, GREEN or BLACK?

One option would be to map RED to a value of 0, BLUE to a value of 1, GREEN to a value of 2 and BLACK to a value of 3. The problem with this mapping is that it turns an unordered set (the set of colors) into an ordered set (the set $\{0, 1, 2, 3\}$). In itself, this is not necessarily a bad thing. But when we go to *use* these features, we will measure examples based on their distances to each other. By doing this mapping, we are essentially saying that RED and BLUE are more similar (distance of 1) than RED and BLACK (distance of 3). This is probably not what we want to say!

A solution is to turn a categorical feature that can take four different values (say: RED, BLUE, GREEN and BLACK) into four binary features (say: IsItRed?, IsItBlue?, IsItGreen? and IsItBlack?). In general, if we start from a categorical feature that takes $V$ values, we can map it to $V$-many binary indicator features.

With that, you should be able to take a data set and map each example to a feature vector through the following mapping:

- Real-valued features get copied directly.
- Binary features become 0 (for false) or 1 (for true).
- Categorical features with $V$ possible values get mapped to $V$-many binary indicator features.

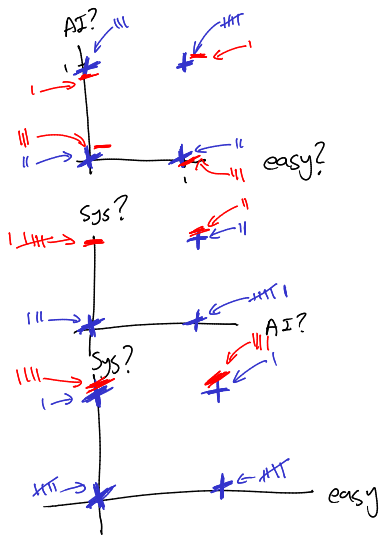



Figure 2.1: A figure showing projections of data in two dimension in three ways – see text. Top: horizontal axis corresponds to the first feature (TODO) and the vertical axis corresponds to the second feature (TODO); Middle: horizonal is second feature and vertical is third; Bottom: horizonal is first and vertical is third.

? Match the example ids from Table **??** with the points in Figure 2.1.

? The computer scientist in you might be saying: actually we could map it to $\log_2 K$-many binary features! Is this a good idea or not?

After this mapping, you can think of a single example as a **vector** in a high-dimensional **feature space**. If you have $D$-many features (after expanding categorical features), then this **feature vector** will have $D$-many components. We will denote feature vectors as $\boldsymbol{x} = \langle x_1, x_2, \ldots, x_D \rangle$, so that $x_d$ denotes the value of the $d$th feature of $\boldsymbol{x}$. Since these are vectors with real-valued components in $D$-dimensions, we say that they belong to the space $\mathbb{R}^D$.

For $D = 2$, our feature vectors are just points in the plane, like in Figure 2.1. For $D = 3$ this is three dimensional space. For $D > 3$ it becomes quite hard to visualize. (You should resist the temptation to think of $D = 4$ as "time" – this will just make things confusing.) Unfortunately, for the sorts of problems you will encounter in machine learning, $D \approx 20$ is considered "low dimensional," $D \approx 1000$ is "medium dimensional" and $D \approx 100000$ is "high dimensional."

? Can you think of problems (perhaps ones already mentioned in this book!) that are low dimensional? That are medium dimensional? That are high dimensional?

## 2.2 K-Nearest Neighbors

The biggest advantage to thinking of examples as vectors in a high dimensional space is that it allows us to apply geometric concepts to machine learning. For instance, one of the most basic things that one can do in a vector space is compute **distances**. In two-dimensional space, the distance between $\langle 2, 3 \rangle$ and $\langle 6, 1 \rangle$ is given by $\sqrt{(2-6)^2 + (3-1)^2} = \sqrt{18} \approx 4.24$. In general, in $D$-dimensional space, the **Euclidean distance** between vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ is given by Eq (2.1) (see Figure 2.2 for geometric intuition in three dimensions):

$$d(\boldsymbol{a}, \boldsymbol{b}) = \left[ \sum_{d=1}^{D} (a_d - b_d)^2 \right]^{\frac{1}{2}} \tag{2.1}$$

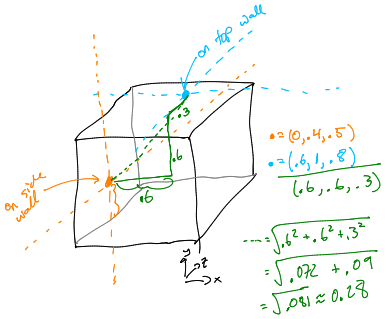



Figure 2.2: A figure showing Euclidean distance in three dimensions

? Verify that $d$ from Eq (2.1) gives the same result (4.24) for the previous computation.

Now that you have access to distances between examples, you can start thinking about what it means to learn again. Consider Figure 2.3. We have a collection of training data consisting of positive examples and negative examples. There is a test point marked by a question mark. Your job is to guess the correct label for that point.

Most likely, you decided that the label of this test point is positive. One reason why you might have thought that is that you believe that the label for an example should be similar to the label of nearby points. This is an example of a new form of **inductive bias**.

The **nearest neighbor** classifier is build upon this insight. In comparison to decision trees, the algorithm is ridiculously simple. At training time, we simply store the entire training set. At test time, we get a test example $\hat{\boldsymbol{x}}$. To predict its label, we find the training example $\boldsymbol{x}$ that is most similar to $\hat{\boldsymbol{x}}$. In particular, we find the training

**Algorithm 3** KNN-PREDICT($\mathbf{D}$, $K$, $\hat{x}$)

```
S ← [ ]
for n = 1 to N do
    S ← S ⊕ ⟨d(x_n, x̂), n⟩          // store distance to training example n
end for
S ← SORT(S)                          // put lowest-distance objects first
ŷ ← 0
for k = 1 to K do
    ⟨dist,n⟩ ← S_k                   // n this is the kth closest data point
    ŷ ← ŷ + y_n                      // vote according to the label for the nth training point
end for
return SIGN(ŷ)                       // return +1 if ŷ > 0 and −1 if ŷ < 0
```

example $\boldsymbol{x}$ that *minimizes* $d(\boldsymbol{x}, \hat{\boldsymbol{x}})$. Since $\boldsymbol{x}$ is a training example, it has a corresponding label, $y$. We predict that the label of $\hat{\boldsymbol{x}}$ is also $y$.

Despite its simplicity, this nearest neighbor classifier is incredibly effective. (Some might say *frustratingly* effective.) However, it is particularly prone to overfitting label noise. Consider the data in Figure 2.4. You would probably want to label the test point positive. Unfortunately, it's nearest neighbor happens to be negative. Since the nearest neighbor algorithm only looks at the *single* nearest neighbor, it cannot consider the "preponderance of evidence" that this point should probably actually be a positive example. It will make an unnecessary error.

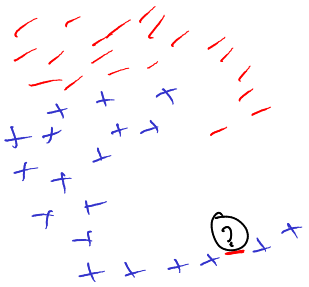


Figure 2.4: A figure showing an easy NN classification problem where the test point is a ? and should be positive, but its NN is actually a negative point that's noisy.

A solution to this problem is to consider more than just the single nearest neighbor when making a classification decision. We can consider the **$K$-nearest neighbors** and let them **vote** on the correct class for this test point. If you consider the 3-nearest neighbors of the test point in Figure 2.4, you will see that two of them are positive and one is negative. Through voting, positive would win.

? Why is it a good idea to use an odd number for $K$?

The full algorithm for $K$-nearest neighbor classification is given in Algorithm 2.2. Note that there actually is no "training" phase for $K$-nearest neighbors. In this algorithm we have introduced five new conventions:

1. The training data is denoted by $\mathbf{D}$.
2. We assume that there are $N$-many training examples.
3. These examples are pairs $(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_N, y_N)$. (Warning: do not confuse $\boldsymbol{x}_n$, the $n$th training example, with $x_d$, the $d$th feature for example $\boldsymbol{x}$.)
4. We use [ ]to denote an empty list and $\oplus \cdot$ to append $\cdot$ to that list.
5. Our prediction on $\hat{\boldsymbol{x}}$ is called $\hat{y}$.

The first step in this algorithm is to compute distances from the test point to all training points (lines 2-4). The data points are then sorted according to distance. We then apply a clever trick of *summing* the class labels for each of the $K$ nearest neighbors (lines 6-10) and using the SIGN of this sum as our prediction.

The big question, of course, is how to choose $K$. As we've seen, with $K = 1$, we run the risk of overfitting. On the other hand, if $K$ is large (for instance, $K = N$), then KNN-PREDICT will always predict the majority class. Clearly that is underfitting. So, $K$ is a hyperparameter of the KNN algorithm that allows us to trade-off between overfitting (small value of $K$) and underfitting (large value of $K$).

? Why is the sign of the sum computed in lines 2-4 the same as the majority vote of the associated training examples?

One aspect of **inductive bias** that we've seen for KNN is that it assumes that nearby points should have the same label. Another aspect, which is quite different from decision trees, is that all features are equally important! Recall that for decision trees, the key question was *which features are most useful for classification?* The whole learning algorithm for a decision tree hinged on finding a small set of good features. This is all thrown away in KNN classifiers: every feature is used, and they are all used the same amount. This means that if you have data with only a few relevant features and lots of irrelevant features, KNN is likely to do poorly.

? Why can't you simply pick the value of $K$ that does best on the training data? In other words, why do we have to treat it like a hyperparameter rather than just a parameter.

A related issue with KNN is **feature scale**. Suppose that we are trying to classify whether some object is a ski or a snowboard (see Figure 2.5). We are given two features about this data: the width and height. As is standard in skiing, width is measured in millimeters and height is measured in centimeters. Since there are only two features, we can actually plot the entire training set; see Figure 2.6 where ski is the positive class. Based on this data, you might guess that a KNN classifier would do well.

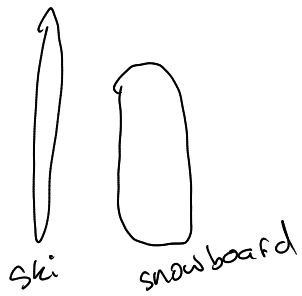



Figure 2.5: A figure of a ski and snowboard with width (mm) and height (cm).

Suppose, however, that our measurement of the width was computed in millimeters (instead of centimeters). This yields the data shown in Figure 2.7. Since the width values are now tiny, in comparison to the height values, a KNN classifier will effectively *ignore* the width values and classify almost purely based on height. The predicted class for the displayed test point had changed because of this feature scaling.

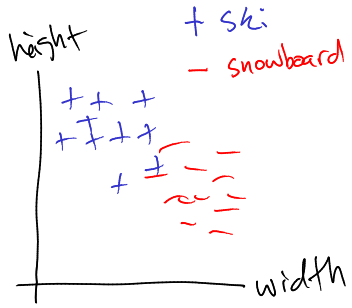
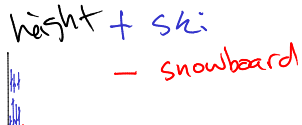



Figure 2.6: Classification data for ski vs snowboard in 2d

We will discuss feature scaling more in Chapter 4. For now, it is just important to keep in mind that KNN does not have the power to decide which features are important.



## 2.3 *Decision Boundaries*

The standard way that we've been thinking about learning algorithms up to now is in the *query model*. Based on training data, you learn something. I then give you a query example and you have to guess it's label.

An alternative, less passive, way to think about a learned model is to ask: what sort of test examples will it classify as positive, and what sort will it classify as negative. In Figure 2.9, we have a set of training data. The background of the image is colored blue in regions that *would* be classified as positive (if a query were issued there) and colored red in regions that *would* be classified as negative. This coloring is based on a 1-nearest neighbor classifier.

In Figure 2.9, there is a solid line separating the positive regions from the negative regions. This line is called the **decision boundary** for this classifier. It is the line with positive land on one side and negative land on the other side.

Decision boundaries are useful ways to visualize the **complexity** of a learned model. Intuitively, a learned model with a decision boundary that is really jagged (like the coastline of Norway) is really complex and prone to overfitting. A learned model with a decision boundary that is really simple (like the bounary between Arizona and Utah) is potentially underfit. In Figure **??**, you can see the decision boundaries for KNN models with $K \in \{1, 3, 5, 7\}$. As you can see, the boundaries become simpler and simpler as $K$ gets bigger.

Now that you know about decision boundaries, it is natural to ask: what do decision boundaries for decision trees look like? In order to answer this question, we have to be a bit more formal about how to build a decision tree on real-valued features. (Remember that the algorithm you learned in the previous chapter implicitly assumed *binary* feature values.) The idea is to allow the decision tree to ask questions of the form: "is the value of feature 5 greater than 0.2?" That is, for real-valued features, the decision tree nodes are parameterized by a feature and a threshold for that feature. An example decision tree for classifying skis versus snowboards is shown in Figure 2.10.

Now that a decision tree can handle feature vectors, we can talk about decision boundaries. By example, the decision boundary for the decision tree in Figure 2.10 is shown in Figure 2.11. In the figure, space is first split in half according to the first query along one axis. Then, depending on which half of the space you you look at, it is either split again along the other axis, or simple classified.

Figure 2.11 is a good visualization of decision boundaries for decision trees in general. Their decision boundaries are axis-aligned

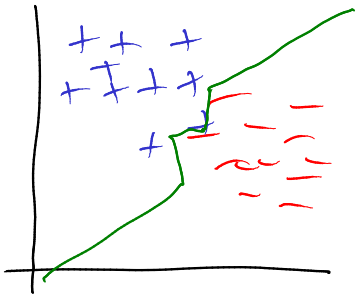


Figure 2.8: decision boundary for 1nn.

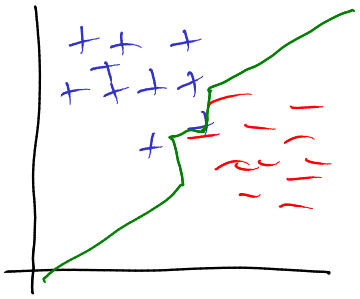


Figure 2.9: decision boundary for knn with k=3.

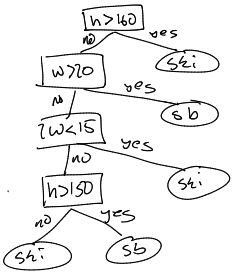



Figure 2.10: decision tree for ski vs. snowboard

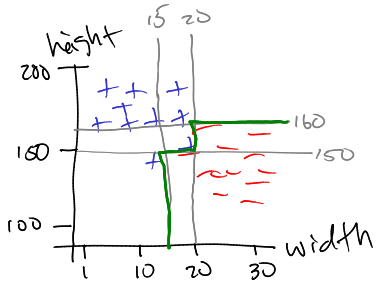



Figure 2.11: decision boundary for dt in previous figure

cuts. The cuts must be axis-aligned because nodes can only query on a single feature at a time. In this case, since the decision tree was so shallow, the decision boundary was relatively simple.

> ? What sort of data might yield a very simple decision boundary with a decision tree and very complex decision boundary with 1-nearest neighbor? What about the other way around?

## 2.4 K-Means Clustering

Up through this point, you have learned all about supervised learning (in particular, binary classification). As another example of the use of geometric intuitions and data, we are going to temporarily consider an **unsupervised learning** problem. In unsupervised learning, our data consists *only* of examples $\boldsymbol{x}_n$ and does *not* contain corresponding labels. Your job is to make sense of this data, even though no one has provided you with correct labels. The particular notion of "making sense of" that we will talk about now is the **clustering** task.

Consider the data shown in Figure 2.12. Since this is unsupervised learning and we do not have access to labels, the data points are simply drawn as black dots. Your job is to split this data set into three clusters. That is, you should label each data point as A, B or C in whatever way you want.

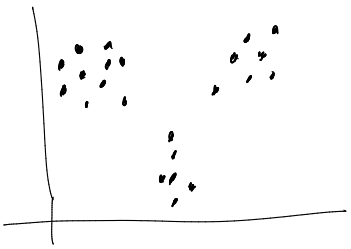


Figure 2.12: simple clustering data... clusters in UL, UR and BC.

For this data set, it's pretty clear what you should do. You probably labeled the upper-left set of points A, the upper-right set of points B and the bottom set of points C. Or perhaps you permuted these labels. But chances are your clusters were the same as mine.

The *K*-means clustering algorithm is a particularly simple and effective approach to producing clusters on data like you see in Figure 2.12. The idea is to represent each cluster by it's cluster center. Given cluster centers, we can simply assign each point to its nearest center. Similarly, if we know the assignment of points to clusters, we can compute the centers. This introduces a chicken-and-egg problem. If we knew the clusters, we could compute the centers. If we knew the centers, we could compute the clusters. But we don't know either.

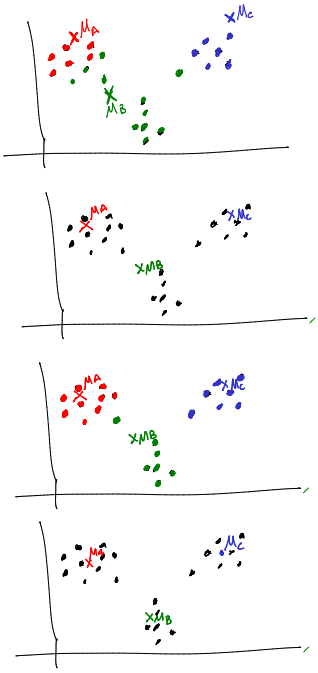


Figure 2.13: first few iterations of k-means running on previous data set

The general computer science answer to chicken-and-egg problems is **iteration**. We will start with a guess of the cluster centers. Based on that guess, we will assign each data point to its closest center. Given these new assignments, we can recompute the cluster centers. We repeat this process until clusters stop moving. The first few iterations of the *K*-means algorithm are shown in Figure 2.13. In this example, the clusters converge very quickly.

Algorithm 2.4 spells out the *K*-means clustering algorithm in detail. The cluster centers are initialized randomly. In line 6, data point $\boldsymbol{x}_n$ is compared against each cluster center $\boldsymbol{\mu}_k$. It is assigned to cluster $k$ if $k$ is the center with the smallest distance. (That is the "argmin" step.) The variable $z_n$ stores the assignment (a value from 1 to $K$) of example $n$. In lines 8-12, the cluster centers are re-computed. First, $\mathbf{X}_k$

**Algorithm 4** K-MEANS($\mathbf{D}$, $K$)

```
for k = 1 to K do
    μ_k ← some random location          // randomly initialize mean for kth cluster
end for
repeat
    for n = 1 to N do
        z_n ← argmin_k ||μ_k − x_n||    // assign example n to closest center
    end for
    for k = 1 to K do
        X_k ← { x_n : z_n = k }         // points assigned to cluster k
        μ_k ← MEAN(X_k)                 // re-estimate mean of cluster k
    end for
until μs stop changing
return z                                // return cluster assignments
```

**MATH REVIEW | VECTOR ARITHMETIC, NORMS AND MEANS**

define vector addition, scalar addition, subtraction, scalar multiplication and norms. define mean.

Figure 2.14:

stores all examples that have been assigned to cluster $k$. The center of cluster $k$, $\boldsymbol{\mu}_k$ is then computed as the mean of the points assigned to it. This process repeats until the means converge.

An obvious question about this algorithm is: does it converge? A second question is: how long does it take to converge. The first question is actually easy to answer. Yes, it does. And in practice, it usually converges quite quickly (usually fewer than 20 iterations). In Chapter 13, we will actually *prove* that it converges. The question of how long it takes to converge is actually a really interesting question. Even though the $K$-means algorithm dates back to the mid 1950s, the best known convergence rates were *terrible* for a long time. Here, terrible means exponential in the number of data points. This was a sad situation because empirically we knew that it converged very quickly. New algorithm analysis techniques called "smoothed analysis" were invented in 2001 and have been used to show very fast convergence for $K$-means (among other algorithms). These techniques are well beyond the scope of this book (and this author!) but suffice it to say that $K$-means is fast in practice and is provably fast in theory.

It is important to note that although $K$-means is guaranteed to converge and guaranteed to converge quickly, it is *not* guaranteed to converge to the "right answer." The key problem with unsupervised learning is that we have no way of knowing what the "right answer" is. Convergence to a bad solution is usually due to poor initialization. For example, poor initialization in the data set from before yields convergence like that seen in Figure **??**. As you can see, the algorithm

*has* converged. It has just converged to something less than satisfactory.

? What is the difference between unsupervised and supervised learning that means that we know what the "right answer" is for supervised learning but not for unsupervised learning?

## 2.5 Warning: High Dimensions are Scary

Visualizing one hundred dimensional space is incredibly difficult for humans. After huge amounts of training, some people have reported that they can visualize four dimensional space in their heads. But beyond that seems impossible.[1]

In addition to being hard to visualize, there are at least two additional problems in high dimensions, both refered to as **the curse of dimensionality**. One is computational, the other is mathematical.

From a computational perspective, consider the following problem. For $K$-nearest neighbors, the speed of prediction is slow for a very large data set. At the very least you have to look at every training example every time you want to make a prediction. To speed things up you might want to create an *indexing* data structure. You can break the plane up into a grid like that shown in Figure **??**. Now, when the test point comes in, you can quickly identify the grid cell in which it lies. Now, instead of considering *all* training points, you can limit yourself to training points *in that grid cell* (and perhaps the neighboring cells). This can potentially lead to huge computational savings.

In two dimensions, this procedure is effective. If we want to break space up into a grid whose cells are $0.2 \times 0.2$, we can clearly do this with 25 grid cells in two dimensions (assuming the range of the features is 0 to 1 for simplicity). In three dimensions, we'll need $125 = 5 \times 5 \times 5$ grid cells. In four dimensions, we'll need 625. By the time we get to "low dimensional" data in 20 dimensions, we'll need $95,367,431,640,625$ grid cells (that's 95 trillion, which is about 6 to 7 times the US national dept as of January 2011). So if you're in 20 dimensions, this gridding technique will only be useful if you have at least 95 trillion training examples.

For "medium dimensional" data (approximately 1000) dimesions, the number of grid cells is a 9 followed by 698 numbers before the decimal point. For comparison, the number of atoms in the universe is approximately 1 followed by 80 zeros. So even if each atom yielding a googul training examples, we'd still have far fewer examples than grid cells. For "high dimensional" data (approximately 100000) dimensions, we have a 1 followed by just under $70,000$ zeros. Far too big a number to even really comprehend.

Suffice it to say that for even moderately high dimensions, the amount of computation involved in these problems is enormous.

In addition to the computational difficulties of working in high

[1] If you want to try to get an intuitive sense of what four dimensions looks like, I highly recommend the short 1884 book *Flatland: A Romance of Many Dimensions* by Edwin Abbott Abbott. You can even read it online at gutenberg.org/ebooks/201.

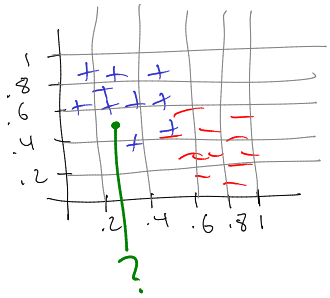



Figure 2.15: 2d knn with an overlaid grid, cell with test point highlighted

? How does the above analysis relate to the number of data points you would need to fill out a full decision tree with $D$-many features? What does this say about the importance of shallow trees?

dimensions, there are a large number of strange mathematical occurances there. In particular, many of your intuitions that you've built up from working in two and three dimensions just do not carry over to high dimensions. We will consider two effects, but there are countless others. The first is that high dimensional spheres look more like porcupines than like balls.[2] The second is that distances between points in high dimensions are all approximately the same.

Let's start in two dimensions as in Figure 2.16. We'll start with four green spheres, each of radius one and each touching exactly two other green spheres. (Remember than in two dimensions a "sphere" is just a "circle.") We'll place a red sphere in the middle so that it touches all four green spheres. We can easily compute the radius of this small sphere. The pythagorean theorem says that $1^2 + 1^2 = (1 + r)^2$, so solving for $r$ we get $r = \sqrt{2} - 1 \approx 0.41$. Thus, by calculation, the blue sphere lies entirely within the cube (cube = square) that contains the grey spheres. (Yes, this is also obvious from the picture, but perhaps you can see where this is going.)

Now we can do the same experiment in three dimensions, as shown in Figure 2.17. Again, we can use the pythagorean theorem to compute the radius of the blue sphere. Now, we get $1^2 + 1^2 + 1^2 = (1 + r)^2$, so $r = \sqrt{3} - 1 \approx 0.73$. This is still entirely enclosed in the cube of width four that holds all eight grey spheres.

At this point it becomes difficult to produce figures, so you'll have to apply your imagination. In four dimensions, we would have 16 green spheres (called **hyperspheres**), each of radius one. They would still be inside a cube (called a **hypercube**) of width four. The blue hypersphere would have radius $r = \sqrt{4} - 1 = 1$. Continuing to five dimensions, the blue hypersphere embedded in 256 green hyperspheres would have radius $r = \sqrt{5} - 1 \approx 1.23$ and so on.

In general, in $D$-dimensional space, there will be $2^D$ green hyperspheres of radius one. Each green hypersphere will touch exactly $n$-many other hyperspheres. The blue hyperspheres in the middle will touch them all and will have radius $r = \sqrt{D} - 1$.

Think about this for a moment. As the number of dimensions grows, the radius of the blue hypersphere *grows without bound!*. For example, in 9-dimensional the radius of the blue hypersphere is now $\sqrt{9} - 1 = 2$. But with a radius of two, the blue hypersphere is now "squeezing" between the green hypersphere and *touching* the edges of the hypercube. In 10 dimensional space, the radius is approximately 2.16 and it pokes outside the cube.

This is why we say that high dimensional spheres look like porcupines and not balls (see Figure 2.18). The moral of this story from a *machine learning* perspective is that intuitions you have about space might not carry over to high dimensions. For example, what you

[2] This results was related to me by Mark Reid, who heard about it from Marcus Hutter.

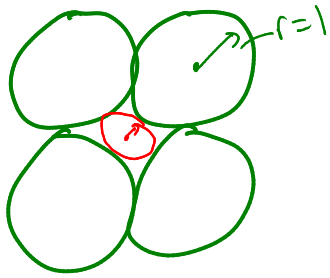



Figure 2.16: 2d spheres in spheres

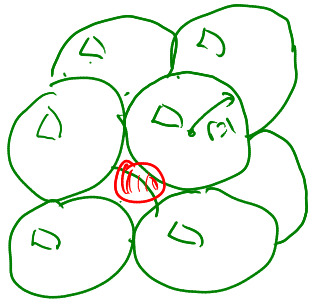


Figure 2.17: 3d spheres in spheres

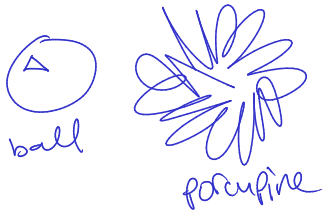



Figure 2.18: porcupine versus ball

think looks like a "round" cluster in two or three dimensions, might not look so "round" in high dimensions.

The second strange fact we will consider has to do with the distances between points in high dimensions. We start by considering random points in one dimension. That is, we generate a fake data set consisting of 100 random points between zero and one. We can do the same in two dimensions and in three dimensions. See Figure 2.19 for data distributed uniformly on the **unit hypercube** in different dimensions.

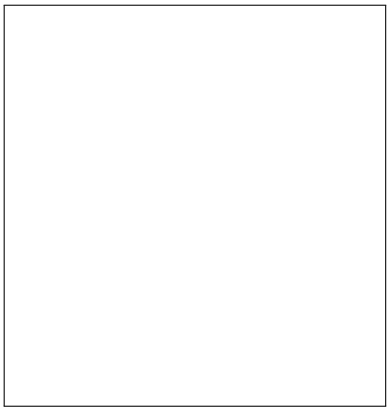


Figure 2.19: `knn:uniform`: 100 uniform random points in 1, 2 and 3 dimensions

Now, pick two of these points at random and compute the distance between them. Repeat this process for all pairs of points and average the results. For the data shown in Figure 2.19, the average distance between points in one dimension is *TODO*; in two dimensions is *TODO*; and in three dimensions is *TODO*.

You can actually compute these value analytically. Write $\mathcal{U}ni_D$ for the uniform distribution in $D$ dimensions. The quantity we are interested in computing is:

$$avgDist(D) = \mathbb{E}_{\boldsymbol{a}\sim\mathcal{U}ni_D}\Big[\mathbb{E}_{\boldsymbol{b}\sim\mathcal{U}ni_D}\Big[\,||\boldsymbol{a}-\boldsymbol{b}||\,\Big]\Big] \tag{2.2}$$

We can actually compute this in closed form (see Exercise **??** for a bit of calculus refresher) and arrive at $avgDist(D) = TODO$. Consider what happens as $D \to \infty$. As $D$ grows, the average distance between points in $D$ dimensions goes to 1! In other words, *all* distances become about the same in high dimensions.

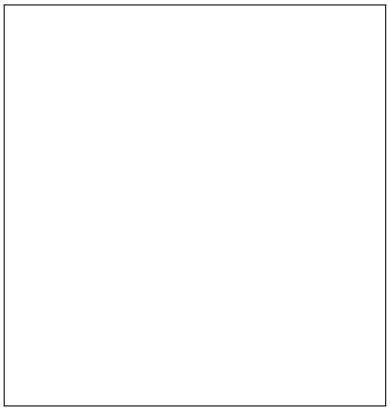


Figure 2.20: `knn:uniformhist`: histogram of distances in D=1,2,3,10,20,100

When I first saw and re-proved this result, I was skeptical, as I imagine you are. So I implemented it. In Figure 2.20 you can see the results. This presents a *histogram* of distances between random points in $D$ dimensions for $D \in \{1, 2, 3, 10, 20, 100\}$. As you can see, all of these distances begin to concentrate around 1, even for "medium dimension" problems.

You should now be terrified: the only bit of information that KNN gets is distances. And you've just seen that in moderately high dimensions, all distances becomes equal. So then isn't is the case that KNN simply cannot work?

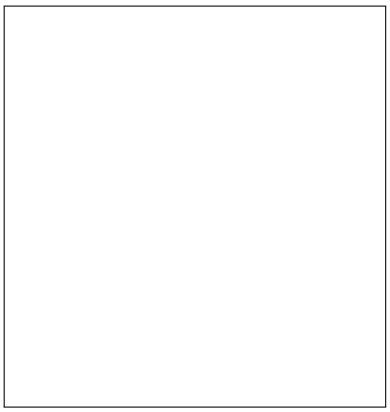


Figure 2.21: `knn:mnist`: histogram of distances in multiple D for mnist

The answer has to be no. The reason is that the data that we get is *not* uniformly distributed over the unit hypercube. We can see this by looking at two real-world data sets. The first is an image data set of hand-written digits (zero through nine); see Section **??**. Although this data is originally in 256 dimensions (16 pixels by 16 pixels), we can artifically reduce the dimensionality of this data. In Figure 2.21 you can see the histogram of average distances between points in this data at a number of dimensions. Figure 2.22 shows the same sort of histogram for a text data set (Section **??**.

As you can see from these histograms, distances have *not* con-

centrated around a single value. This is very good news: it means that there is hope for learning algorithms to work! Nevertheless, the moral is that high dimensions are weird.

## 2.6 Extensions to KNN

There are several fundamental problems with KNN classifiers. First, some neighbors might be "better" than others. Second, test-time performance scales badly as your number of training examples increases. Third, it treats each dimension independently. We will not address the third issue, as it has not really been solved (though it makes a great thought question!).

Regarding neighborliness, consider Figure 2.23. Using $K = 5$ nearest neighbors, the test point would be classified as positive. However, we might actually believe that it should be classified negative because the two negative neighbors are *much* closer than the three positive neighbors.

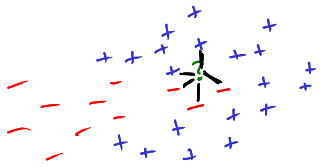


Figure 2.23: data set with 5nn, test point closest to two negatives, then to three far positives

There are at least two ways of addressing this issue. The first is the **$\epsilon$-ball** solution. Instead of connecting each data point to some fixed number ($K$) of nearest neighbors, we simply connect it to *all* neighbors that fall within some ball of radius $\epsilon$. Then, the majority class of all the points in the $\epsilon$ ball wins. In the case of a tie, you would have to either guess, or report the majority class. Figure 2.24 shows an $\epsilon$ ball around the test point that happens to yield the proper classification.

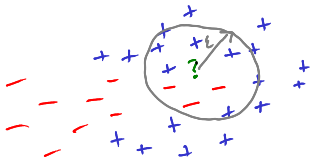


Figure 2.24: same as previous with $\epsilon$ ball

When using $\epsilon$-ball nearest neighbors rather than KNN, the hyperparameter changes from $K$ to $\epsilon$. You would need to set it in the same way as you would for KNN.

An alternative to the $\epsilon$-ball solution is to do **weighted nearest neighbors**. The idea here is to still consider the $K$-nearest neighbors of a test point, but give them uneven votes. Closer points get more vote than further points. When classifying a point $\hat{\boldsymbol{x}}$, the usual strategy is to give a training point $\boldsymbol{x}_n$ a vote that decays exponentially in the distance between $\hat{\boldsymbol{x}}$ and $\boldsymbol{x}_n$. Mathematically, the vote that neighbor $n$ gets is:

$$\exp\left[-\frac{1}{2}||\hat{\boldsymbol{x}} - \boldsymbol{x}_n||^2\right] \tag{2.3}$$

Thus, nearby points get a vote very close to 1 and far away points get a vote very close to 0. The overall prediction is positive if the sum of votes from positive neighbors outweighs the sum of votes from negative neighbors.

? One issue with $\epsilon$-balls is that the $\epsilon$-ball for some test point might be empty. How would you handle this?

The second issue with KNN is scaling. To predict the label of a single test point, we need to find the $K$ nearest neighbors of that

? Could you combine the $\epsilon$-ball idea with the weighted voting idea? Does it make sense, or does one idea seem to trump the other?

test point in the training data. With a standard implementation, this will take $\mathcal{O}(ND + K \log K)$ time[3]. For very large data sets, this is impractical.

A first attempt to speed up the computation is to represent each class by a *representative.* A natural choice for a representative would be the mean. We would collapse all positive examples down to their mean, and all negative examples down to their mean. We could then just run 1-nearest neighbor and check whether a test point is closer to the mean of the positive points or the mean of the negative points. Figure 2.25 shows an example in which this would probably work well, and an example in which this would probably work poorly. The problem is that collapsing each class to its mean is too aggressive.

A less aggressive approach is to make use of the $K$-means algorithm for clustering. You can cluster the positive examples into $L$ clusters (we are using $L$ to avoid variable overloading!) and then cluster the negative examples into $L$ separate clusters. This is shown in Figure 2.26 with $L = 2$. Instead of storing the entire data set, you would only store the *means* of the $L$ positive clusters and the means of the $L$ negative clusters. At test time, you would run the $K$-nearest neighbors algorithm against these *means* rather than against the full training set. This leads a a much faster runtime of just $\mathcal{O}(LD + K \log K)$, which is probably dominated by $LD$.

[3] The $ND$ term comes from computing distances between the test point and all training points. The $K \log K$ term comes from finding the $K$ smallest values in the list of distances, using a median-finding algorithm. Of course, $ND$ almost always dominates $K \log K$ in practice.

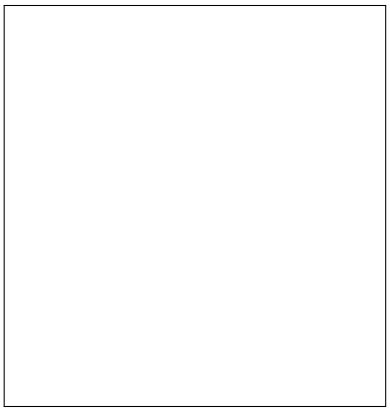


Figure 2.25: `knn:collapse`: two figures of points collapsed to mean, one with good results and one with dire results

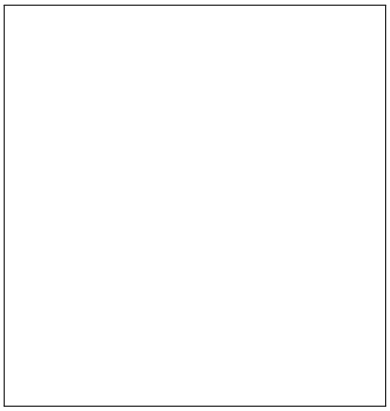


Figure 2.26: `knn:collapse2`: data from previous bad case collapsed into L=2 cluster and test point classified based on means and 1-nn

? Clustering of classes was introduced as a way of making things faster. Will it make things worse, or could it help?

## 2.7 Exercises

**Exercise 2.1. TODO...**

# 3 | The Perceptron

**Learning Objectives:**

- Describe the biological motivation behind the perceptron.
- Classify learning algorithms based on whether they are error-driven or not.
- Implement the perceptron algorithm for binary classification.
- Draw perceptron weight vectors and the corresponding decision boundaries in two dimensions.
- Contrast the decision boundaries of decision trees, nearest neighbor algorithms and perceptrons.
- Compute the margin of a given weight vector on a given data set.

So far, you've seen two types of learning models: in decision trees, only a small number of features are used to make decisions; in nearest neighbor algorithms, all features are used equally. Neither of these extremes is always desirable. In some problems, we might want to use most of the features, but use some more than others.

In this chapter, we'll discuss the **perceptron** algorithm for learning **weights** for features. As we'll see, learning weights for features amounts to learning a **hyperplane** classifier: that is, basically a division of space into two halves by a straight line, where one half is "positive" and one half is "negative." In this sense, the perceptron can be seen as explicitly finding a good **linear decision boundary**.

Dependencies: Chapter 1, Chapter 2

## 3.1 *Bio-inspired Learning*

Folk biology tells us that our brains are made up of a bunch of little units, called **neurons**, that send electrical signals to one another. The *rate* of firing tells us how "activated" a neuron is. A single neuron, like that shown in Figure 3.1 might have three incoming neurons. These incoming neurons are firing at different rates (i.e., have different **activations**). Based on how much these incoming neurons are firing, and how "strong" the neural connections are, our main neuron will "decide" how strongly it wants to fire. And so on through the whole brain. Learning in the brain happens by neurons becoming connected to other neurons, and the strengths of connections adapting over time.

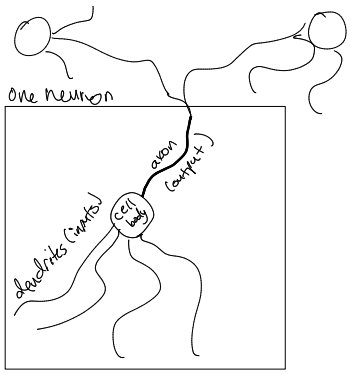



Figure 3.1: a picture of a neuron

The real biological world is much more complicated than this. However, our goal isn't to build a brain, but to simply be *inspired* by how they work. We are going to think of our learning algorithm as a *single* neuron. It receives input from $D$-many other neurons, one for each input feature. The strength of these inputs are the feature values. This is shown schematically in Figure **??**. Each incoming connection has a weight and the neuron simply sums up all the weighted inputs. Based on this sum, it decides whether to "fire" or

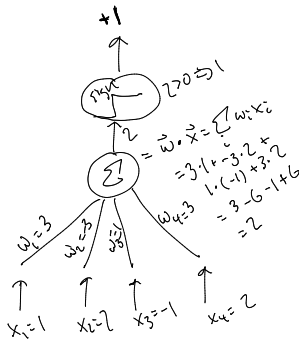



Figure 3.2: figure showing feature vector and weight vector and products and sum

not. Firing is interpreted as being a positive example and not firing is interpreted as being a negative example. In particular, if the weighted sum is positive, it "fires" and otherwise it doesn't fire. This is shown diagramatically in Figure 3.2.

Mathematically, an input vector $\boldsymbol{x} = \langle x_1, x_2, \ldots, x_D \rangle$ arrives. The neuron stores $D$-many weights, $w_1, w_2, \ldots, w_D$. The neuron computes the sum:

$$a = \sum_{d=1}^{D} w_d x_d \tag{3.1}$$

to determine it's amount of "activation." If this activiation is positive (i.e., $a > 0$) it predicts that this example is a positive example. Otherwise it predicts a negative example.

The weights of this neuron are fairly easy to interpret. Suppose that a feature, for instance "is this a System's class?" gets a zero weight. Then the activation is the same regardless of the value of this feature. So features with zero weight are ignored. Features with positive weights are indicative of positive examples because they cause the activation to increase. Features with negative weights are indicative of negative examples because they cause the activiation to decrease.

? What would happen if we encoded binary features like "is this a System's class" as no=0 and yes=−1 (rather than the standard no=0 and yes=+1)?

It is often convenient to have a non-zero **threshold**. In other words, we might want to predict positive if $a > \theta$ for some value $\theta$. The way that is most convenient to achieve this is to introduce a **bias** term into the neuron, so that the activation is always increased by some fixed value $b$. Thus, we compute:

$$a = \left[ \sum_{d=1}^{D} w_d x_d \right] + b \tag{3.2}$$

? If you wanted the activation threshold to be $a > \theta$ instead of $a > 0$, what value would $b$ have to be?

This is the complete neural model of learning. The model is parameterized by $D$-many weights, $w_1, w_2, \ldots, w_D$, and a single scalar bias value $b$.

## 3.2 *Error-Driven Updating: The Perceptron Algorithm*

**VIGNETTE: THE HISTORY OF THE PERCEPTRON**

todo

The **perceptron** is a classic learning algorithm for the neural model of learning. Like $K$-nearest neighbors, it is one of those frustrating algorithms that is incredibly simple and yet works amazingly well, for some types of problems.

**Algorithm 5** PERCEPTRONTRAIN(**D**, *MaxIter*)

```
1:  w_d ← 0, for all d = 1 ... D                          // initialize weights
2:  b ← 0                                                  // initialize bias
3:  for iter = 1 ... MaxIter do
4:      for all (x,y) ∈ D do
5:          a ← Σ_{d=1}^{D} w_d x_d + b                    // compute activation for this example
6:          if ya ≤ 0 then
7:              w_d ← w_d + y x_d, for all d = 1 ... D     // update weights
8:              b ← b + y                                  // update bias
9:          end if
10:     end for
11: end for
12: return w_0, w_1, ..., w_D, b
```

**Algorithm 6** PERCEPTRONTEST($w_0$, $w_1$, ..., $w_D$, $b$, $\hat{\boldsymbol{x}}$)

```
1:  a ← Σ_{d=1}^{D} w_d x̂_d + b                            // compute activation for the test example
2:  return SIGN(a)
```

The algorithm is actually quite different than either the decision tree algorithm or the KNN algorithm. First, it is **online**. This means that instead of considering the *entire data set* at the same time, it only ever looks at one example. It processes that example and then goes on to the next one. Second, it is **error driven**. This means that, so long as it is doing well, it doesn't bother updating its parameters.

The algorithm maintains a "guess" at good parameters (weights and bias) as it runs. It processes one example at a time. For a given example, it makes a prediction. It checks to see if this prediction is correct (recall that this is *training data*, so we have access to true labels). If the prediction is correct, it does nothing. Only when the prediction is incorrect does it change its parameters, and it changes them in such a way that it would do better on this example next time around. It then goes on to the next example. Once it hits the last example in the training set, it loops back around for a specified number of iterations.

The training algorithm for the perceptron is shown in Algorithm 3.2 and the corresponding prediction algorithm is shown in Algorithm 3.2. There is one "trick" in the training algorithm, which probably seems silly, but will be useful later. It is in line 6, when we check to see if we want to make an update or not. We want to make an update if the current prediction (just SIGN($a$)) is incorrect. The trick is to multiply the true label $y$ by the activation $a$ and compare this against zero. Since the label $y$ is either $+1$ or $-1$, you just need to realize that $ya$ is positive whenever $a$ and $y$ have the same sign. In other words, the product $ya$ is positive if the current prediction is correct.

? It is very very important to check $ya \leq 0$ rather than $ya < 0$. Why?

The particular form of update for the perceptron is quite simple. The weight $w_d$ is increased by $yx_d$ and the bias is increased by $y$. The goal of the update is to adjust the parameters so that they are "better" for the current example. In other words, if we saw this example twice in a row, we should do a better job the second time around.

To see why this particular update achieves this, consider the following scenario. We have some current set of parameters $w_1, \ldots, w_D, b$. We observe an example $(\boldsymbol{x}, y)$. For simplicity, suppose this is a positive example, so $y = +1$. We compute an activation $a$, and make an error. Namely, $a < 0$. We now update our weights and bias. Let's call the new weights $w'_1, \ldots, w'_D, b'$. Suppose we observe the same example again and need to compute a new activation $a'$. We proceed by a little algebra:

$$a' = \sum_{d=1}^{D} w'_d x_d + b' \tag{3.3}$$

$$= \sum_{d=1}^{D} (w_d + x_d) x_d + (b + 1) \tag{3.4}$$

$$= \sum_{d=1}^{D} w_d x_d + b + \sum_{d=1}^{D} x_d x_d + 1 \tag{3.5}$$

$$= a + \sum_{d=1}^{D} x_d^2 + 1 \quad > \quad a \tag{3.6}$$

So the difference between the old activation $a$ and the new activation $a'$ is $\sum_d x_d^2 + 1$. But $x_d^2 \geq 0$, since it's squared. So this value is *always* at least one. Thus, the new activation is always at least the old activation plus one. Since this was a positive example, we have successfully moved the activation in the proper direction. (Though note that there's no guarantee that we will correctly classify this point the second, third or even fourth time around!)

The only **hyperparameter** of the perceptron algorithm is *MaxIter*, the number of passes to make over the training data. If we make many many passes over the training data, then the algorithm is likely to overfit. (This would be like studying *too long* for an exam and just confusing yourself.) On the other hand, going over the data only one time might lead to underfitting. This is shown experimentally in Figure 3.3. The x-axis shows the number of passes over the data and the y-axis shows the training error and the test error. As you can see, there is a "sweet spot" at which test performance begins to degrade due to overfitting.

One aspect of the perceptron algorithm that is left underspecified is line 4, which says: loop over all the training examples. The natural implementation of this would be to loop over them in a constant order. The is actually a bad idea.

? This analysis hold for the case positive examples ($y = +1$). It should also hold for negative examples. Work it out.

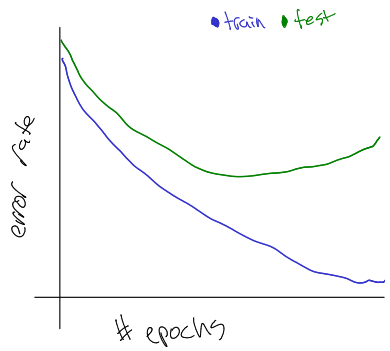



Figure 3.3: training and test error via early stopping

Consider what the perceptron algorithm would do on a data set that consisted of 500 positive examples followed by 500 negative examples. After seeing the first few positive examples (maybe five), it would likely decide that *every* example is positive, and would stop learning anything. It would do well for a while (next 495 examples), until it hit the batch of negative examples. Then it would take a while (maybe ten examples) before it would start predicting everything as negative. By the end of one pass through the data, it would really only have learned from a handful of examples (fifteen in this case).

So one thing you need to avoid is presenting the examples in some fixed order. This can easily be accomplished by permuting the order of examples once in the beginning and then cycling over the data set in the same (permuted) order each iteration. However, it turns out that you can actually do *better* if you re-permute the examples in each iteration. Figure 3.4 shows the effect of re-permuting on convergence speed. In practice, permuting each iteration tends to yield about 20% savings in number of iterations. In theory, you can actually prove that it's expected to be about twice as fast.

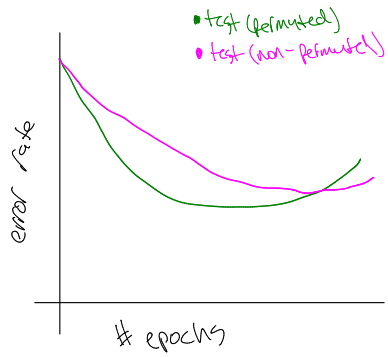



Figure 3.4: training and test error for permuting versus not-permuting

? If permuting the data each iteration saves somewhere between 20% and 50% of your time, are there any cases in which you might *not* want to permute the data every iteration?

## 3.3 Geometric Intrepretation

A question you should be asking yourself by now is: what does the decision boundary of a perceptron look like? You can actually answer that question mathematically. For a perceptron, the decision boundary is precisely where the sign of the activation, $a$, changes from $-1$ to $+1$. In other words, it is the set of points $\boldsymbol{x}$ that achieve *zero* activation. The points that are not clearly positive nor negative. For simplicity, we'll first consider the case where there is no "bias" term (or, equivalently, the bias is zero). Formally, the decision boundary $\mathcal{B}$ is:

$$\mathcal{B} = \left\{ \boldsymbol{x} \;:\; \sum_d w_d x_d = 0 \right\} \tag{3.7}$$

We can now apply some linear algebra. Recall that $\sum_d w_d x_d$ is just the **dot product** between the vector $\boldsymbol{w} = \langle w_1, w_2, \ldots, w_D \rangle$ and the vector $\boldsymbol{x}$. We will write this as $\boldsymbol{w} \cdot \boldsymbol{x}$. Two vectors have a zero dot product if and only if they are **perpendicular**. Thus, if we think of the weights as a vector $\boldsymbol{w}$, then the decision boundary is simply the plane perpendicular to $\boldsymbol{w}$.

**MATH REVIEW | DOT PRODUCTS**

dot products, definition, perpendicular, normalization and projections... think about basis vectors for projections. quadratic rule on vectors. also that dot products onto unit vectors are maximized when they point in the same direction so a*a >= a*b blah blah blah.

Figure 3.5:

This is shown pictorially in Figure 3.6. Here, the weight vector is shown, together with it's perpendicular plane. This plane forms the decision boundary between positive points and negative points. The vector points in the direction of the positive examples and away from the negative examples.

One thing to notice is that the *scale* of the weight vector is irrelevant from the perspective of classification. Suppose you take a weight vector $\boldsymbol{w}$ and replace it with $2\boldsymbol{w}$. All activations are now doubled. But their sign does not change. This makes complete sense geometrically, since all that matters is which side of the plane a test point falls on, now how far it is from that plane. For this reason, it is common to work with **normalized** weight vectors, $\boldsymbol{w}$, that have length one; i.e., $||\boldsymbol{w}|| = 1$.

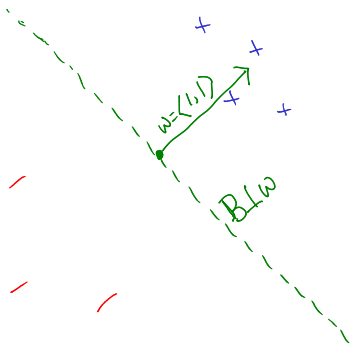



Figure 3.6: picture of data points with hy

? If I give you an arbitrary non-zero weight vector $\boldsymbol{w}$, how do I compute a weight vector $\boldsymbol{w}'$ that points in the same direction but has a norm of one?

The geometric intuition can help us even more when we realize that dot products compute projections. That is, the value $\boldsymbol{w} \cdot \boldsymbol{x}$ is just the distance of $\boldsymbol{x}$ from the origin when projected *onto* the vector $\boldsymbol{w}$. This is shown in Figure 3.7. In that figure, all the data points are projected onto $\boldsymbol{w}$. Below, we can think of this as a one-dimensional version of the data, where each data point is placed according to its projection along $\boldsymbol{w}$. This distance along $\boldsymbol{w}$ is exactly the *activiation* of that example, with no bias.

From here, you can start thinking about the role of the bias term. Previously, the threshold would be at zero. Any example with a negative projection onto $\boldsymbol{w}$ would be classified negative; any example with a positive projection, positive. The bias simply moves this threshold. Now, after the projection is computed, $b$ is added to get the overall activation. The projection *plus* $b$ is then compared against zero.

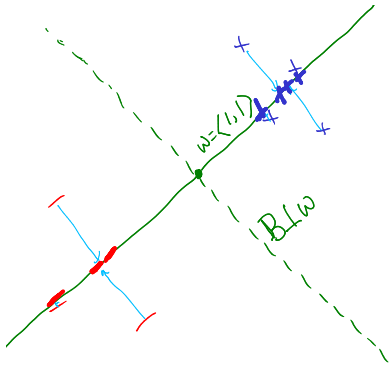



Figure 3.7: same picture as before, but with projections onto weight vector; TODO: then, below, those points along a one-dimensional axis with zero marked.

Thus, from a geometric perspective, the role of the bias is to *shift* the decision boundary away from the origin, in the direction of $\boldsymbol{w}$. It is shifted exactly $-b$ units. So if $b$ is positive, the boundary is shifted away from $\boldsymbol{w}$ and if $b$ is negative, the boundary is shifted toward $\boldsymbol{w}$. This is shown in Figure **??**. This makes intuitive sense: a positive bias means that more examples should be classified positive. By moving the decision boundary in the negative direction, more space yields a

positive classification.

The decision boundary for a perceptron is a very magical thing. In $D$ dimensional space, it is always a $D-1$-dimensional hyperplane. (In two dimensions, a 1-d hyperplane is simply a line. In three dimensions, a 2-d hyperplane is like a sheet of paper.) This hyperplane divides space in half. In the rest of this book, we'll refer to the weight vector, and to hyperplane it defines, interchangeably.

The perceptron update can also be considered geometrically. (For simplicity, we will consider the **unbiased** case.) Consider the situation in Figure **??**. Here, we have a current guess as to the hyperplane, and positive training example comes in that is currently misclassified. The weights are updated: $\boldsymbol{w} \leftarrow \boldsymbol{w} + y\boldsymbol{x}$. This yields the new weight vector, also shown in the Figure. In this case, the weight vector changed enough that this training example is now correctly classified.

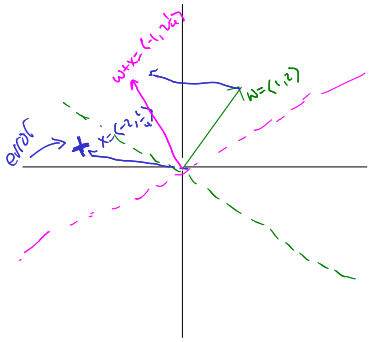



Figure 3.8: perceptron picture with update, no bias

## 3.4 *Interpreting Perceptron Weights*

TODO

## 3.5 *Perceptron Convergence and Linear Separability*

You already have an intuitive feeling for why the perceptron works: it moves the decision boundary in the direction of the training examples. A question you should be asking yourself is: does the perceptron converge? If so, what does it converge to? And how long does it take?

It is easy to construct data sets on which the perceptron algorithm will never converge. In fact, consider the (very uninteresting) learning problem with *no features*. You have a data set consisting of one positive example and one negative example. Since there are no features, the only thing the perceptron algorithm will ever do is adjust the bias. Given this data, you can run the perceptron for a bajillion iterations and it will never settle down. As long as the bias is non-negative, the negative example will cause it to decrease. As long as it is non-positive, the positive example will cause it to increase. Ad infinitum. (Yes, this is a very contrived example.)

What does it mean for the perceptron to converge? It means that it can make an entire pass through the training data without making *any* more updates. In other words, it has correctly classified *every* training example. Geometrically, this means that it was found some hyperplane that correctly segregates the data into positive and negative examples, like that shown in Figure 3.9.

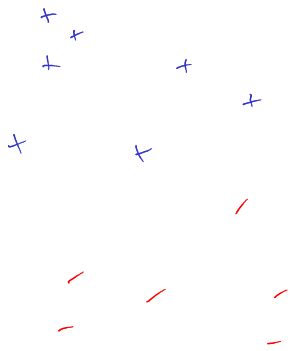


Figure 3.9: separable data

In this case, this data is **linearly separable**. This means that there

exists *some* hyperplane that puts all the positive examples on one side and all the negative examples on the other side. If the training is *not* linearly separable, like that shown in Figure 3.10, then the perceptron has no hope of converging. It could never possibly classify each point correctly.

The somewhat surprising thing about the perceptron algorithm is that *if* the data is linearly separable, *then* it will converge to a weight vector that separates the data. (And if the data is inseparable, then it will never converge.) This is great news. It means that the perceptron converges whenever it is even remotely possible to converge.

The second question is: how long does it take to converge? By "how long," what we really mean is "how many updates?" As is the case for much learning theory, you will not be able to get an answer of the form "it will converge after 5293 updates." This is asking too much. The sort of answer we can hope to get is of the form "it will converge after *at most* 5293 updates."

What you might expect to see is that the perceptron will converge more quickly for easy learning problems than for hard learning problems. This certainly fits intuition. The question is how to *define* "easy" and "hard" in a meaningful way. One way to make this definition is through the notion of **margin**. If I give you a data set and hyperplane that separates it (like that shown in Figure **??**) then the *margin* is the distance between the hyperplane and the nearest point. Intuitively, problems with large margins should be easy (there's lots of "wiggle room" to find a separating hyperplane); and problems with small margins should be hard (you really have to get a very specific well tuned weight vector).

Formally, given a data set $\mathbf{D}$, a weight vector $\boldsymbol{w}$ and bias $b$, the margin of $\boldsymbol{w}, b$ on $\mathbf{D}$ is defined as:

$$\mathit{margin}(\mathbf{D}, \boldsymbol{w}, b) = \begin{cases} \min_{(\boldsymbol{x},y)\in\mathbf{D}} y(\boldsymbol{w}\cdot\boldsymbol{x}+b) & \text{if } \boldsymbol{w} \text{ separates } \mathbf{D} \\ -\infty & \text{otherwise} \end{cases} \tag{3.8}$$

In words, the margin is only defined if $\boldsymbol{w}, b$ actually separate the data (otherwise it is just $-\infty$). In the case that it separates the data, we find the point with the minimum activation, after the activation is multiplied by the label.

? So long as the margin is not $-\infty$, it is always positive. Geometrically this makes sense, but what does Eq (3.8) yeild this?

For some historical reason (that is unknown to the author), margins are always denoted by the Greek letter $\gamma$ (gamma). One often talks about the **margin of a data set**. The margin of a data set is the largest attainable margin on this data. Formally:

$$\mathit{margin}(\mathbf{D}) = \sup_{\boldsymbol{w},b} \mathit{margin}(\mathbf{D}, \boldsymbol{w}, b) \tag{3.9}$$

In words, to compute the margin of a data set, you "try" every possible $\boldsymbol{w}, b$ pair. For each pair, you compute its margin. We then take the

largest of these as the overall margin of the data.[1] If the data is not linearly separable, then the value of the sup, and therefore the value of the margin, is $-\infty$.

There is a famous theorem due to Rosenblatt[2] that shows that the number of errors that the perceptron algorithm makes is bounded by $\gamma^{-2}$. More formally:

**Theorem 1** (Perceptron Convergence Theorem)**.** *Suppose the perceptron algorithm is run on a linearly separable data set* $\mathbf{D}$ *with margin* $\gamma > 0$*. Assume that* $||\boldsymbol{x}|| \leq 1$ *for all* $\boldsymbol{x} \in \mathbf{D}$*. Then the algorithm will converge after at most* $\frac{1}{\gamma^2}$ *updates.*

todo: comment on norm of w and norm of x also some picture about maximum margins.

The proof of this theorem is elementary, in the sense that it does not use any fancy tricks: it's all just algebra. The *idea* behind the proof is as follows. If the data is linearly separable with margin $\gamma$, then there exists some weight vector $\boldsymbol{w}^*$ that achieves this margin. Obviously we don't know what $\boldsymbol{w}^*$ is, but we know it exists. The perceptron algorithm is trying to find a weight vector $\boldsymbol{w}$ that points roughly in the same direction as $\boldsymbol{w}^*$. (For large $\gamma$, "roughly" can be very rough. For small $\gamma$, "roughly" is quite precise.) Every time the perceptron makes an update, the angle between $\boldsymbol{w}$ and $\boldsymbol{w}^*$ changes. What we prove is that the angle actually *decreases*. We show this in two steps. First, the dot product $\boldsymbol{w} \cdot \boldsymbol{w}^*$ increases a lot. Second, the norm $||\boldsymbol{w}||$ does not increase very much. Since the dot product is increasing, but $\boldsymbol{w}$ isn't getting too long, the angle between them has to be shrinking. The rest is algebra.

*Proof of Theorem 1.* The margin $\gamma > 0$ must be realized by some set of parameters, say $\boldsymbol{x}^*$. Suppose we train a perceptron on this data. Denote by $\boldsymbol{w}^{(0)}$ the initial weight vector, $\boldsymbol{w}^{(1)}$ the weight vector after the *first update*, and $\boldsymbol{w}^{(k)}$ the weight vector after the $k$th update. (We are essentially ignoring data points on which the perceptron doesn't update itself.) First, we will show that $\boldsymbol{w}^* \cdot \boldsymbol{w}^{(k)}$ grows quicky as a function of $k$. Second, we will show that $\left|\left|\boldsymbol{w}^{(k)}\right|\right|$ does not grow quickly.

First, suppose that the $k$th update happens on example $(\boldsymbol{x}, y)$. We are trying to show that $\boldsymbol{w}^{(k)}$ is becoming aligned with $\boldsymbol{w}^*$. Because we updated, know that this example was misclassified: $y\boldsymbol{w}^{(k-1)} \cdot \boldsymbol{x} < 0$. After the update, we get $\boldsymbol{w}^{(k)} = \boldsymbol{w}^{(k-1)} + y\boldsymbol{x}$. We do a little computation:

$$\begin{aligned}
\boldsymbol{w}^* \cdot \boldsymbol{w}^{(k)} &= \boldsymbol{w}^* \cdot \boldsymbol{w}^{(k-1)} + y\boldsymbol{x} && \text{definition of } \boldsymbol{w}^{(k)} && (3.10)\\
&= \boldsymbol{w}^* \cdot \boldsymbol{w}^{(k-1)} + y\boldsymbol{w}^* \cdot \boldsymbol{x} && \text{vector algebra} && (3.11)\\
&\geq \boldsymbol{w}^* \cdot \boldsymbol{w}^{(k-1)} + \gamma && \boldsymbol{w}^* \text{ has margin } \gamma && (3.12)
\end{aligned}$$

[1] You can read "sup" as "max" if you like: the only difference is a technical difference in how the $-\infty$ case is handled.

[2] Rosenblatt 1958

Thus, every time $\boldsymbol{w}^{(k)}$ is updated, its projection onto $\boldsymbol{w}^*$ incrases by at least $\gamma$. Therefore: $\boldsymbol{w}^* \cdot \boldsymbol{w}^{(k)} \geq k\gamma$.

Next, we need to show that the increase of $\gamma$ along $\boldsymbol{w}^*$ occurs because $\boldsymbol{w}^{(k)}$ is getting closer to $\boldsymbol{w}^*$, not just because it's getting exceptionally long. To do this, we compute the norm of $\boldsymbol{w}^{(k)}$:

$$\left|\left|\boldsymbol{w}^{(k)}\right|\right|^2 = \left|\left|\boldsymbol{w}^{(k-1)} + y\boldsymbol{x}\right|\right|^2 \qquad \text{definition of } \boldsymbol{w}^{(k)} \tag{3.13}$$

$$= \left|\left|\boldsymbol{w}^{(k-1)}\right|\right|^2 + y^2\,||\boldsymbol{x}||^2 + 2y\boldsymbol{w}^{(k-1)} \cdot \boldsymbol{x} \qquad \text{quadratic rule on vectors} \tag{3.14}$$

$$\leq \left|\left|\boldsymbol{w}^{(k-1)}\right|\right|^2 + 1 + 0 \qquad \text{assumption on } ||\boldsymbol{x}|| \text{ and } a < 0 \tag{3.15}$$

Thus, the squared norm of $\boldsymbol{w}^{(k)}$ increases by at most one every update. Therefore: $\left|\left|\boldsymbol{w}^{(k)}\right|\right|^2 \leq k$.

Now we put together the two things we have learned before. By our first conclusion, we know $\boldsymbol{w}^* \cdot \boldsymbol{w}^{(k)} \geq k\gamma$. But our second conclusion, $\sqrt{k} \geq \left|\left|\boldsymbol{w}^{(k)}\right|\right|^2$. Finally, because $\boldsymbol{w}^*$ is a unit vector, we know that $\left|\left|\boldsymbol{w}^{(k)}\right|\right| \geq \boldsymbol{w}^* \cdot \boldsymbol{w}^{(k)}$. Putting this together, we have:

$$\sqrt{k} \quad \geq \quad \left|\left|\boldsymbol{w}^{(k)}\right|\right| \quad \geq \quad \boldsymbol{w}^* \cdot \boldsymbol{w}^{(k)} \quad \geq \quad k\gamma \tag{3.16}$$

Taking the left-most and right-most terms, we get that $\sqrt{k} \geq k\gamma$. Dividing both sides by $k$, we get $\frac{1}{\sqrt{k}} \geq \gamma$ and therefore $k \leq \frac{1}{\sqrt{\gamma}}$. This means that once we've made $\frac{1}{\gamma^2}$ updates, we cannot make any more! □

It is important to keep in mind what this proof shows and what it does not show. It shows that if I give the perceptron data that is linearly separable with margin $\gamma > 0$, then the perceptron will converge to a solution that separates the data. And it will converge quickly when $\gamma$ is large. It does not say anything about the solution, *other than* the fact that it separates the data. In particular, the proof makes use of the maximum margin separator. But the perceptron is not guaranteed to *find* this maximum margin separator. The data may be separable with margin 0.9 and the perceptron might still find a separating hyperplane with a margin of only 0.000001. Later (in Chapter **??**), we will see algorithms that explicitly try to find the maximum margin solution.

? Perhaps we don't want to assume that all $\boldsymbol{x}$ have norm at most 1. If they have all have norm at most $R$, you can achieve a very similar bound. Modify the perceptron convergence proof to handle this case.

? Why does the perceptron convergence bound not contradict the earlier claim that poorly ordered data points (e.g., all positives followed by all negatives) will cause the perceptron to take an astronomically long time to learn?

## 3.6 *Improved Generalization: Voting and Averaging*

In the beginning of this chapter, there was a comment that the perceptron works amazingly well. This was a half-truth. The "vanilla"

perceptron algorithm does well, but not *amazingly* well. In order to make it more competitive with other learning algorithms, you need to modify it a bit to get better generalization. The key issue with the vanilla perceptron is that *it counts later points more than it counts earlier points*.

To see why, consider a data set with 10,000 examples. Suppose that after the first 100 examples, the perceptron has learned a really good classifier. It's so good that it goes over the next 9899 examples without making *any* updates. It reaches the 10,000th example and makes an error. It updates. For all we know, the update on this 10,000th example *completely ruines* the weight vector that has done so well on 99.99% of the data!

What we would like is for weight vectors that "survive" a long time to get more say than weight vectors that are overthrown quickly. One way to achieve this is by **voting**. As the perceptron learns, it remembers how long each hyperplane survives. At test time, each hyperplane encountered during training "votes" on the class of a test example. If a particular hyperplane survived for 20 examples, then it gets a vote of 20. If it only survived for one example, it only gets a vote of 1. In particular, let $(\boldsymbol{w}, b)^{(1)}, \ldots, (\boldsymbol{w}, b)^{(K)}$ be the $K+1$ weight vectors encountered during training, and $c^{(1)}, \ldots, c^{(K)}$ be the survival times for each of these weight vectors. (A weight vector that gets immediately updated gets $c = 1$; one that survives another round gets $c = 2$ and so on.) Then the prediction on a test point is:

$$\hat{y} = \text{sign}\left(\sum_{k=1}^{K} c^{(k)} \text{sign}\left(\boldsymbol{w}^{(k)} \cdot \hat{\boldsymbol{x}} + b^{(k)}\right)\right) \tag{3.17}$$

This algorithm, known as the **voted perceptron** works quite well in practice, and there is some nice theory showing that it is guaranteed to generalize better than the vanilla perceptron. Unfortunately, it is also completely impractical. If there are 1000 updates made during perceptron learning, the voted perceptron requires that you store 1000 weight vectors, together with their counts. This requires an absurd amount of storage, and makes prediction 1000 times slower than the vanilla perceptron.

A much more practical alternative is the **averaged perceptron**. The idea is similar: you maintain a collection of weight vectors and survival times. However, at test time, you predict according to the *average* weight vector, rather than the voting. In particular, the prediction is:

$$\hat{y} = \text{sign}\left(\sum_{k=1}^{K} c^{(k)} \left(\boldsymbol{w}^{(k)} \cdot \hat{\boldsymbol{x}} + b^{(k)}\right)\right) \tag{3.18}$$

The only difference between the voted prediction, Eq (**??**), and the

? The *training algorithm* for the voted perceptron is the same as the vanilla perceptron. In particular, in line 5 of Algorithm 3.2, the activation on a training example is computed based on the *current weight vector*, not based on the voted prediction. Why?

**Algorithm 7** AVERAGEDPERCEPTRONTRAIN(**D**, *MaxIter*)

```
w ← ⟨0, 0, ... 0⟩   ,   b ← 0                    // initialize weights and bias
u ← ⟨0, 0, ... 0⟩   ,   β ← 0                    // initialize cached weights and bias
c ← 1                                            // initialize example counter to one
for iter = 1 ... MaxIter do
    for all (x,y) ∈ D do
        if y(w · x + b) ≤ 0 then
            w ← w + y x                          // update weights
            b ← b + y                            // update bias
            u ← u + y c x                        // update cached weights
            β ← β + y c                          // update cached bias
        end if
        c ← c + 1                                // increment counter regardless of update
    end for
end for
return  w - (1/c) u, b - (1/c) β                 // return averaged weights and bias
```

averaged prediction, Eq (3.18), is the presense of the interior sign operator. With a little bit of algebra, we can rewrite the test-time prediction as:

$$\hat{y} = \text{sign}\left(\left(\sum_{k=1}^{K} c^{(k)} \boldsymbol{w}^{(k)}\right) \cdot \hat{\boldsymbol{x}} + \sum_{k=1}^{K} c^{(k)} b^{(k)}\right) \tag{3.19}$$

The advantage of the averaged perceptron is that we can simply maintain a *running sum* of the averaged weight vector (the blue term) and averaged bias (the red term). Test-time prediction is then just as efficient as it is with the vanilla perceptron.

The full training algorithm for the averaged perceptron is shown in Algorithm 3.6. Some of the notation is changed from the original perceptron: namely, vector operations are written as vector operations, and the activation computation is folded into the error checking.

It is probably not immediately apparent from Algorithm 3.6 that the computation unfolding is precisely the calculation of the averaged weights and bias. The most *natural* implementation would be to keep track of an averaged weight vector $\boldsymbol{u}$. At the end of every example, you would increase $\boldsymbol{u} \leftarrow \boldsymbol{u} + \boldsymbol{w}$ (and similarly for the bias). However, such an implementation would require that you updated the averaged vector on *every* example, rather than just on the examples that were incorrectly classified! Since we hope that eventually the perceptron learns to do a good job, we would hope that it will not make updates on every example. So, ideally, you would like to only update the averaged weight vector when the actual weight vector changes. The slightly clever computation in Algorithm 3.6 achieves this.

The averaged perceptron is almost always better than the per-

? By writing out the computation of the averaged weights from Eq (??) as a telescoping sum, derive the computation from Algorithm 3.6.

ceptron, in the sense that it generalizes better to test data. However, that does not free you from having to do **early stopping**. It will, eventually, overfit. Figure 3.11 shows the performance of the vanilla perceptron and the averaged perceptron on the same data set, with both training and test performance. As you can see, the averaged perceptron *does* generalize better. But it also does begin to overfit eventually.

## 3.7 Limitations of the Perceptron

Although the perceptron is very useful, it is fundamentally limited in a way that neither decision trees nor KNN are. Its limitation is that its decision boundaries can *only* be linear. The classic way of showing this limitation is through the XOR problem (XOR = exclusive or). The XOR problem is shown graphically in Figure 3.12. It consists of four data points, each at a corner of the unit square. The labels for these points are the same, along the diagonals. You can try, but you will not be able to find a linear decision boundary that perfectly separates these data points.

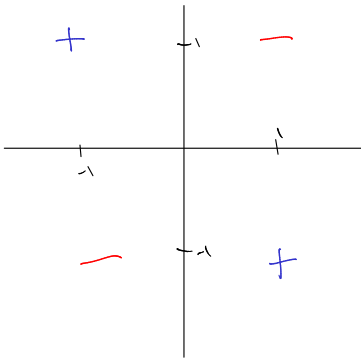


Figure 3.12: picture of xor problem

One question you might ask is: do XOR-like problems exist in the real world? Unfortunately for the perceptron, the answer is yes. Consider a sentiment classification problem that has three features that simply say whether a given word is contained in a review of a course. These features are: EXCELLENT, TERRIBLE and NOT. The EXCELLENT feature is indicative of positive reviews and the TERRIBLE feature is indicative of negative reviews. But in the presence of the NOT feature, this categorization flips.

One way to address this problem is by adding **feature combinations**. We could add two additional features: EXCELLENT-AND-NOT and TERRIBLE-AND-NOT that indicate a conjunction of these base features. By assigning weights as follows, you can achieve the desired effect:

$$w_{\text{EXECELLENT}} = +1 \qquad w_{\text{TERRIBLE}} = -1 \qquad w_{\text{NOT}} = 0$$
$$w_{\text{EXECLLENT-AND-NOT}} = -2 \qquad w_{\text{TERRIBLE-AND-NOT}} = +2$$

In this particular case, we have addressed the problem. However, if we start with $D$-many features, if we want to add all pairs, we'll blow up to $\binom{D}{2} = \mathcal{O}(D^2)$ features through this **feature mapping**. And there's no guarantee that pairs of features is enough. We might need triples of features, and now we're up to $\binom{D}{3} = \mathcal{O}(D^2)$ features. These additional features will drastically increase computation and will often result in a stronger propensity to overfitting.

In fact, the "XOR problem" is so significant that it basically killed research in classifiers with linear decision boundaries for a decade

? Suppose that you took the XOR problem and added one new feature: $x_3 = x_1 \wedge x_2$ (the logical and of the two existing features). Write out feature weights and a bias that would achieve perfect classification on this data.

or two. Later in this book, we will see two alternative approaches to taking key ideas from the perceptron and generating classifiers with non-linear decision boundaries. One approach is to combine multiple perceptrons in a single framework: this is the **neural networks** approach (see Chapter 8). The second approach is to find computationally efficient ways of doing feature mapping in a computationally and statistically efficient way: this is the **kernels** approach (see Chapter 9).

## 3.8 Exercises

**Exercise 3.1. TODO...**

# 4 | Practical Issues

In theory, there is no difference between theory and practice.
But, in practice, there is. – Jan L.A. van de Snepscheut

**Learning Objectives:**

- Translate between a problem description and a concrete learning problem.
- Perform basic feature engineering on image and text data.
- Explain how to use cross-validation to tune hyperparameters and estimate future performance.
- Compare and contrast the differences between several evaluation metrics.
- Explain why feature combinations are important for learning with some models but not others.
- Explain the relationship between the three learning techniques you have seen so far.
- Apply several debugging techniques to learning algorithms.

Dependencies: Chapter **??**,Chapter **??**,Chapter **??**

TODO: one two two examples per feature

At this point, you have seen three qualitatively different models for learning: decision trees, nearest neighbors, and perceptrons. You have also learned about clustering with the $K$-means algorithm. You will shortly learn about more complex models, most of which are variants on things you already know. However, before attempting to understand more complex models of learning, it is important to have a firm grasp on how to use machine learning in practice. This chapter is all about how to go from an abstract learning problem to a concrete implementation. You will see some examples of "best practices" along with justifications of these practices.

In many ways, going from an abstract problem to a concrete learning task is more of an art than a science. However, this art can have a huge impact on the practical performance of learning systems. In many cases, moving to a more complicated learning algorithm will gain you a few percent improvement. Going to a better representation will gain you an order of magnitude improvement. To this end, we will discuss several high level ideas to help you develop a better artistic sensibility.

## 4.1 *The Importance of Good Features*

Machine learning is magical. You give it data and it manages to classify that data. For many, it can actually outperform a human! But, like so many problems in the world, there is a significant "garbage in, garbage out" aspect to machine learning. If the data you give it is trash, the learning algorithm is unlikely to be able to overcome it.

Consider a problem of object recognition from images. If you start with a $100 \times 100$ pixel image, a very easy feature representation of this image is as a $30,000$ dimensional vector, where each dimension corresponds to the red, green or blue component of some pixel in the image. So perhaps feature 1 is the amount of red in pixel $(1,1)$; feature 2 is the amount of green in that pixel; and so on. This is the

**pixel representation** of images.

One thing to keep in mind is that the pixel representation *throws away* all locality information in the image. Learning algorithms don't care about features: they only care about feature values. So I can *permute* all of the features, with no effect on the learning algorithm (so long as I apply the same permutation to all training and test examples). Figure 4.1 shows some images whos pixels have been randomly permuted (in this case only the pixels are permuted, not the colors). All of these objects are things that you've seen plenty of examples of; can you identify them? Should you expect a machine to be able to?

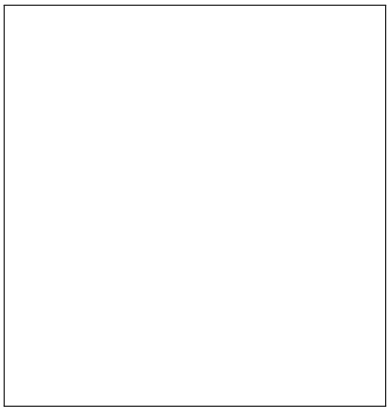


Figure 4.1: `prac:imagepix`: object recognition in pixels

An alternative representation of images is the **patch representation**, where the unit of interest is a small rectangular block of an image, rather than a single pixel. Again, permuting the patches has no effect on the classifier. Figure 4.2 shows the same images in patch representation. Can you identify them? A final representation is a **shape representation**. Here, we throw out all color and pixel information and simply provide a bounding polygon. Figure 4.3 shows the same images in this representation. Is this now enough to identify them? (If not, you can find the answers at the end of this chapter.)

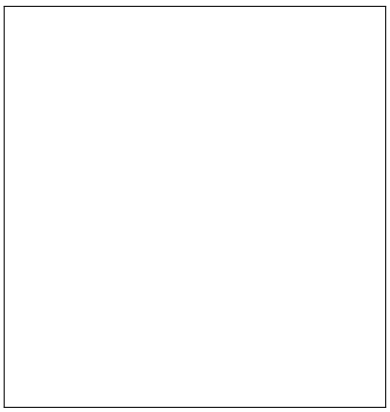


Figure 4.2: `prac:imagepatch`: object recognition in patches

In the context of **text categorization** (for instance, the sentiment recognition task), one standard representation is the **bag of words** representation. Here, we have one feature for each unique word that appears in a document. For the feature HAPPY, the feature value is the number of times that the word "happy" appears in the document. The bag of words (BOW) representation throws away all position information. Figure 4.4 shows a BOW representation for two documents: one positive and one negative. Can you tell which is which?

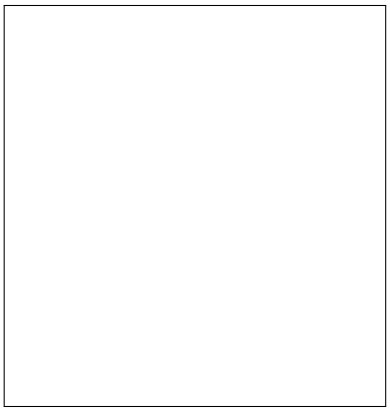


Figure 4.3: `prac:imageshape`: object recognition in shapes

## 4.2 *Irrelevant and Redundant Features*

One big difference between learning models is how robust they are to the addition of noisy or irrelevant features. Intuitively, an irrelevant feature is one that is completely uncorrelated with the prediction task. A feature $f$ whose expectation does not depend on the label $\mathbb{E}[f \mid Y] = \mathbb{E}[f]$ might be irrelevant. For instance, the presence of the word "the" might be largely irrelevant for predicting whether a course review is positive or negative.

A secondary issue is how well these algorithms deal with **redundant features**. Two features are redundant if they are highly correlated, regardless of whether they are correlated with the task or not. For example, having a bright red pixel in an image at position $(20, 93)$ is probably highly redundant with having a bright red pixel

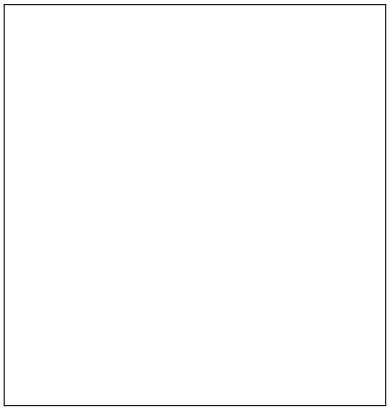


Figure 4.4: `prac:bow`: BOW repr of one positive and one negative review

at position $(21, 93)$. Both might be useful (eg., for identifying fire hydrants), but because of how images are structured, these two features are likely to co-occur frequently.

When thinking about robustness to irrelevant or redundant features, it is usually not worthwhile thinking of the case where one has 999 great features and 1 bad feature. The interesting case is when the bad features outnumber the good features, and often outnumber by a large degree. For instance, perhaps the number of good features is something like $\log D$ out of a set of $D$ total features. The question is how robust are algorithms in this case.[1]

[1] You might think it's crazy to have so many irrelevant features, but the cases you've seen so far (bag of words, bag of pixels) are both reasonable examples of this! How many words, out of the entire English vocabulary (roughly $10,000 - 100,000$ words), are actually useful for predicting positive and negative course reviews?

For shallow **decision trees**, the model explicitly selects features that are highly correlated with the label. In particular, by limiting the depth of the decision tree, one can at least *hope* that the model will be able to throw away irrelevant features. Redundant features are almost certainly thrown out: once you select one feature, the second feature now looks mostly useless. The only possible issue with irrelevant features is that even though they're irrelevant, they *happen to* correlate with the class label on the training data, but chance.

As a thought experiment, suppose that we have $N$ training examples, and exactly half are positive examples and half are negative examples. Suppose there's some binary feature, $f$, that is completely uncorrelated with the label. This feature has a 50/50 chance of appearing in any example, regardless of the label. In principle, the decision tree should *not* select this feature. But, by chance, especially if $N$ is small, the feature might *look* correlated with the label. This is analogous to flipping two coins simultaneously $N$ times. Even though the coins are independent, it's entirely possible that you will observe a sequence like $(H, H), (T, T), (H, H), (H, H)$, which makes them look entirely correlated! The hope is that as $N$ grows, this becomes less and less likely. In fact, we can explicitly compute how likely this is to happen.

To do this, let's fix the sequence of $N$ labels. We now flip a coin $N$ times and consider how likely it is that it exactly matches the label. This is easy: the probability is $0.5^N$. Now, we would also be confused if it exactly matched *not* the label, which has the same probability. So the chance that it looks perfectly correlated is $0.5^N + 0.5^N = 0.5^{N-1}$. Thankfully, this shrinks down very small (eg., $10^{-6}$) after only 21 data points.

This makes us happy. The problem is that we don't have one irrelevant feature: we have $D - \log D$ irrelevant features! If we randomly pick two irrelevant feature values, each has the same probability of perfectly correlating: $0.5^{N-1}$. But since there are two and they're independent coins, the chance that *either* correlates perfectly is $2 \times 0.5^{N-1} = 0.5^{N-2}$. In general, if we have $K$ irrelevant features, all

of which are random independent coins, the chance that at least one of them perfectly correlates is $0.5^{N-K}$. This suggests that if we have a sizeable number $K$ of irrelevant features, we'd better have at *least* $K + 21$ training examples.

Unfortunately, the situation is actually worse than this. In the above analysis we only considered the case of *perfect* correlation. We could also consider the case of *partial* correlation, which would yield even higher probabilities. (This is left as Exercise **??** for those who want some practice with probabilistic analysis.) Suffice it to say that even decision trees can become confused.

In the case of ***K*-nearest neighbors**, the situation is perhaps more dire. Since KNN weighs each feature just as much as another feature, the introduction of irrelevant features can completely mess up KNN prediction. In fact, as you saw, in high dimensional space, randomly distributed points all look approximately the same distance apart. If we add lots and lots of randomly distributed features to a data set, then all distances still converge. This is shown experimentally in Figure **??**, where we start with the digit categorization data and continually add irrelevant, uniformly distributed features, and generate a histogram of distances. Eventually, all distances converge.

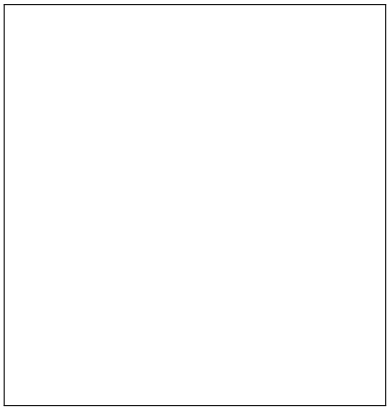


Figure 4.5: `prac:addirel`: data from high dimensional warning, interpolated

In the case of the **perceptron**, one can *hope* that it might learn to assign zero weight to irrelevant features. For instance, consider a binary feature is randomly one or zero independent of the label. If the perceptron makes just as many updates for positive examples as for negative examples, there is a reasonable chance this feature weight will be zero. At the very least, it should be small.

? What happens with the perceptron with truly redundant features (i.e., one is literally a copy of the other)?

To get a better practical sense of how sensitive these algorithms are to irrelevant features, Figure 4.6 shows the *test* performance of the three algorithms with an increasing number of compltely noisy features. In all cases, the hyperparameters were tuned on validation data. TODO...

## 4.3 *Feature Pruning and Normalization*

In text categorization problems, some words simply do not appear very often. Perhaps the word "groovy"[2] appears in exactly one training document, which is positive. Is it really worth keeping this word around as a feature? It's a dangerous endeavor because it's hard to tell with just one training example if it is really correlated with the positive class, or is it just noise. You could hope that your learning algorithm is smart enough to figure it out. Or you could just remove it. That means that (a) the learning algorithm won't have to figure it out, and (b) you've reduced the number of dimensions you have, so things should be more efficient, and less "scary."

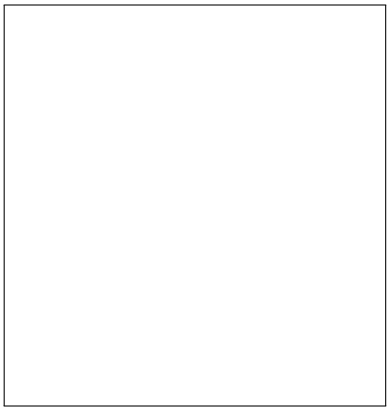


Figure 4.6: `prac:noisy`: dt,knn,perc on increasing amounts of noise

[2] This is typically positive indicator, or at least it was back in the US in the 1970s.

**MATH REVIEW | DATA STATISTICS: MEANS AND VARIANCES**

data mean, variance, moments, expectations, etc...

This idea of feature pruning is very useful and applied in many applications. It is easiest in the case of binary features. If a binary feature only appears some small number $K$ times (in the training data: no fair looking at the test data!), you simply remove it from consideration. (You might also want to remove features that appear in all-but-$K$ many documents, for instance the word "the" appears in pretty much every English document ever written.) Typical choices for $K$ are $1, 2, 5, 10, 20, 50$, mostly depending on the size of the data. On a text data set with 1000 documents, a cutoff of 5 is probably reasonable. On a text data set the size of the web, a cut of of 50 or even 100 or 200 is probably reasonable[3]. Figure 4.7 shows the effect of pruning on a sentiment analysis task. In the beginning, pruning does not hurt (and sometimes helps!) but eventually we prune away all the interesting words and performance suffers.

In the case of real-valued features, the question is how to extend the idea of "does not occur much" to real values. A reasonable definition is to look for features with *low variance*. In fact, for binary features, ones that almost never appear or almost always appear will also have low variance. Figure 4.9 shows the result of pruning low-variance features on the digit recognition task. Again, at first pruning does not hurt (and sometimes helps!) but eventually we have thrown out all the useful features.

Once you have pruned away irrelevant features, it is often useful to **normalize** the data so that it is consistent in some way. There are two basic types of normalization: **feature normalization** and **example normalization**. In feature normalization, you go through each feature and adjust it the same way across all examples. In example normalization, each example is adjusted individually.

The goal of both types of normalization is to make it *easier* for your learning algorithm to learn. In feature normalization, there are two standard things to do:

1. Centering: moving the entire data set so that it is centered around the origin.
2. Scaling: rescaling each feature so that one of the following holds:
   (a) Each feature has variance 1 across the training data.
   (b) Each feature has maximum absolute value 1 across the training data.

Figure 4.8:

[3] According to Google, the following words (among many others) appear 200 times on the web: moudlings, agaggagctg, setgravity, rogov, prosomeric, spunlaid, piyushtwok, telelesson, nesmysl, brighnasa. For comparison, the word "the" appears 19,401,194,714 (19 billion) times.

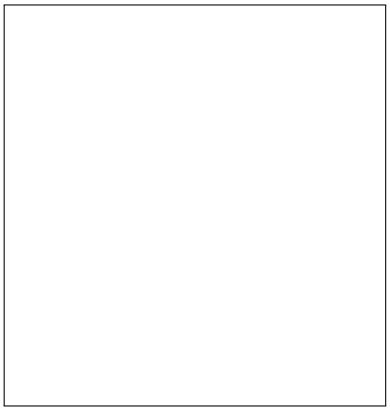


Figure 4.9: `prac:variance`: effect of pruning on vision

? Earlier we discussed the problem of *scale* of features (eg., millimeters versus centimeters). Does this have an impact on variance-based feature pruning?

These transformations are shown geometrically in Figure 4.10. The goal of centering is to make sure that no features are arbitrarily large. The goal of scaling is to make sure that all features have roughly the same scale (to avoid the issue of centimeters versus millimeters).

These computations are fairly straightforward. Here, $x_{n,d}$ refers to the $d$th feature of example $n$. Since it is very rare to apply scaling without previously applying centering, the expressions below for scaling assume that the data is already centered.

? For the three models you know about (KNN, DT, Perceptron), which are most sensitive to centering? Which are most sensitive to scaling?

$$\text{Centering:} \qquad x_{n,d} \leftarrow x_{n,d} - \mu_d \tag{4.1}$$

$$\text{Variance Scaling:} \qquad x_{n,d} \leftarrow x_{n,d} / \sigma_d \tag{4.2}$$

$$\text{Absolute Scaling:} \qquad x_{n,d} \leftarrow x_{n,d} / r_d \tag{4.3}$$

$$\textit{where:} \qquad \mu_d = \frac{1}{N} \sum_n x_{n,d} \tag{4.4}$$

$$\sigma_d = \sqrt{\frac{1}{N} \sum_n (x_{n,d} - \mu_d)^2} \tag{4.5}$$

$$r_d = \max_n \left| x_{n,d} \right| \tag{4.6}$$

In practice, if the dynamic range of your features is already some subset of $[-2, 2]$ or $[-3, 3]$, then it is probably not worth the effort of centering and scaling. (It's an effort because you have to keep around your centering and scaling calculations so that you can apply them to the test data as well!) However, if some of your features are orders of magnitude larger than others, it might be helpful. Remember that you might know best: if the difference in scale is actually significant for your problem, then rescaling might throw away useful information.

One thing to be wary of is centering binary data. In many cases, binary data is very *sparse*: for a given example, only a few of the features are "on." For instance, out of a vocabulary of 10,000 or 100,000 words, a given document probably only contains about 100. From a storage and computation perspective, this is very useful. However, after centering, the data will no longer sparse and you will pay dearly with outrageously slow implementations.

In **example normalization**, you view examples one at a time. The most standard normalization is to ensure that the length of each example vector is one: namely, each example lies somewhere on the unit hypersphere. This is a simple transformation:

$$\text{Example Normalization:} \qquad \boldsymbol{x}_n \leftarrow \boldsymbol{x}_n / \left|\left|\boldsymbol{x}_n\right|\right| \tag{4.7}$$

This transformation is depicted in Figure 4.11.

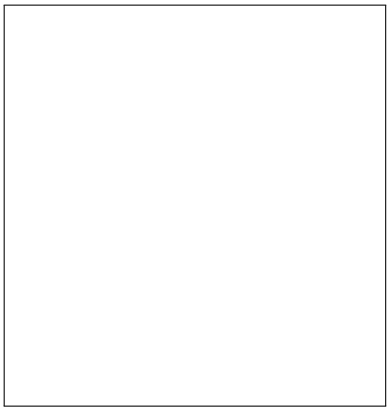


Figure 4.11: `prac:exnorm`: example of example normalization

The main advantage to example normalization is that it makes comparisons more straightforward across data sets. If I hand you

two data sets that differ only in the norm of the feature vectors (i.e., one is just a scaled version of the other), it is difficult to compare the learned models. Example normalization makes this more straightforward. Moreover, as you saw in the perceptron convergence proof, it is often just mathematically easier to assume normalized data.

## 4.4 Combinatorial Feature Explosion

You learned in Chapter 3 that linear models (like the perceptron) cannot solve the XOR problem. You also learned that by performing a combinatorial feature explosion, they could. But that came at the computational expense of gigantic feature vectors.

Of the algorithms that you've seen so far, the perceptron is the one that has the most to gain by feature combination. And the decision tree is the one that has the least to gain. In fact, the decision tree construction is essentially building meta features for you. (Or, at least, it is building meta features constructed purely through "logical ands.")

This observation leads to a heuristic for constructing meta features *for* perceptrons *from* decision trees. The idea is to train a decision tree on the training data. From that decision tree, you can extract meta features by looking at feature combinations along branches. You can then add *only* those feature combinations as meta features to the feature set for the perceptron. Figure 4.12 shows a small decision tree and a set of meta features that you might extract from it. There is a hyperparameter here of what length paths to extract from the tree: in this case, only paths of length two are extracted. For bigger trees, or if you have more data, you might benefit from longer paths.

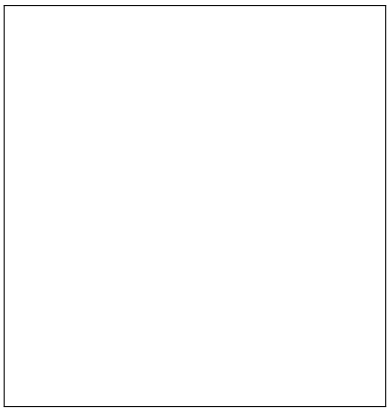


Figure 4.12: `prac:dttoperc`: turning a DT into a set of meta features

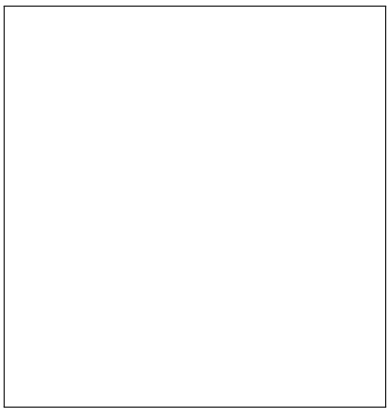


Figure 4.13: `prac:log`: performance on text categ with word counts versus log word counts

In addition to combinatorial transformations, the **logarithmic transformation** can be quite useful in practice. It seems like a strange thing to be useful, since it doesn't seem to fundamentally change the data. However, since many learning algorithms operate by linear operations on the features (both perceptron and KNN do this), the log-transform is a way to get product-like operations. The question is which of the following feels more applicable to your data: (1) every time this feature increases by one, I'm equally more likely to predict a positive label; (2) every time this feature doubles, I'm equally more like to predict a positive label. In the first case, you should stick with linear features and in the second case you should switch to a log-transform. This is an important transformation in text data, where the presence of the word "excellent" once is a good indicator of a positive review; seeing "excellent" twice is a better indicator; but the difference between seeing "excellent" 10 times and seeing it 11 times really isn't a big deal any more. A log-transform achieves

this. Experimentally, you can see the difference in test performance between word count data and log-word count data in Figure 4.13. Here, the transformation is actually $x_d \mapsto \log_2(x_d + 1)$ to ensure that zeros remain zero and sparsity is retained.

## 4.5 *Evaluating Model Performance*

So far, our focus has been on classifiers that achieve *high accuracy*. In some cases, this is not what you might want. For instance, if you are trying to predict whether a patient has cancer or not, it might be better to err on one side (saying they have cancer when they don't) than the other (because then they die). Similarly, letting a little spam slip through might be better than accidentally blocking one email from your boss.

There are two major types of binary classification problems. One is "X versus Y." For instance, positive versus negative sentiment. Another is "X versus not-X." For instance, spam versus non-spam. (The argument being that there are lots of types of non-spam.) Or in the context of web search, relevant document versus irrelevant document. This is a subtle and subjective decision. But "X versus not-X" problems often have more of the feel of "X spotting" rather than a true distinction between X and Y. (Can you spot the spam? can you spot the relevant documents?)

For spotting problems (X versus not-X), there are often more appropriate success metrics than accuracy. A very popular one from information retrieval is the **precision**/**recall** metric. Precision asks the question: of all the X's that you found, how many of them were actually X's? Recall asks: of all the X's that were out there, how many of them did you find?[4] Formally, precision and recall are defined as:

$$P = \frac{I}{S} \tag{4.8}$$

$$R = \frac{I}{T} \tag{4.9}$$

$$S = \text{number of Xs that your system found} \tag{4.10}$$

$$T = \text{number of Xs in the data} \tag{4.11}$$

$$I = \text{number of correct Xs that your system found} \tag{4.12}$$

Here, $S$ is mnemonic for "System," $T$ is mnemonic for "Truth" and $I$ is mnemonic for "Intersection." It is generally accepted that $0/0 = 1$ in these definitions. Thus, if you system found nothing, your precision is always perfect; and if there is nothing to find, your recall is always perfect.

Once you can compute precision and recall, you are often able to produce **precision/recall curves**. Suppose that you are attempting

[4] A colleague make the analogy to the US court system's saying "Do you promise to tell the whole truth and nothing but the truth?" In this case, the "whole truth" means high recall and "nothing but the truth" means high precision."

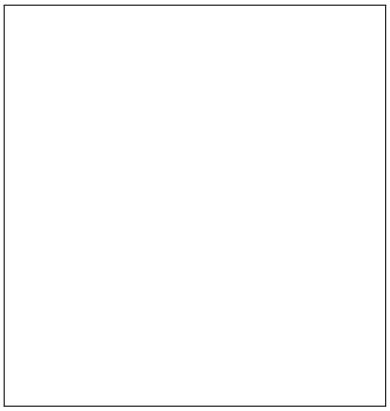


Figure 4.14: `prac:spam`: show a bunch of emails spam/nospam sorted by model predicion, not perfect

to identify spam. You run a learning algorithm to make predictions on a test set. But instead of just taking a "yes/no" answer, you allow your algorithm to produce its confidence. For instance, in perceptron, you might use the distance from the hyperplane as a confidence measure. You can then sort all of your test emails according to this ranking. You may put the most spam-like emails at the top and the least spam-like emails at the bottom, like in Figure 4.14.

? How would you get a confidence out of a decision tree or KNN?

Once you have this sorted list, you can choose how aggressively you want your spam filter to be by setting a threshold *anywhere* on this list. One would hope that if you set the threshold very high, you are likely to have high precision (but low recall). If you set the threshold very low, you'll have high recall (but low precision). By considering *every possible* place you could put this threshold, you can trace out a curve of precision/recall values, like the one in Figure 4.15. This allows us to ask the question: for some fixed precision, what sort of recall can I get. Obviously, the closer your curve is to the upper-right corner, the better. And when comparing learning algorithms A and B you can say that A **dominates** B if A's precision/recall curve is always higher than B's.

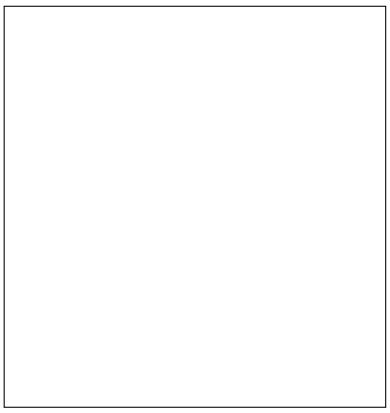


Figure 4.15: `prac:prcurve`: precision recall curve

Precision/recall curves are nice because they allow us to visualize many ways in which we could use the system. However, sometimes we like to have a *single number* that informs us of the quality of the solution. A popular way of combining precision and recall into a single number is by taking their harmonic mean. This is known as the balanced f-measure (or f-score):

$$F = \frac{2 \times P \times R}{P + R} \tag{4.13}$$

The reason that you want to use a harmonic mean rather than an arithmetic mean (the one you're more used to) is that it favors systems that achieve roughly equal precision and recall. In the extreme case where $P = R$, then $F = P = R$. But in the imbalanced case, for instance $P = 0.1$ and $R = 0.9$, the overall f-measure is a modest 0.18. Table 4.1 shows f-measures as a function of precision and recall, so that you can see how important it is to get balanced values.

| | 0.0 | 0.2 | 0.4 | 0.6 | 0.8 | 1.0 |
|---|---|---|---|---|---|---|
| 0.0 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 0.2 | 0.00 | 0.20 | 0.26 | 0.30 | 0.32 | 0.33 |
| 0.4 | 0.00 | 0.26 | 0.40 | 0.48 | 0.53 | 0.57 |
| 0.6 | 0.00 | 0.30 | 0.48 | 0.60 | 0.68 | 0.74 |
| 0.8 | 0.00 | 0.32 | 0.53 | 0.68 | 0.80 | 0.88 |
| 1.0 | 0.00 | 0.33 | 0.57 | 0.74 | 0.88 | 1.00 |

Table 4.1: Table of f-measures when varying precision and recall values.

In some cases, you might believe that precision is more important than recall. This idea leads to the *weighted* f-measure, which is parameterized by a weight $\beta \in [0, \infty)$ (beta):

$$F_\beta = \frac{(1 + \beta^2) \times P \times R}{\beta^2 \times P + R} \tag{4.14}$$

For $\beta = 1$, this reduces to the standard f-measure. For $\beta = 0$, it focuses entirely on recall and for $\beta \to \infty$ it focuses entirely on precision. The interpretation of the weight is that $F_\beta$ measures the perfor-

mance for a user who cares $\beta$ times as much about precision as about recall.

One thing to keep in mind is that precision and recall (and hence f-measure) depend crucially on which class is considered the thing you wish to find. In particular, if you take a binary data set if flip what it means to be a positive or negative example, you will end up with completely difference precision and recall values. It is *not* the case that precision on the flipped task is equal to recall on the original task (nor vice versa). Consequently, f-measure is also not the same. For some tasks where people are less sure about what they want, they will occasionally report two sets of precision/recall/f-measure numbers, which vary based on which class is considered the thing to spot.

There are other standard metrics that are used in different communities. For instance, the medical community is fond of the **sensitivity**/**specificity** metric. A sensitive classifier is one which almost always finds everything it is looking for: it has high recall. In fact, sensitivity is exactly the same as recall. A specific classifier is one which does a good job *not* finding the things that it doesn't want to find. Specificity is precision on the negation of the task at hand.

You can compute curves for sensitivity and specificity much like those for precision and recall. The typical plot, referred to as the **receiver operating characteristic** (or **ROC curve**) plots the sensitivity against $1 -$ specificity. Given an ROC curve, you can compute the **area under the curve** (or **AUC**) metric, which also provides a meaningful single number for a system's performance. Unlike f-measures, which tend to be low because the require agreement, AUC scores tend to be very high, even for not great systems. This is because random chance will give you an AUC of 0.5 and the best possible AUC is 1.0.

The main message for evaluation metrics is that you should choose whichever one makes the most sense. In many cases, several might make sense. In that case, you should do whatever is more commonly done in your field. There is no reason to be an outlier without cause.

## 4.6 Cross Validation

In Chapter 1, you learned about using development data (or held-out data) to set hyperparameters. The main disadvantage to the development data approach is that you throw out some of your training data, just for estimating one or two hyperparameters.

An alternative is the idea of **cross validation**. In cross validation, you break your training data up into 10 equally-sized partitions. You train a learning algorithm on 9 of them and test it on the remaining

**Algorithm 8** CROSSVALIDATE(*LearningAlgorithm*, *Data*, *K*)

```
1:  ê ← ∞                                          // store lowest error encountered so far
2:  α̂ ← unknown                                    // store the hyperparameter setting that yielded it
3:  for all hyperparameter settings α do
4:      err ← [ ]                                  // keep track of the K-many error estimates
5:      for k = 1 to K do
6:          train ← {(x_n, y_n) ∈ Data : n mod K ≠ k − 1}
7:          test ← {(x_n, y_n) ∈ Data : n mod K = k − 1}   // test every Kth example
8:          model ← Run LearningAlgorithm on train
9:          err ← err ⊕ error of model on test     // add current error to list of errors
10:     end for
11:     avgErr ← mean of set err
12:     if avgErr < ê then
13:         ê ← avgErr                             // remember these settings
14:         α̂ ← α                                  // because they're the best so far
15:     end if
16: end for
```

1. You do this 10 times, each time holding out a different partition as the "development" part. You can then average your performance over all ten parts to get an estimate of how well your model will perform in the future. You can repeat this process for every possible choice of hyperparameters to get an estimate of which one performs best. The general $K$-fold cross validation technique is shown in Algorithm 4.6, where $K = 10$ in the preceeding discussion.

In fact, the development data approach can be seen as an approximation to cross validation, wherein only one of the $K$ loops (line 5 in Algorithm 4.6) is executed.

Typical choices for $K$ are 2, 5, 10 and $N - 1$. By far the most common is $K = 10$: 10-fold cross validation. Sometimes 5 is used for efficiency reasons. And sometimes 2 is used for subtle statistical reasons, but that is quite rare. In the case that $K = N - 1$, this is known as **leave-one-out cross validation** (or abbreviated as **LOO cross validation**). After running cross validation, you have two choices. You can either select one of the $K$ trained models as your final model to make predictions with, or you can train a *new* model on all of the data, using the hyperparameters selected by cross-validation. If you have the time, the latter is probably a better options.

It may seem that LOO cross validation is prohibitively expensive to run. This is true for most learning algorithms *except for* $K$-nearest neighbors. For KNN, leave-one-out is actually very natural. We loop through each training point and ask ourselves whether this example would be correctly classified for all different possible values of $K$. This requires only as much computation as computing the $K$ nearest neighbors for the highest value of $K$. This is such a popular and effective approach for KNN classification that it is spelled out in

**Algorithm 9** KNN-TRAIN-LOO(**D**)

```
err_k ← 0, ∀1 ≤ k ≤ N − 1                    // err_k stores how well you do with kNN
for n = 1 to N do
    S_m ← ⟨||x_n − x_m|| , m⟩, ∀m ≠ n          // compute distances to other points
    S ← SORT(S)                               // put lowest-distance objects first
    ŷ ← 0                                     // current label prediction
    for k = 1 to N − 1 do
        ⟨dist,m⟩ ← S_k
        ŷ ← ŷ + y_m                           // let kth closest point vote
        if ŷ ≠ y_m then
            err_k ← err_k + 1                 // one more error for kNN
        end if
    end for
end for
return argmin_k err_k                         // return the K that achieved lowest error
```

Algorithm **??**.

Overall, the main advantage to cross validation over development data is robustness. The main advantage of development data is speed.

One warning to keep in mind is that the goal of both cross validation and development data is to estimate how well you will do in the future. This is a question of statistics, and holds *only if* your test data really looks like your training data. That is, it is drawn from the same distribution. In many practical cases, this is not entirely true.

For example, in person identification, we might try to classify every pixel in an image based on whether it contains a person or not. If we have 100 training images, each with $10,000$ pixels, then we have a total of $1m$ training examples. The classification for a pixel in image 5 is highly dependent on the classification for a neighboring pixel in the same image. So if one of those pixels happens to fall in training data, and the other in development (or cross validation) data, your model will do unreasonably well. In this case, it is important that when you cross validate (or use development data), you do so over *images*, not over *pixels*. The same goes for text problems where you sometimes want to classify things at a word level, but are handed a collection of documents. The important thing to keep in mind is that it is the *images* (or documents) that are drawn independently from your data distribution and *not* the pixels (or words), which are drawn dependently.

## 4.7 *Hypothesis Testing and Statistical Significance*

**VIGNETTE: THE LADY DRINKING TEA**

story

Suppose that you've presented a machine learning solution to your boss that achieves 7% error on cross validation. Your nemesis, Gabe, gives a solution to your boss that achieves 6.9% error on cross validation. How impressed should your boss be? It depends. If this 0.1% improvement was measured over 1000 examples, perhaps not too impressed. It would mean that Gabe got exactly one more example right than you did. (In fact, he probably got 15 more right and 14 more wrong.) If this 0.1% impressed was measured over $1,000,000$ examples, perhaps this is more impressive.

This is one of the most fundamental questions in statistics. You have a scientific hypothesis of the form "Gabe's algorithm is better than mine." You wish to test whether this hypothesis is true. You are testing it against the **null hypothesis**, which is that Gabe's algorithm is no better than yours. You've collected data (either 1000 or $1m$ data points) to measure the strength of this hypothesis. You want to ensure that the difference in performance of these two algorithms is **statistically significant**: i.e., is probably not just due to random luck. (A more common question statisticians ask is whether one drug treatment is better than another, where "another" is either a placebo or the competitor's drug.)

There are about $\infty$-many ways of doing **hypothesis testing**. Like evaluation metrics and the number of folds of cross validation, this is something that is very discipline specific. Here, we will discuss two popular tests: the **paired t-test** and **bootstrapping**. These tests, and other statistical tests, have underlying assumptions (for instance, assumptions about the distribution of observations) and strengths (for instance, small or large samples). In most cases, the goal of hypothesis testing is to compute a **p-value**: namely, the probability that the observed difference in performance was by chance. The standard way of reporting results is to say something like "there is a 95% chance that this difference was not by chance." The value 95% is arbitrary, and occasionally people use weaker (90%) test or stronger (99.5%) tests.

The **t-test** is an example of a **parametric test**. It is applicable when the null hypothesis states that the difference between two responses has mean zero and unknown variance. The t-test actually assumes that data is distributed according to a Gaussian distribution, which is

probably *not* true of binary responses. Fortunately, for large samples (at least a few hundred), binary seamples are well approximated by a Gaussian distribution. So long as your sample is sufficiently large, the t-test is reasonable either for regression or classification problems.

Suppose that you evaluate two algorithm on $N$-many examples. On each example, you can compute whether the algorithm made the correct prediction. Let $a_1, \ldots, a_N$ denote the error of the first algorithm on each example. Let $b_1, \ldots, b_N$ denote the error of the second algorithm. You can compute $\mu_a$ and $\mu_b$ as the means of $\boldsymbol{a}$ and $\boldsymbol{b}$, respecitively. Finally, center the data as $\hat{a} = \boldsymbol{a} - \mu_a$ and $\hat{b} = \boldsymbol{b} - \mu_b$. The t-statistic is defined as:

$$t = (\mu_a - \mu_b) \sqrt{\frac{N(N-1)}{\sum_n (\hat{a}_n - \hat{b}_n)^2}} \tag{4.15}$$

After computing the $t$-value, you can compare it to a list of values for computing **confidence intervals**. Assuming you have a lot of data ($N$ is a few hundred or more), then you can compare your $t$-value to Table 4.2 to determine the significance level of the difference.

| $t$ | significance |
|---|---|
| $\geq 1.28$ | 90.0% |
| $\geq 1.64$ | 95.0% |
| $\geq 1.96$ | 97.5% |
| $\geq 2.58$ | 99.5% |

Table 4.2: Table of significance values for the t-test.

One disadvantage to the t-test is that it cannot easily be applied to evaluation metrics like f-score. This is because f-score is a computed over an entire test set and does not decompose into a set of individual errors. This means that the t-test cannot be applied.

? What does it mean for the means $\mu_a$ and $\mu_b$ to become further apart? How does this affect the $t$-value? What happens if the variance of $\boldsymbol{a}$ increases?

Fortunately, **cross validation** gives you a way around this problem. When you do $K$-fold cross validation, you are able to compute $K$ error metrics over the same data. For example, you might run 5-fold cross validation and compute f-score for every fold. Perhaps the f-scores are 92.4, 93.9, 96.1, 92.2 and 94.4. This gives you an average f-score of 93.8 over the 5 folds. The standard deviation of this set of f-scores is:

$$\sigma = \sqrt{\frac{1}{N-1} \sum_n (a_i - \mu)^2} \tag{4.16}$$

$$= \sqrt{\frac{1}{4}(1.96 + 0.01 + 5.29 + 2.56 + 0.36)} \tag{4.17}$$

$$= 1.595 \tag{4.18}$$

You can now assume that the distribution of scores is approximately Gaussian. If this is true, then approximately 70% of the probability mass lies in the range $[\mu - \sigma, \mu + \sigma]$; 95% lies in the range $[\mu - 2\sigma, \mu + 2\sigma]$; and 99.5% lies in the range $[\mu - 3\sigma, \mu + 3\sigma]$. So, if we were comparing our algorithm against one whose average f-score was 90.6%, we could be 95% certain that our superior performance was not due to chance.[5]

**WARNING:** A confidence of 95% does not mean "There is a 95% chance that I am better." All it means is that if I reran the same ex-

[5] Had we run 10-fold cross validation we might be been able to get tighter confidence intervals.

**Algorithm 10** BOOTSTRAPEVALUATE($\boldsymbol{y}$, $\hat{\boldsymbol{y}}$, *NumFolds*)

```
scores ← [ ]
for k = 1 to NumFolds do
    truth ← [ ]                                  // list of values we want to predict
    pred ← [ ]                                   // list of values we actually predicted
    for n = 1 to N do
        m ← uniform random value from 1 to N     // sample a test point
        truth ← truth ⊕ y_m                      // add on the truth
        pred ← pred ⊕ ŷ_m                        // add on our prediction
    end for
    scores ← scores ⊕ F-SCORE(truth, pred)       // evaluate
end for
return (MEAN(scores), STDDEV(scores))
```

periment 100 times, then in 95 of those experiments I would still win. These are *very* different statements. If you say the first one, people who know about statistics will get very mad at you!

One disadvantage to cross validation is that it is computationally expensive. More folds typically leads to better estimates, but every new fold requires training a new classifier. This can get very time consuming. The technique of **bootstrapping** (and closely related idea of **jack-knifing** can address this problem.

Suppose that you didn't want to run cross validation. All you have is a single held-out test set with 1000 data points in it. You can run your classifier and get predictions on these 1000 data points. You would like to be able to compute a metric like f-score on this test set, but also get confidence intervals. The idea behind bootstrapping is that this set of 1000 is a random draw from some distribution. We would like to get multiple random draws from this distribution on which to evaluate. We can *simulate* multiple draws by repeatedly subsampling from these 1000 examples, with replacement.

To perform a *single* bootstrap, you will sample 1000 random points from your test set of 1000 random points. This sampling must be done with replacement (so that the same example can be sampled more than once), otherwise you'll just end up with your original test set. This gives you a bootstrapped sample. On this sample, you can compute f-score (or whatever metric you want). You then do this 99 more times, to get a 100-fold bootstrap. For each bootstrapped sample, you will be a different f-score. The mean and standard deviation of this set of f-scores can be used to estimate a confidence interval for your algorithm.

The bootstrap resampling procedure is sketched in Algorithm 4.7. This takes three arguments: the true labels $\boldsymbol{y}$, the predicted labels $\hat{\boldsymbol{y}}$ and the number of folds to run. It returns the mean and standard deviation from which you can compute a confidence interval.

## 4.8 Debugging Learning Algorithms

Learning algorithms are notoriously hard to debug, as you may have already experienced if you have implemented any of the models presented so far. The main issue is that when a learning algorithm doesn't learn, it's unclear if this is because there's a bug or because the learning problem is too hard (or there's too much noise, or ...). Moreover, sometimes bugs lead to learning algorithms performing *better* than they should: these are especially hard to catch (and always a bit disappointing when you do catch them).

Obviously if you have a reference implementation, you can attempt to match its output. Otherwise, there are two things you can do to try to help debug. The first is to do everything in your power to get the learning algorithm to overfit. If it cannot *at least* overfit the training data, there's definitely something wrong. The second is to feed it some tiny hand-specified two dimensional data set on which you *know* what it should do and you can plot the output.

The easiest way to try to get a learning algorithm to overfit is to add a new feature to it. You can call this feature the CheatingIsFun feature[6]. The feature value associated with this feature is $+1$ if this is a positive example and $-1$ (or zero) if this is a negative example. In other words, this feature is a *perfect* indicator of the class of this example.

[6] Note: cheating is actually *not* fun and you shouldn't do it!

If you add the CheatingIsFun feature and your algorithm does not get near 0% training error, this could be because there are too many noisy features confusing it. You could either remove a lot of the other features, or make the feature value for CheatingIsFun either $+100$ or $-100$ so that the algorithm *really* looks at it. If you do this and your algorithm still cannot overfit then you likely have a bug. (Remember to remove the CheatingIsFun feature from your final implementation!)

A second thing to try is to hand-craft a data set on which you know your algorithm should work. This is also useful if you've managed to get your model to overfit and have simply noticed that it does not generalize. For instance, you could run KNN on the XOR data. Or you could run perceptron on some easily linearly separable data (for instance positive points along the line $x_2 = x_1 + 1$ and negative points along the line $x_2 = x_1 - 1$). Or a decision tree on nice axis-aligned data.

When debugging on hand-crafted data, remember whatever you know about the models you are considering. For instance, you know that the perceptron should converge on linearly separable data, so try it on a linearly separable data set. You know that decision trees should do well on data with only a few relevant features, so make

your label some easy combination of features, such as $y = x_1 \vee (x_2 \wedge \neg x_3)$. You know that KNN should work well on data sets where the classes are well separated, so try such data sets.

The most important thing to keep in mind is that a *lot* goes in to getting good test set performance. First, the model has to be right for the data. So crafting your own data is helpful. Second, the model has to fit the training data well, so try to get it to overfit. Third, the model has to generalize, so make sure you tune hyperparameters well.

TODO: answers to image questions

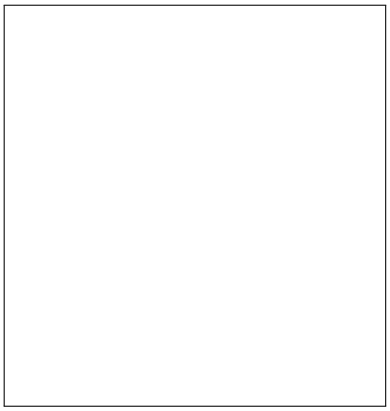


Figure 4.16: `prac:imageanswers`: object recognition answers

## 4.9 *Exercises*

**Exercise 4.1. TODO...**

# 5 | Beyond Binary Classification

**Learning Objectives:**

- Represent complex prediction problems in a formal learning setting.
- Be able to artifically "balance" imbalanced data.
- Understand the positive and negative aspects of several reductions from multiclass classification to binary classification.
- Recognize the difference between regression and ordinal regression.
- Implement stacking as a method of collective classification.

Dependencies:

In the preceeding chapters, you have learned all about a very simple form of prediction: predicting bits. In the real world, however, we often need to predict much more complex objects. You may need to categorize a document into one of several categories: sports, entertainment, news, politics, etc. You may need to rank web pages or ads based on relevance to a query. You may need to simultaneously classify a collection of objects, such as web pages, that have important information in the links between them. These problems are all commonly encountered, yet fundamentally more complex than binary classification.

In this chapter, you will learn how to use everything you already know about binary classification to solve these more complicated problems. You will see that it's relatively easy to think of a binary classifier as a black box, which you can reuse for solving these more complex problems. This is a very useful abstraction, since it allows us to reuse knowledge, rather than having to build new learning models and algorithms from scratch.

## 5.1 *Learning with Imbalanced Data*

Your boss tells you to build a classifier that can identify fraudulent transactions in credit card histories. Fortunately, most transactions are legitimate, so perhaps only 0.1% of the data is a positive instance. The **imbalanced data** problem refers to the fact that for a large number of real world problems, the number of positive examples is dwarfed by the number of negative examples (or vice versa). This is actually something of a misnomer: it is not the *data* that is imbalanced, but the *distribution* from which the data is drawn. (And since the distribution is imbalanced, so must the data be.)

Imbalanced data is a problem because machine learning algorithms are too smart for your own good. For most learning algorithms, if you give them data that is 99.9% negative and 0.1% positive, they will simply learn to always predict negative. Why? Because

they are trying to minimize error, and they can achieve 0.1% error by doing nothing! If a teacher told you to study for an exam with 1000 true/false questions and only one of them is true, it is unlikely you will study very long.

Really, the problem is not with the data, but rather with the way that you have defined the learning problem. That is to say, what you care about is *not* accuracy: you care about something else. If you want a learning algorithm to do a reasonable job, you have to tell it what you want!

Most likely, what you want is *not* to optimize accuracy, but rather to optimize some other measure, like f-score or AUC. You want your algorithm to make *some* positive predictions, and simply prefer those to be "good." We will shortly discuss two heuristics for dealing with this problem: subsampling and weighting. In subsampling, you *throw out* some of you negative examples so that you are left with a balanced data set (50% positive, 50% negative). This might scare you a bit since throwing out data seems like a bad idea, but at least it makes learning much more efficient. In weighting, instead of throwing out positive examples, we just given them lower weight. If you assign an **importance weight** of 0.00101 to each of the positive examples, then there will be as much *weight* associated with positive examples as negative examples.

Before formally defining these heuristics, we need to have a mechanism for formally defining supervised learning problems. We will proceed by example, using binary classification as the canonical learning problem.

**TASK: BINARY CLASSIFICATION**

*Given:*

1. An input space $\mathcal{X}$
2. An unknown distribution $\mathcal{D}$ over $\mathcal{X} \times \{-1, +1\}$

*Compute:* A function $f$ minimizing: $\mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}}\big[f(\boldsymbol{x}) \neq y\big]$

As in all the binary classification examples you've seen, you have some input space (which has always been $\mathbb{R}^D$). There is some distribution that produces labeled examples over the input space. You do not have access to that distribution, but can obtain samples from it. Your goal is to find a classifier that minimizes error on that distribution.

A small modification on this definition gives a $\alpha$-weighted classification problem, where you believe that the positive class is $\alpha$-times as

**Algorithm 11** SUBSAMPLEMAP($\mathcal{D}^{weighted}$, $\alpha$)

```
while true do
    (x, y) ~ D^weighted          // draw an example from the weighted distribution
    u ~ uniform random variable in [0, 1]
    if y = +1 or u < 1/α then
        return (x, y)
    end if
end while
```

important as the negative class.

**TASK: $\alpha$-WEIGHTED BINARY CLASSIFICATION**

*Given:*

1. An input space $\mathcal{X}$
2. An unknown distribution $\mathcal{D}$ over $\mathcal{X} \times \{-1, +1\}$

*Compute:* A function $f$ minimizing: $\mathbb{E}_{(x,y)\sim\mathcal{D}}\left[\alpha^{y=1}\left[f(x) \neq y\right]\right]$

The objects given to you in weighted binary classification are identical to standard binary classification. The only difference is that the *cost* of misprediction for $y = +1$ is $\alpha$, while the cost of misprediction for $y = -1$ is 1. In what follows, we assume that $\alpha > 1$. If it is not, you can simply swap the labels and use $1/\alpha$.

The question we will ask is: suppose that I have a good algorithm for solving the BINARY CLASSIFICATION problem. Can I turn that into a good algorithm for solving the $\alpha$-WEIGHTED BINARY CLASSIFICATION problem?

In order to do this, you need to define a *transformation* that maps a concrete weighted problem into a concrete unweighted problem. This transformation needs to happen both at training time and at test time (though it need not be the same transformation!). Algorithm **??** sketches a training-time **sub-sampling** transformation and Algorithm **??** sketches a test-time transformation (which, in this case, is trivial). All the training algorithm is doing is retaining all positive examples and a $1/\alpha$ fraction of all negative examples. The algorithm is explicitly turning the distribution over weighted examples into a (different) distribution over binary examples. A vanilla binary classifier is trained on this **induced distribution**.

Aside from the fact that this algorithm throws out a lot of data (especially for large $\alpha$), it does seem to be doing a reasonable thing. In fact, from a **reductions** perspective, it is an optimal algorithm. You can prove the following result:

**Theorem 2** (Subsampling Optimality). *Suppose the binary classifier trained in Algorithm* **??** *achieves a binary error rate of $\epsilon$. Then the error rate of the weighted predictor is equal to $\alpha\epsilon$.*

This theorem states that if your binary classifier does well (on the induced distribution), then the learned predictor will also do well (on the original distribution). Thus, we have successfully converted a weighted learning problem into a plain classification problem! The fact that the error rate of the weighted predictor is exactly $\alpha$ times more than that of the unweighted predictor is unavoidable: the error metric on which it is evaluated is $\alpha$ times bigger!

The proof of this theorem is so straightforward that we will prove it here. It simply involves some algebra on expected values.

? Why is it unreasonable to expect to be able to achieve, for instance, an error of $\sqrt{\alpha}\epsilon$, or anything that is sublinear in $\alpha$?

*Proof of Theorem* **??**. Let $\mathcal{D}^w$ be the original distribution and let $\mathcal{D}^b$ be the induced distribution. Let $f$ be the binary classifier trained on data from $\mathcal{D}^b$ that achieves a binary error rate of $\epsilon^b$ on that distribution. We will compute the expected error $\epsilon^w$ of $f$ on the weighted problem:

$$\begin{aligned}
\epsilon^w &= \mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}^w}\Big[\alpha^{y=1}\big[f(\boldsymbol{x}) \neq y\big]\Big] && (5.1)\\
&= \sum_{\boldsymbol{x}\in\mathcal{X}}\sum_{y\in\pm 1} \mathcal{D}^w(\boldsymbol{x},y)\alpha^{y=1}\big[f(\boldsymbol{x}) \neq y\big] && (5.2)\\
&= \alpha \sum_{\boldsymbol{x}\in\mathcal{X}} \Big(\mathcal{D}^w(\boldsymbol{x},+1)\big[f(\boldsymbol{x}) \neq +1\big] + \mathcal{D}^w(\boldsymbol{x},-1)\frac{1}{\alpha}\big[f(\boldsymbol{x}) \neq -1\big]\Big) && (5.3)\\
&= \alpha \sum_{\boldsymbol{x}\in\mathcal{X}} \Big(\mathcal{D}^b(\boldsymbol{x},+1)\big[f(\boldsymbol{x}) \neq +1\big] + \mathcal{D}^b(\boldsymbol{x},-1)\big[f(\boldsymbol{x}) \neq -1\big]\Big) && (5.4)\\
&= \alpha\mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}^b}\big[f(\boldsymbol{x}) \neq y\big] && (5.5)\\
&= \alpha\epsilon^b && (5.6)
\end{aligned}$$

And we're done! (We implicitly assumed $\mathcal{X}$ is discrete. In the case of continuous data, you need to replace all the sums over $\boldsymbol{x}$ with integrals over $\boldsymbol{x}$, but the result still holds.) □

Instead of subsampling the low-cost class, you could alternatively **oversample** the high-cost class. The easiest case is when $\alpha$ is an integer, say 5. Now, whenever you get a positive point, you include 5 copies of it in the induced distribution. Whenever you get a negative point, you include a single copy.

? How can you handle non-integral $\alpha$, for instance 5.5?

This oversampling algorithm achieves exactly the same theoretical result as the subsampling algorithm. The main advantage to the oversampling algorithm is that it does not throw out any data. The main advantage to the subsampling algorithm is that it is more computationally efficient.

? Modify the proof of optimality for the subsampling algorithm so that it applies to the oversampling algorithm.

You might be asking yourself: intuitively, the oversampling algorithm seems like a much better idea than the subsampling algorithm, at least if you don't care about computational efficiency. But the theory tells us that they are the same! What is going on? Of course the theory isn't wrong. It's just that the assumptions are effectively different in the two cases. Both theorems state that if you can get error of $\epsilon$ on the binary problem, you automatically get error of $\alpha\epsilon$ on the weighted problem. But they do not say anything about how possible it is to get error $\epsilon$ on the binary problem. Since the oversampling algorithm produces more data points than the subsampling algorithm it is very concievable that you could get lower binary error with oversampling than subsampling.

The primary drawback to oversampling is computational inefficiency. However, for many learning algorithms, it is straightforward to include *weighted* copies of data points at no cost. The idea is to store only the unique data points and maintain a counter saying how many times they are replicated. This is not easy to do for the perceptron (it can be done, but takes work), but it *is* easy for both decision trees and KNN. For example, for decision trees (recall Algorithm 1.3), the only changes are to: (1) ensure that line 1 computes the most frequent *weighted* answer, and (2) change lines 10 and 11 to compute weighted errors.

? Why is it hard to change the perceptron? (Hint: it has to do with the fact that perceptron is online.)

## 5.2 Multiclass Classification

Multiclass classification is a natural extension of binary classification. The goal is still to assign a discrete label to examples (for instance, is a document about entertainment, sports, finance or world news?). The difference is that you have $K > 2$ classes to choose from.

? How would you modify KNN to take into account weights?

**TASK: MULTICLASS CLASSIFICATION**

*Given:*

1. An input space $\mathcal{X}$ and number of classes $K$
2. An unknown distribution $\mathcal{D}$ over $\mathcal{X} \times [K]$

*Compute:* A function $f$ minimizing: $\mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}}\big[f(\boldsymbol{x}) \neq y\big]$

Note that this is *identical* to binary classification, except for the presence of $K$ classes. (In the above, $[K] = \{1, 2, 3, \ldots, K\}$.) In fact, if you set $K = 2$ you exactly recover binary classification.

The game we play is the same: someone gives you a binary classifier and you have to use it to solve the multiclass classification prob-

**Algorithm 12** OneVersusAllTrain($\mathbf{D}^{multiclass}$, BinaryTrain)

```
for i = 1 to K do
    D^bin ← relabel D^multiclass so class i is positive and ¬i is negative
    f_i ← BinaryTrain(D^bin)
end for
return f_1, ..., f_K
```

**Algorithm 13** OneVersusAllTest($f_1, \ldots, f_K, \hat{x}$)

```
score ← ⟨0, 0, ..., 0⟩                    // initialize K-many scores to zero
for i = 1 to K do
    y ← f_i(x̂)
    score_i ← score_i + y
end for
return argmax_k score_k
```

lem. A very common approach is the **one versus all** technique (also called **OVA** or **one versus rest**). To perform OVA, you train $K$-many binary classifiers, $f_1, \ldots, f_K$. Each classifier sees *all* of the training data. Classifier $f_i$ receives all examples labeled class $i$ as positives and all other examples as negatives. At test time, whichever classifier predicts "positive" wins, with ties broken randomly.

The training and test algorithms for OVA are sketched in Algorithms 5.2 and 5.2. In the testing procedure, the prediction of the $i$th classifier is added to the overall score for class $i$. Thus, if the prediction is positive, class $i$ gets a vote; if the prdiction is negative, everyone else (implicitly) gets a vote. (In fact, if your learning algorithm can output a confidence, as discussed in Section **??**, you can often do better by using the confidence as $y$, rather than a simple $\pm 1$.)

? Suppose that you have $N$ data points in $K$ classes, evenly divided. How long does it take to train an OVA classifier, if the base binary classifier takes $\mathcal{O}(N)$ time to train? What if the base classifier takes $\mathcal{O}(N^2)$ time?

OVA is very natural, easy to implement, and quite natural. It also works very well in practice, so long as you do a good job choosing a good binary classification algorithm *tuning* its hyperparameters well. Its weakness is that it can be somewhat brittle. Intuitively, it is not particularly robust to errors in the underlying classifiers. If *one* classifier makes a mistake, it eis possible that the entire prediction is erroneous. In fact, it is entirely possible that *none* of the $K$ classifiers predicts positive (which is actually the worst-case scenario from a theoretical perspective)! This is made explicit in the OVA error bound below.

? Why would using a confidence help.

**Theorem 3** (OVA Error Bound)**.** *Suppose the average binary error of the $K$ binary classifiers is $\epsilon$. Then the error rate of the OVA multiclass predictor is* at most $(K-1)\epsilon$.

*Proof of Theorem 3.* The key question is erroneous predictions from the binary classifiers lead to multiclass errors. We break it down into false negatives (predicting -1 when the truth is +1) and false positives

(predicting +1 when the truth is -1).

When a false negative occurs, then the testing procedure chooses randomly between available options, which is all labels. This gives a $(K-1)/K$ probability of multiclass error. Since only *one* binary error is necessary to make this happen, the *efficiency* of this error mode is $[(K-1)/K]/1 = (K-1)/K$.

Multiple false positives can occur simultaneously. Suppose there are $m$ false positives. If there is simultaneously a false negative, the error is 1. In order for this to happen, there have to be $m+1$ errors, so the efficiency is $1/(M+1)$. In the case that there is not a simultaneous false negative, the error probability is $m/(m+1)$. This requires $m$ errors, leading to an efficiency of $1/(m+1)$.

The worse case, therefore, is the false negative case, which gives an efficiency of $(K-1)/K$. Since we have $K$-many opportunities to err, we multiply this by $K$ and get a bound of $(K-1)\epsilon$. □

The constants in this are relatively unimportant: the aspect that matters is that this scales *linearly* in $K$. That is, as the number of classes grows, so does your expected error.

To develop alternative approaches, a useful way to think about turning multiclass classification problems into binary classification problems is to think of them like tournaments (football, soccer–aka football, cricket, tennis, or whatever appeals to you). You have $K$ teams entering a tournament, but unfortunately the sport they are playing only allows two to compete at a time. You want to set up a way of pairing the teams and having them compete so that you can figure out which team is best. In learning, the teams are now the classes and you're trying to figure out which class is best.[1]

[1] The sporting analogy breaks down a bit for OVA: $K$ games are played, wherein each team will play simultaneously against all other teams.

One natural approach is to have every team compete against every other team. The team that wins the majority of its matches is declared the winner. This is the **all versus all** (or **AVA**) approach (sometimes called **all pairs**). The most natural way to think about it is as training $\binom{K}{2}$ classifiers. Say $f_{ij}$ for $1 \leq i < j \leq k$ is the classifier that pits class $i$ against class $j$. This classifier receives all of the class $i$ examples as "positive" and all of the class $j$ examples as "negative." When a test point arrives, it is run through all $f_{ij}$ classifiers. Every time $f_{ij}$ predicts positive, class $i$ gets a point; otherwise, class $j$ gets a point. After running all $\binom{K}{2}$ classifiers, the class with the most votes wins.

? Suppose that you have $N$ data points in $K$ classes, evenly divided. How long does it take to train an AVA classifier, if the base binary classifier takes $\mathcal{O}(N)$ time to train? What if the base classifier takes $\mathcal{O}(N^2)$ time? How does this compare to OVA?

The training and test algorithms for AVA are sketched in Algorithms 5.2 and 5.2. In theory, the AVA mapping is more complicated than the weighted binary case. The result is stated below, but the proof is omitted.

**Theorem 4** (AVA Error Bound). *Suppose the average binary error of*

**Algorithm 14** AllVersusAllTrain($\mathbf{D}^{multiclass}$, BinaryTrain)

```
f_ij ← ∅, ∀1 ≤ i < j ≤ K
for i = 1 to K-1 do
    D^pos ← all x ∈ D^multiclass labeled i
    for j = i+1 to K do
        D^neg ← all x ∈ D^multiclass labeled j
        D^bin ← {(x, +1) : x ∈ D^pos} ∪ {(x, −1) : x ∈ D^neg}
        f_ij ← BinaryTrain(D^bin)
    end for
end for
return all f_ijs
```

**Algorithm 15** AllVersusAllTest(all $f_{ij}$, $\hat{\boldsymbol{x}}$)

```
score ← ⟨0, 0, ..., 0⟩          // initialize K-many scores to zero
for i = 1 to K-1 do
    for j = i+1 to K do
        y ← f_ij(x̂)
        score_i ← score_i + y
        score_j ← score_j - y
    end for
end for
return argmax_k score_k
```

*the* $\binom{K}{2}$ *binary classifiers is* $\epsilon$*. Then the error rate of the AVA multiclass predictor is* at most $2(K-1)\epsilon$.

? The bound for AVA is $2(K-1)\epsilon$; the bound for OVA is $(K-1)\epsilon$. Does this mean that OVA is necessarily better than AVA? Why or why not?

At this point, you might be wondering if it's possible to do better than something linear in $K$. Fortunately, the answer is yes! The solution, like so much in computer science, is divide and conquer. The idea is to construct a *binary tree* of classifiers. The leaves of this tree correspond to the $K$ labels. Since there are only $\log_2 K$ decisions made to get from the root to a leaf, then there are only $\log_2 K$ chances to make an error.

An example of a classification tree for $K = 8$ classes is shown in Figure 5.2. At the root, you distinguish between classes $\{1,2,3,4\}$ and classes $\{5,6,7,8\}$. This means that you will train a binary classifier whose positive examples are all data points with multiclass label $\{1,2,3,4\}$ and whose negative examples are all data points with multiclass label $\{5,6,7,8\}$. Based on what decision is made by this classifier, you can walk down the appropriate path in the tree. When $K$ is not a powwr of 2, the tree will not be full. This classification tree algorithm achieves the following bound.

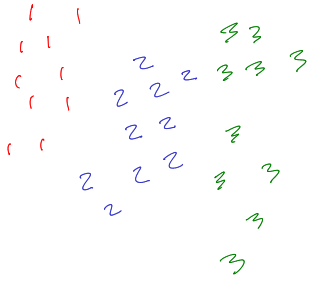


Figure 5.1: data set on which OVA will do terribly with linear classifiers

? Consider the data in Figure 5.1 and assume that you are using a perceptron as the base classifier. How well will OVA do on this data? What about AVA?

**Theorem 5** (Tree Error Bound)**.** *Suppose the average binary classifiers error is* $\epsilon$*. Then the error rate of the tree classifier is* at most $\lceil \log_2 K \rceil \epsilon$.

*Proof of Theorem 5.* A multiclass error is made if any classifier on

the path from the root to the correct leaf makes an error. Each has probability $\epsilon$ of making an error and the path consists of at most $\lceil \log_2 K \rceil$ binary decisions. □

One think to keep in mind with tree classifiers is that you have control over how the tree is defined. In OVA and AVA you have no say in what classification problems are created. In tree classifiers, the only thing that matters is that, at the root, half of the classes are considered positive and half are considered negative. You want to split the classes in such a way that this classification decision is as easy as possible. You can use whatever you happen to know about your classification problem to try to separate the classes out in a reasonable way.

Can you do better than $\lceil \log_2 K \rceil \epsilon$? It turns out the answer is yes, but the algorithms to do so are relatively complicated. You can actually do as well as $2\epsilon$ using the idea of error-correcting tournaments. Moreover, you can prove a *lower bound* that states that the best you could possible do is $\epsilon/2$. This means that error-correcting tournaments are at most a factor of four worse than optimal.

## 5.3 Ranking

You start a new web search company called Goohooing. Like other search engines, a user inputs a query and a set of documents is retrieved. Your goal is to rank the resulting documents based on relevance to the query. The ranking problem is to take a collection of items and sort them according to some notion of preference. One of the trickiest parts of doing ranking through learning is to properly define the loss function. Toward the end of this section you will see a very general loss function, but before that let's consider a few special cases.

Continuing the web search example, you are given a collection of queries. For each query, you are also given a collection of documents, together with a desired ranking over those documents. In the following, we'll assume that you have $N$-many queries and for each query you have $M$-many documents. (In practice, $M$ will probably vary by query, but for ease we'll consider the simplified case.) The goal is to train a binary classifier to predict a **preference function**. Given a query $q$ and two documents $d_i$ and $d_j$, the classifier should predict whether $d_i$ should be preferred to $d_j$ with respect to the query $q$.

As in all the previous examples, there are two things we have to take care of: (1) how to train the classifier that predicts preferences; (2) how to turn the predicted preferences into a ranking. Unlike the previous examples, the second step is somewhat complicated in the

**Algorithm 16** NAIVERANKTRAIN(*RankingData*, BINARYTRAIN)

```
D ← [ ]
for n = 1 to N do
  for all i, j = 1 to M and i ≠ j do
    if i is prefered to j on query n then
      D ← D ⊕ (x_nij, +1)
    else if j is prefered to i on query n then
      D ← D ⊕ (x_nij, −1)
    end if
  end for
end for
return BINARYTRAIN(D)
```

**Algorithm 17** NAIVERANKTEST($f$, $\hat{\boldsymbol{x}}$)

```
score ← ⟨0, 0, ..., 0⟩          // initialize M-many scores to zero
for all i, j = 1 to M and i ≠ j do
  y ← f(x̂_ij)                    // get predicted ranking of i and j
  score_i ← score_i + y
  score_j ← score_j - y
end for
return ARGSORT(score)            // return queries sorted by score
```

ranking case. This is because we need to predict an entire ranking of a large number of documents, somehow assimilating the preference function into an overall permutation.

For notationally simplicity, let $\boldsymbol{x}_{nij}$ denote the features associated with comparing document $i$ to document $j$ on query $n$. Training is fairly straightforward. For every $n$ and every pair $i \neq j$, we will create a binary classification example based on features $\boldsymbol{x}_{nij}$. This example is positive if $i$ is preferred to $j$ in the true ranking. It is negative if $j$ is preferred to $i$. (In some cases the true ranking will not express a preference between two objects, in which case we exclude the $i, j$ and $j, i$ pair from training.)

Now, you might be tempted to evaluate the classification performance of this binary classifier on its own. The problem with this approach is that it's impossible to tell—just by looking at its output on one $i, j$ pair—how good the overall ranking is. This is because there is the intermediate step of turning these pairwise predictions into a coherent ranking. What you need to do is measure how well the ranking based on your predicted preferences compares to the true ordering. Algorithms 5.3 and 5.3 show naive algorithms for training and testing a ranking function.

These algorithms actually work quite well in the case of **bipartite ranking problems**. A bipartite ranking problem is one in which you are only ever trying to predict a binary response, for instance "is this

document relevant or not?" but are being evaluated according to a metric like **AUC**. This is essentially because the only goal in bipartite problems to to ensure that all the relevant documents are ahead of all the irrelevant documents. There is no notion that one relevant document is *more relevant* than another.

For non-bipartite ranking problems, you can do better. First, when the preferences that you get at training time are more nuanced than "relevant or not," you can incorporate these preferences at training time. Effectively, you want to give a higher weight to binary problems that are very different in terms of perference than others. Second, rather than producing a list of scores and then calling an arbitrary sorting algorithm, you can actually use the preference function as the sorting function inside your own implementation of quicksort.

We can now formalize the problem. Define a ranking as a function $\sigma$ that maps the objects we are ranking (documents) to the desired position in the list, $1, 2, \dots M$. If $\sigma_u < \sigma_v$ then $u$ is preferred to $v$ (i.e., appears earlier on the ranked document list). Given data with observed rankings $\sigma$, our goal is to learn to predict rankings for new objects, $\hat{\sigma}$. We define $\Sigma_M$ as the set of all ranking functions over $M$ objects. We also wish to express the fact that making a mistake on some pairs is worse than making a mistake on others. This will be encoded in a cost function $\omega$ (omega), where $\omega(i, j)$ is the cost for accidentally putting something in position $j$ when it should have gone in position $i$. To be a valid cost function valid, $\omega$ must be (1) symmetric, (2) monotonic and (3) satisfy the triangle inequality. Namely: (1) $\omega(i, j) = \omega(j, i)$; (2) if $i < j < k$ or $i > j > k$ then $\omega(i, j) \leq \omega(i, k)$; (3) $\omega(i, j) + \omega(j, k) \geq \omega(i, k)$. With these definitions, we can properly define the ranking problem.

**TASK: $\omega$-RANKING**

*Given:*

1. An input space $\mathcal{X}$
2. An unknown distribution $\mathcal{D}$ over $\mathcal{X} \times \Sigma_M$

*Compute:* A function $f : \mathcal{X} \to \Sigma_M$ minimizing:

$$\mathbb{E}_{(\boldsymbol{x}, \sigma) \sim \mathcal{D}} \left[ \sum_{u \neq v} [\sigma_u < \sigma_v] \, [\hat{\sigma}_v < \hat{\sigma}_u] \, \omega(\sigma_u, \sigma_v) \right] \tag{5.7}$$

where $\hat{\sigma} = f(\boldsymbol{x})$

In this definition, the only complex aspect is the loss function 5.7. This loss sums over all pairs of objects $u$ and $v$. If the true ranking ($\sigma$)

**Algorithm 18** RANKTRAIN($\mathbf{D}^{rank}$, $\omega$, BINARYTRAIN)

```
1: D^bin ← [ ]
2: for all (x, σ) ∈ D^rank do
3:     for all u ≠ v do
4:         y ← sign(σ_v - σ_u)                    // y is +1 if u is prefered to v
5:         w ← ω(σ_u, σ_v)                        // w is the cost of misclassification
6:         D^bin ← D^bin ⊕ (y, w, x_uv)
7:     end for
8: end for
9: return BinaryTrain(D^bin)
```

prefers $u$ to $v$, but the predicted ranking ($\hat{\sigma}$) prefers $v$ to $u$, then you incur a cost of $\omega(\sigma_u, \sigma_v)$.

Depending on the problem you care about, you can set $\omega$ to many "standard" options. If $\omega(i,j) = 1$ whenever $i \neq j$, then you achieve the Kemeny distance measure, which simply counts the number of pairwise misordered items. In many applications, you may only care about getting the top $K$ predictions correct. For instance, your web search algorithm may only display $K = 10$ results to a user. In this case, you can define:

$$\omega(i,j) = \begin{cases} 1 & \text{if } \min\{i,j\} \leq K \text{ and } i \neq j \\ 0 & \text{otherwise} \end{cases} \tag{5.8}$$

In this case, only errors in the top $K$ elements are penalized. Swapping items 55 and 56 is irrelevant (for $K < 55$).

Finally, in the bipartite ranking case, you can express the **area under the curve** (**AUC**) metric as:

$$\omega(i,j) = \frac{\binom{M}{2}}{M^+(M-M^+)} \times \begin{cases} 1 & \text{if } i \leq M^+ \text{ and } j > M^+ \\ 1 & \text{if } j \leq M^+ \text{ and } i > M^+ \\ 0 & \text{otherwise} \end{cases} \tag{5.9}$$

Here, $M$ is the total number of objects to be ranked and $M^+$ is the number that are actually "good." (Hence, $M - M^+$ is the number that are actually "bad," since this is a bipartite problem.) You are only penalized if you rank a good item in position greater than $M^+$ or if you rank a bad item in a position less than or equal to $M^+$.

In order to *solve* this problem, you can follow a recipe similar to the naive approach sketched earlier. At training time, the biggest change is that you can *weight* each training example by how bad it would be to mess it up. This change is depicted in Algorithm 5.3, where the binary classiciation data has *weights* $w$ provided for saying how important a given example is. These weights are derived from the cost function $\omega$.

At test time, instead of predicting scores and then sorting the list, you essentially run the quicksort algorith, using $f$ as a comparison

**Algorithm 19** RANKTEST($f$, $\hat{\boldsymbol{x}}$, $obj$)

```
if obj contains 0 or 1 elements then
    return obj
else
    p ← randomly chosen object in obj                          // pick pivot
    left ← [ ]                                  // elements that seem smaller than p
    right ← [ ]                                 // elements that seem larger than p
    for all u ∈ obj \{p} do
        ŷ ← f(x_up)                             // what is the probability that u precedes p
        if uniform random variable < ŷ then
            left ← left ⊕ u
        else
            right ← right ⊕ u
        end if
    end for
    left ← RANKTEST(f, x̂, left)                                // sort earlier elements
    right ← RANKTEST(f, x̂, right)                              // sort later elements
    return left ⊕ ⟨p⟩ ⊕ right
end if
```

function. At each step in Algorithm 5.3, a pivot $p$ is chosen. Every other object $u$ is compared to $p$ using $f$. If $f$ thinks $u$ is better, then it is sorted on the left; otherwise it is sorted on the right. There is one major difference between this algorithmand quicksort: the comparison function is allowed to be *probabilistic*. If $f$ outputs probabilities, for instance it predicts that $u$ has an 80% probability of being better than $p$, then it puts it on the left with 80% probability and on the right with 20% probability. (The pseudocode is written in such a way that even if $f$ just predicts $-1, +1$, the algorithm still works.)

This algorithm is better than the naive algorithm in at least two ways. First, it only makes $\mathcal{O}(M \log_2 M)$ calls to $f$ (in expectation), rather than $\mathcal{O}(M^2)$ calls in the naive case. Second, it achieves a better error bound, shown below:

**Theorem 6** (Rank Error Bound)**.** *Suppose the average binary error of $f$ is $\epsilon$. Then the ranking algorithm achieves a test error of at most $2\epsilon$ in the general case, and $\epsilon$ in the bipartite case.*


Figure 5.3: example face finding image and pixel mask

## 5.4 Collective Classification

You are writing new software for a digital camera that does face identification. However, instead of simply finding a bounding box around faces in an image, you must predict where a face is *at the pixel level*. So your input is an image (say, 100×100 pixels: this is a really low resolution camera!) and your output is a set of 100×100 binary predictions about each pixel. You are given a large collection

of training examples. An example input/output pair is shown in Figure 5.3.

Your first attempt might be to train a binary classifier to predict whether pixel $(i, j)$ is part of a face or not. You might feed in features to this classifier about the RGB values of pixel $(i, j)$ as well as pixels in a window arround that. For instance, pixels in the region $\{(i + k, j + l) : k \in [-5, 5], l \in [-5, 5]\}$.

You run your classifier and notice that it predicts weird things, like what you see in Figure 5.4. You then realize that predicting each pixel independently is a bad idea! If pixel $(i, j)$ is part of a face, then this significantly increases the chances that pixel $(i + 1, j)$ is also part of a face. (And similarly for other pixels.) This is a **collective classification** problem because you are trying to predict multiple, correlated objects at the same time.

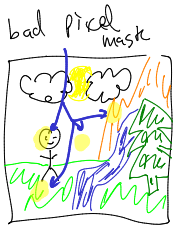



Figure 5.4: bad pixel mask for previous image

? Similar problems come up all the time. Cast the following as collective classification problems: web page categorization; labeling words in a sentence as noun, verb, adjective, etc.; finding genes in DNA sequences; predicting the stock market.

The most general way to formulate these problems is as (undirected) **graph** prediction problems. Our input now takes the form of a graph, where the vertices are input/output pairs and the edges represent the correlations among the putputs. (Note that edges do not need to express correlations among the inputs: these can simply be encoded on the nodes themselves.) For example, in the face identification case, each pixel would correspond to an vertex in the graph. For the vertex that corresponds to pixel $(5, 10)$, the input would be whatever set of features we want about that pixel (including features about neighboring pixels). There would be edges between that vertex and (for instance) vertices $(4, 10)$, $(6, 10)$, $(5, 9)$ and $(5, 11)$. If we are predicting one of $K$ classes at each vertex, then we are given a graph whose vertices are labeled by pairs $(x, k) \in \mathcal{X} \times [K]$. We will write $\mathcal{G}(\mathcal{X} \times [K])$ to denote the set of all such graphs. A graph in this set is denoted as $G = (V, E)$ with vertices $V$ and edges $E$. Our goal is a function $f$ that takes as input a graph from $\mathcal{G}(\mathcal{X})$ and predicts a label from $[K]$ for each of its vertices.

? Formulate the example problems above as graph prediction problems.

**TASK: COLLECTIVE CLASSIFICATION**

*Given:*

1. An input space $\mathcal{X}$ and number of classes $K$
2. An unknown distribution $\mathcal{D}$ over $\mathcal{G}(\mathcal{X} \times [K])$

*Compute:* A function $f : \mathcal{G}(\mathcal{X}) \to \mathcal{G}([K])$ minimizing: $\mathbb{E}_{(V,E)\sim\mathcal{D}} \left[ \sum_{v \in V} \left[ \hat{y}_v \neq y_v \right] \right]$, where $y_v$ is the label associated with vertex $v$ in $G$ and $\hat{y}_v$ is the label predicted by $f(G)$.

In collective classification, you would like to be able to use the

labels of neighboring vertices to help predict the label of a given vertex. For instance, you might want to add features to the predict of a given vertex based on the labels of each neighbor. At training time, this is easy: you get to see the true labels of each neighbor. However, at test time, it is much more difficult: you are, yourself, predicting the labels of each neighbor.

This presents a chicken and egg problem. You are trying to predict a collection of labels. But the prediction of each label depends on the prediction of other labels. If you remember from before, a general solution to this problem is iteration: you can begin with some guesses, and then try to improve these guesses over time. [2]

This is the idea of **stacking** for solving collective classification (see Figure 5.5. You can train 5 classifiers. The first classifier *just* predicts the value of each pixel independently, like in Figure 5.4. This doesn't use any of the graph structure at all. In the second level, you can repeat the classification. However, you can use the outputs from the first level as initial guesses of labels. In general, for the $K$th level in the stack, you can use the inputs (pixel values) as well as the predictions for all of the $K-1$ previous levels of the stack. This means training $K$-many binary classifiers based on different feature sets.

The prediction technique for stacking is sketched in Algorithm 5.4. This takes a list of $K$ classifiers, corresponding to each level in the stack, and an input graph $G$. The variable $\hat{Y}_{k,v}$ stores the prediction of classifier $k$ on vertex $v$ in the graph. You first predict every node in the vertex using the first layer in the stack, and no neighboring information. For the rest of the layers, you add on features to each node based on the predictions made by lower levels in the stack for neighboring nodes ($\mathcal{N}(u)$ denotes the neighbors of $u$).

The training procedure follows a similar scheme, sketched in Algorithm 5.4. It largely follows the same schematic as the prediction algorithm, but with training fed in. After the classifier for the $k$ level has been trained, it is used to predict labels on every node in the graph. These labels are used by later levels in the stack, as features.

One thing to be aware of is that MULTICLASSTRAIN could conceivably overfit its training data. For example, it is possible that the first layer might actually achieve 0% error, in which case there is no reason to iterate. But at test time, it will probably *not* get 0% error, so this is misleading. There are (at least) two ways to address this issue. The first is to use cross-validation during training, and to use the predictions obtained during cross-validation as the predictions from **StackTest**. This is typically very safe, but somewhat expensive. The alternative is to simply *over-regularize* your training algorithm. In particular, instead of trying to find hyperparameters that get the

[2] Alternatively, the fact that we're using a graph might scream to you "dynamic programming." Rest assured that you can do this too: skip forward to Chapter 18 for lots more detail here!

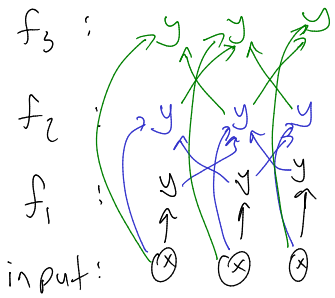



Figure 5.5: a charicature of how stacking works

**Algorithm 20** StackTrain($\mathcal{D}^{cc}$, $K$, MulticlassTrain)

$\mathbf{D}^{mc} \leftarrow [\,]$ // our generated multiclass data
$\hat{Y}_{k,n,v} \leftarrow o, \forall k \in [K], n \in [N], v \in G_n$ // initialize predictions for all levels
**for** $k = 1$ **to** $K$ **do**
  **for** $n = 1$ **to** $N$ **do**
    **for all** $v \in G_n$ **do**
      $(\boldsymbol{x}, y) \leftarrow$ features and label for node $v$
      $\boldsymbol{x} \leftarrow \boldsymbol{x} \oplus \hat{Y}_{l,n,u}, \forall u \in \mathcal{N}(u), \forall l \in [k-1]$ // add on features for
        // neighboring nodes from lower levels in the stack
      $\mathbf{D}^{mc} \leftarrow \mathbf{D}^{mc} \oplus (y, \boldsymbol{x})$ // add to multiclass data
    **end for**
  **end for**
  $f_k \leftarrow$ MulticlassTrain($\mathbf{D}^{bin}$) // train $k$th level classifier
  **for** $n = 1$ **to** $N$ **do**
    $\hat{Y}_{k,n,v} \leftarrow$ StackTest($f_1, \ldots, f_k, G_n$) // predict using $k$th level classifier
  **end for**
**end for**
**return** $f_1, \ldots, f_K$ // return all classifiers

**Algorithm 21** StackTest($f_1, \ldots, f_K, G$)

$\hat{Y}_{k,v} \leftarrow o, \forall k \in [K], v \in G$ // initialize predictions for all levels
**for** $k = 1$ **to** $K$ **do**
  **for all** $v \in G$ **do**
    $\boldsymbol{x} \leftarrow$ features for node $v$
    $\boldsymbol{x} \leftarrow \boldsymbol{x} \oplus \hat{Y}_{l,u}, \forall u \in \mathcal{N}(u), \forall l \in [k-1]$ // add on features for
      // neighboring nodes from lower levels in the stack
    $\hat{Y}_{k,v} \leftarrow f_k(\boldsymbol{x})$ // predict according to $k$th level
  **end for**
**end for**
**return** $\{\hat{Y}_{K,v} : v \in G\}$ // return predictions for every node from the last layer

*best* development data performance, try to find hyperparameters that make your *training* performance approximately equal to your *development* performance. This will ensure that your predictions at the $k$th layer are indicative of how well the algorithm will actually do at test time.

TODO: finish this discussion

## 5.5 Exercises

**Exercise 5.1.** **TODO...**

# 6 | Linear Models

> The essence of mathematics is not to make simple things complicated, but to make complicated things simple. – Stanley Gudder

**Learning Objectives:**
- Define and plot four surrogate loss functions: squared loss, logistic loss, exponential loss and hinge loss.
- Compare and contrast the optimization of 0/1 loss and surrogate loss functions.
- Solve the optimization problem for squared loss with a quadratic regularizer in closed form.
- Implement and debug gradient descent and subgradient descent.

Dependencies:

In Chapter ??, you learned about the perceptron algorithm for linear classification. This was both a *model* (linear classifier) and *algorithm* (the perceptron update rule) in one. In this section, we will separate these two, and consider general ways for optimizing linear models. This will lead us into some aspects of optimization (aka mathematical programming), but not very far. At the end of this chapter, there are pointers to more literature on optimization for those who are interested.

The basic idea of the perceptron is to run a particular algorithm until a linear separator is found. You might ask: are there better algorithms for finding such a linear separator? We will follow this idea and formulate a learning problem as an explicit optimization problem: find me a linear separator that is not too complicated. We will see that finding an "optimal" separator is actually computationally prohibitive, and so will need to "relax" the optimality requirement. This will lead us to a **convex** objective that combines a loss function (how well are we doing on the training data?) and a regularizer (how complicated is our learned model?). This learning framework is known as both **Tikhonov regularization** and **structural risk minimization**.

## 6.1 The Optimization Framework for Linear Models

You have already seen the perceptron as a way of finding a weight vector $\boldsymbol{w}$ and bias $b$ that do a good job of separating positive training examples from negative training examples. The perceptron is a **model** and **algorithm** in one. Here, we are interested in *separating* these issues. We will focus on linear models, like the perceptron. But we will think about other, more generic ways of finding good parameters of these models.

The goal of the perceptron was to find a **separating hyperplane** for some training data set. For simplicity, you can ignore the issue of overfitting (but just for now!). Not all data sets are linearly sepa-

rable. In the case that your training data *isn't* linearly separable, you might want to find the hyperplane that makes the *fewest errors* on the training data. We can write this down as a formal mathematics **optimization problem** as follows:

$$\min_{\boldsymbol{w},b} \quad \sum_n \mathbf{1}[y_n(\boldsymbol{w}\cdot\boldsymbol{x}_n + b) > 0] \tag{6.1}$$

In this expression, you are optimizing over two variables, $\boldsymbol{w}$ and $b$. The **objective function** is the thing you are trying to minimize. In this case, the objective function is simply the **error rate** (or **0/1 loss**) of the linear classifier parameterized by $\boldsymbol{w}, b$. In this expression, $\mathbf{1}[\cdot]$ is the **indicator function**: it is one when $(\cdot)$ is true and zero otherwise.

? You should remember the $y\boldsymbol{w}\cdot\boldsymbol{x}$ trick from the perceptron discussion. If not, re-convince yourself that this is doing the right thing.

We know that the perceptron algorithm is guaranteed to find parameters for this model if the data is linearly separable. In other words, if the optimum of Eq (6.1) is zero, then the perceptron will efficiently find parameters for this model. The notion of "efficiency" depends on the margin of the data for the perceptron.

You might ask: what happens if the data is *not* linearly separable? Is there an efficient algorithm for finding an optimal setting of the parameters? Unfortunately, the answer is *no.* There is no polynomial time algorithm for solving Eq (6.1), unless P=NP. In other words, this problem is NP-hard. Sadly, the proof of this is quite complicated and beyond the scope of this book, but it relies on a reduction from a variant of satisfiability. The key idea is to turn a satisfiability problem into an optimization problem where a clause is satisfied exactly when the hyperplane correctly separates the data.

You might then come back and say: okay, well I don't really need an *exact* solution. I'm willing to have a solution that makes one or two more errors than it has to. Unfortunately, the situation is really bad. Zero/one loss is NP-hard to even *appproximately minimize*. In other words, there is no efficient algorithm for even finding a solution that's a small constant worse than optimal. (The best known constant at this time is $418/415 \approx 1.007$.)

However, before getting too disillusioned about this whole enterprise (remember: there's an entire chapter about this framework, so it must be going somewhere!), you should remember that optimizing Eq (6.1) perhaps isn't even what you want to do! In particular, all it says is that you will get minimal *training error.* It says nothing about what your *test error* will be like. In order to try to find a solution that will *generalize* well to test data, you need to ensure that you do not overfit the data. To do this, you can introduce a **regularizer** over the parameters of the model. For now, we will be vague about what this regularizer looks like, and simply call it an arbitrary function $R(\boldsymbol{w}, b)$.

This leads to the following, **regularized objective**:

$$\min_{w,b} \quad \sum_n \mathbf{1}[y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) > 0] + \lambda R(\boldsymbol{w}, b) \tag{6.2}$$

In Eq (6.2), we are now trying to optimize a *trade-off* between a solution that gives low training error (the first term) and a solution that is "simple" (the second term). You can think of the maximum depth hyperparameter of a decision tree as a form of regularization for trees. Here, $R$ is a form of regularization for hyperplanes. In this formulation, $\lambda$ becomes a **hyperparameter** for the optimization.

The key remaining questions, given this formalism, are:

- How can we adjust the optimization problem so that there *are* efficient algorithms for solving it?
- What are good regularizers $R(\boldsymbol{w}, b)$ for hyperplanes?
- Assuming we can adjust the optimization problem appropriately, what algorithms exist for efficiently solving this regularized optimization problem?

We will address these three questions in the next sections.

? Assuming $R$ does the "right thing," what value(s) of $\lambda$ will lead to overfitting? What value(s) will lead to underfitting?

## 6.2 Convex Surrogate Loss Functions

You might ask: why is optimizing zero/one loss so hard? Intuitively, one reason is that small changes to $\boldsymbol{w}, b$ can have a large impact on the value of the objective function. For instance, if there is a positive training example with $\boldsymbol{w}, \boldsymbol{x} \cdot + b = -0.0000001$, then adjusting $b$ upwards by 0.00000011 will decrease your error rate by 1. But adjusting it upwards by 0.00000009 will have no effect. This makes it really difficult to figure out good ways to adjust the parameters.

To see this more clearly, it is useful to look at plots that relate *margin* to *loss*. Such a plot for zero/one loss is shown in Figure 6.1. In this plot, the horizontal axis measure the margin of a data point and the vertical axis measures the loss associated with that margin. For zero/one loss, the story is simple. If you get a positive margin (i.e., $y(\boldsymbol{w} \cdot \boldsymbol{x} + b) > 0$) then you get a loss of zero. Otherwise you get a loss of one. By thinking about this plot, you can see how changes to the parameters that change the margin *just a little bit* can have an enormous effect on the overall loss.

You might decide that a reasonable way to address this problem is to replace the non-smooth zero/one loss with a smooth approximation. With a bit of effort, you could probably concoct an "S"-shaped function like that shown in Figure 6.2. The benefit of using such an S-function is that it is smooth, and potentially easier to optimize. The difficulty is that it is not **convex**.

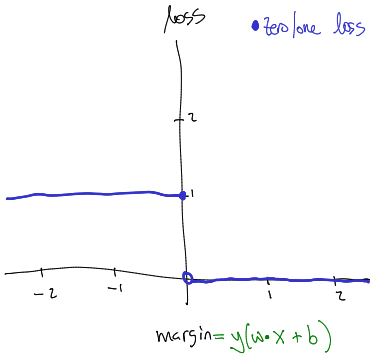



Figure 6.1: plot of zero/one versus margin

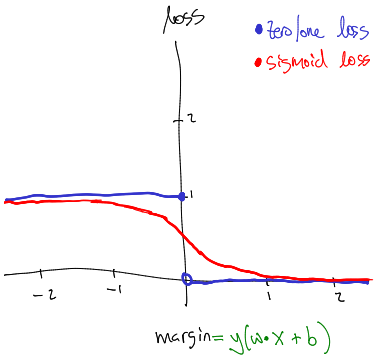



Figure 6.2: plot of zero/one versus margin and an S version of it

If you remember from calculus, a convex function is one that looks like a happy face ($\smile$). (On the other hand, a **concave** function is one that looks like a sad face ($\frown$); an easy mnemonic is that you can hide under a con**cave** function.) There are two equivalent definitions of a concave function. The first is that it's second derivative is always non-negative. The second, more geometric, defition is that any **chord** of the function lies above it. This is shown in Figure **??**. There you can see a convex function and a non-convex function, both with two chords drawn in. In the case of the convex function, the chords lie above the function. In the case of the non-convex function, there are parts of the chord that lie below the function.

Convex functions are nice because they are *easy to minimize*. Intuitively, if you drop a ball anywhere in a convex function, it will eventually get to the minimum. This is not true for non-convex functions. For example, if you drop a ball on the very left end of the S-function from Figure 6.2, it will not go anywhere.

This leads to the idea of **convex surrogate loss functions**. Since zero/one loss is hard to optimize, you want to optimize something else, instead. Since convex functions are easy to optimize, we want to approximate zero/one loss with a convex function. This approximating function will be called a **surrogate loss**. The surrogate losses we construct will always be *upper bounds* on the true loss function: this guarantees that if you minimize the surrogate loss, you are also pushing down the real loss.

There are four common surrogate loss function, each with their own properties: **hinge loss**, **logistic loss**, **exponential loss** and **squared loss**. These are shown in Figure 6.4 and defined below. These are defined in terms of the true label $y$ (which is just $\{-1, +1\}$) and the predicted value $\hat{y} = \boldsymbol{w} \cdot \boldsymbol{x} + b$.

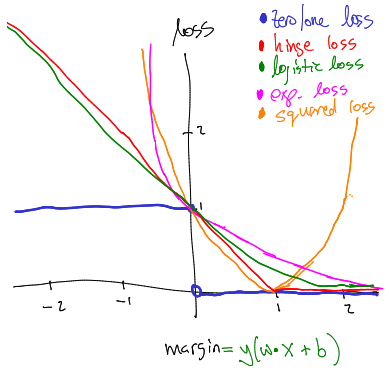



Figure 6.4: surrogate loss fns

$$\text{Zero/one:} \quad \ell^{(0/1)}(y, \hat{y}) = \mathbf{1}[y\hat{y} \leq 0] \tag{6.3}$$

$$\text{Hinge:} \quad \ell^{(\text{hin})}(y, \hat{y}) = \max\{0, 1 - y\hat{y}\} \tag{6.4}$$

$$\text{Logistic:} \quad \ell^{(\log)}(y, \hat{y}) = \frac{1}{\log 2} \log\left(1 + \exp[-y\hat{y}]\right) \tag{6.5}$$

$$\text{Exponential:} \quad \ell^{(\exp)}(y, \hat{y}) = \exp[-y\hat{y}] \tag{6.6}$$

$$\text{Squared:} \quad \ell^{(\text{sqr})}(y, \hat{y}) = (y - \hat{y})^2 \tag{6.7}$$

In the definition of logistic loss, the $\frac{1}{\log 2}$ term out front is there simply to ensure that $\ell^{(\log)}(y, 0) = 1$. This ensures, like all the other surrogate loss functions, that logistic loss upper bounds the zero/one loss. (In practice, people typically omit this constant since it does not affect the optimization.)

There are two big differences in these loss functions. The first difference is how "upset" they get by erroneous predictions. In the

case of hinge loss and logistic loss, the growth of the function as $\hat{y}$ goes negative is linear. For squared loss and exponential loss, it is super-linear. This means that exponential loss would rather get a few examples a little wrong than one example really wrong. The other difference is how they deal with very confident correct predictions. Once $y\hat{y} > 1$, hinge loss does not care any more, but logistic and exponential still think you can do better. On the other hand, squared loss thinks it's just as bad to predict $+3$ on a positive example as it is to predict $-1$ on a positive example.

## 6.3 Weight Regularization

In our learning objective, Eq (**??**), we had a term correspond to the zero/one loss on the training data, plus a **regularizer** whose goal was to ensure that the learned function didn't get too "crazy." (Or, more formally, to ensure that the function did not overfit.) If you replace to zero/one loss with a surrogate loss, you obtain the following objective:

$$\min_{\boldsymbol{w},b} \quad \sum_n \ell(y_n, \boldsymbol{w} \cdot \boldsymbol{x}_n + b) + \lambda R(\boldsymbol{w}, b) \tag{6.8}$$

The question is: what should $R(\boldsymbol{w}, b)$ look like?

From the discussion of surrogate loss function, we would like to ensure that $R$ is convex. Otherwise, we will be back to the point where optimization becomes difficult. Beyond that, a common desire is that the components of the weight vector (i.e., the $w_d$s) should be small (close to zero). This is a form of **inductive bias**.

Why are small values of $w_d$ good? Or, more precisely, why do small values of $w_d$ correspond to *simple functions*? Suppose that we have an example $\boldsymbol{x}$ with label $+1$. We might believe that other examples, $\boldsymbol{x}'$ that are nearby $\boldsymbol{x}$ should also have label $+1$. For example, if I obtain $\boldsymbol{x}'$ by taking $\boldsymbol{x}$ and changing the first component by some small value $\epsilon$ and leaving the rest the same, you might think that the classification would be the same. If you do this, the difference between $\hat{y}$ and $\hat{y}'$ will be exactly $\epsilon w_1$. So if $w_1$ is reasonably small, this is unlikely to have much of an effect on the classification decision. On the other hand, if $w_1$ is large, this could have a large effect.

Another way of saying the same thing is to look at the derivative of the predictions as a function of $w_1$. The derivative of $\boldsymbol{w}, \boldsymbol{x} \cdot + b$ with respect to $w_1$ is:

$$\frac{\partial \boldsymbol{w}, \boldsymbol{x} \cdot + b}{\partial w_1} = \frac{\partial \left[\sum_d w_d x_d + b\right]}{\partial w_1} = x_1 \tag{6.9}$$

Interpreting the derivative as the rate of change, we can see that the rate of change of the prediction function is proportional to the

individual weights. So if you want the function to change slowly, you want to ensure that the weights stay small.

One way to accomplish this is to simply use the norm of the weight vector. Namely $R^{(\text{norm})}(\boldsymbol{w}, b) = ||\boldsymbol{w}|| = \sqrt{\sum_d w_d^2}$. This function is convex and smooth, which makes it easy to minimize. In practice, it's often easier to use the squared norm, namely $R^{(\text{sqr})}(\boldsymbol{w}, b) = ||\boldsymbol{w}||^2 = \sum_d w_d^2$ because it removes the ugly square root term and remains convex. An alternative to using the sum of squared weights is to use the sum of absolute weights: $R^{(\text{abs})}(\boldsymbol{w}, b) = \sum_d |w_d|$. Both of these norms are convex.

? Why do we not regularize the bias term $b$?

In addition to small weights being good, you could argue that *zero* weights are better. If a weight $w_d$ goes to zero, then this means that feature $d$ is not used at all in the classification decision. If there are a large number of irrelevant features, you might want as many weights to go to zero as possible. This suggests an alternative regularizer: $R^{(\text{cnt})}(\boldsymbol{w}, b) = \sum_d \mathbf{1}[x_d \neq 0]$.

This line of thinking leads to the general concept of **$p$-norms**. (Technically these are called $\ell_p$ (or "ell $p$") norms, but this notation clashes with the use of $\ell$ for "loss.") This is a family of norms that all have the same general flavor. We write $||\boldsymbol{w}||_p$ to denote the $p$-norm of $\boldsymbol{w}$.

? Why might you not want to use $R^{(\text{cnt})}$ as a regularizer?

$$||\boldsymbol{w}||_p = \left( \sum_d |w_d|^p \right)^{\frac{1}{p}} \tag{6.10}$$

You can check that the 2-norm exactly corresponds to the usual Euclidean norm, and that the 1-norm corresponds to the "absolute" regularizer described above.

? You can actually identify the $R^{(\text{cnt})}$ regularizer with a $p$-norm as well. Which value of $p$ gives it to you? (Hint: you may have to take a limit.)

When $p$-norms are used to regularize weight vectors, the interesting aspect is how they trade-off multiple features. To see the behavior of $p$-norms in two dimensions, we can plot their **contour** (or **level-set**). Figure 6.5 shows the contours for the same $p$ norms in two dimensions. Each line denotes the two-dimensional vectors to which this norm assignes a total value of 1. By changing the value of $p$, you can interpolate between a square (the so-called "max norm"), down to a circle (2-norm), diamond (1-norm) and pointy-star-shaped-thing ($p < 1$ norm).

In general, smaller values of $p$ "prefer" sparser vectors. You can see this by noticing that the contours of small $p$-norms "stretch" out along the axes. It is for this reason that small $p$-norms tend to yield weight vectors with many zero entries (aka **sparse** weight vectors). Unfortunately, for $p < 1$ the norm becomes non-convex. As you might guess, this means that the 1-norm is a popular choice for sparsity-seeking applications.

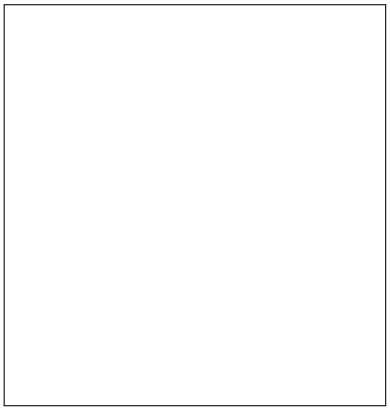


Figure 6.5: `loss:norms2d`: level sets of the same $p$-norms

? The max norm corresponds to $\lim_{p \to \infty}$. Why is this called the max norm?

**MATH REVIEW | GRADIENTS**

... be sure to do enough to do the closed for squared error

Figure 6.6:

**Algorithm 22** GRADIENTDESCENT($\mathcal{F}$, $K$, $\eta_1$, ...)

```
z^(0) ← ⟨0, 0, ..., 0⟩                          // initialize variable we are optimizing
for k = 1 ... K do
    g^(k) ← ∇_z F |_{z^(k-1)}                   // compute gradient at current location
    z^(k) ← z^(k-1) − η^(k) g^(k)               // take a step down the gradient
end for
return z^(K)
```

## 6.4 Optimization with Gradient Descent

Envision the following problem. You're taking up a new hobby: blindfolded mountain climbing. Someone blindfolds you and drops you on the side of a mountain. Your goal is to get to the peak of the mountain as quickly as possible. All you can do is feel the mountain where you are standing, and take steps. How would you get to the top of the mountain? Perhaps you would feel to find out what direction feels the most "upward" and take a step in that direction. If you do this repeatedly, you might hope to get the the top of the mountain. (Actually, if your friend promises always to drop you on purely concave mountains, you *will* eventually get to the peak!)

The idea of gradient-based methods of optimization is exactly the same. Suppose you are trying to find the maximum of a function $f(\boldsymbol{x})$. The optimizer maintains a current estimate of the parameter of interest, $\boldsymbol{x}$. At each step, it measures the **gradient** of the function it is trying optimize. This measurement occurs *at* the current location, $\boldsymbol{x}$. Call the gradient $\boldsymbol{g}$. It then takes a step in the direction of the gradient, where the size of the step is controlled by a parameter $\eta$ (eta). The complete step is $\boldsymbol{x} \leftarrow \boldsymbol{x} + \eta \boldsymbol{g}$. This is the basic idea of **gradient ascent**.

The opposite of gradient ascent is **gradient descent**. All of our learning problems will be framed as *minimization* problems (trying to reach the bottom of a ditch, rather than the top of a hill). Therefore, descent is the primary approach you will use. One of the major conditions for gradient ascent being able to find the true, **global minimum**, of its objective function is convexity. Without convexity, all is lost.

The gradient descent algorithm is sketched in Algorithm 6.4. The function takes as arguments the function $\mathcal{F}$ to be minimized, the number of iterations $K$ to run and a sequence of learning rates

$\eta_1, \ldots, \eta_K$. (This is to address the case that you might want to start your mountain climbing taking large steps, but only take small steps when you are close to the peak.)

The only real work you need to do to apply a gradient descent method is be able to compute derivatives. For concreteness, suppose that you choose exponential loss as a loss function and the 2-norm as a regularizer. Then, the regularized objective function is:

$$\mathcal{L}(\boldsymbol{w}, b) = \sum_n \exp\left[-y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right] + \frac{\lambda}{2} ||\boldsymbol{w}||^2 \tag{6.11}$$

The only "strange" thing in this objective is that we have replaced $\lambda$ with $\frac{\lambda}{2}$. The reason for this change is just to make the gradients cleaner. We can first compute derivatives with respect to $b$:

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial}{\partial b} \sum_n \exp\left[-y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right] + \frac{\partial}{\partial b} \frac{\lambda}{2} ||\boldsymbol{w}||^2 \tag{6.12}$$

$$= \sum_n \frac{\partial}{\partial b} \exp\left[-y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right] + 0 \tag{6.13}$$

$$= \sum_n \left(\frac{\partial}{\partial b} - y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right) \exp\left[-y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right] \tag{6.14}$$

$$= -\sum_n y_n \exp\left[-y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right] \tag{6.15}$$

Before proceeding, it is worth thinking about what this says. From a practical perspective, the optimization will operate by updating $b \leftarrow b - \eta \frac{\partial \mathcal{L}}{\partial b}$. Consider positive examples: examples with $y_n = +1$. We would hope for these examples that the current prediction, $\boldsymbol{w} \cdot \boldsymbol{x}_n + b$, is as large as possible. As this value tends toward $\infty$, the term in the exp[] goes to zero. Thus, such points will not contribute to the step. However, if the current prediction is small, then the exp[] term will be positive and non-zero. This means that the bias term $b$ will be *increased*, which is exactly what you would want. Moreover, once all points are very well classified, the derivative goes to zero.

? This considered the case of positive examples. What happens with negative examples?

Now that we have done the easy case, let's do the gradient with respect to $\boldsymbol{w}$.

$$\nabla_{\boldsymbol{w}} \mathcal{L} = \nabla_{\boldsymbol{w}} \sum_n \exp\left[-y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right] + \nabla_{\boldsymbol{w}} \frac{\lambda}{2} ||\boldsymbol{w}||^2 \tag{6.16}$$

$$= \sum_n (\nabla_{\boldsymbol{w}} - y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)) \exp\left[-y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right] + \lambda \boldsymbol{w} \tag{6.17}$$

$$= -\sum_n y_n \boldsymbol{x}_n \exp\left[-y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right] + \lambda \boldsymbol{w} \tag{6.18}$$

Now you can repeat the previous exercise. The update is of the form $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \nabla_{\boldsymbol{w}} \mathcal{L}$. For well classified points (ones that are tend toward $y_n \infty$), the gradient is near zero. For poorly classified points,

the gradient points in the direction $-y_n \boldsymbol{x}_n$, so the update is of the form $\boldsymbol{w} \leftarrow \boldsymbol{w} + c y_n \boldsymbol{x}_n$, where $c$ is some constant. This is just like the perceptron update! Note that $c$ is large for very poorly classified points and small for relatively well classified points.

By looking at the part of the gradient related to the regularizer, the update says: $\boldsymbol{w} \leftarrow \boldsymbol{w} - \lambda \boldsymbol{w} = (1 - \lambda)\boldsymbol{w}$. This has the effect of *shrinking* the weights toward zero. This is exactly what we expect the regulaizer to be doing!

The success of gradient descent hinges on appropriate choices for the step size. Figure 6.7 shows what can happen with gradient descent with poorly chosen step sizes. If the step size is too big, you can accidentally step over the optimum and end up oscillating. If the step size is too small, it will take way too long to get to the optimum. For a well-chosen step size, you can show that gradient descent will approach the optimal value at a fast *rate*. The notion of convergence here is that the *objective value* converges to the true minimum.

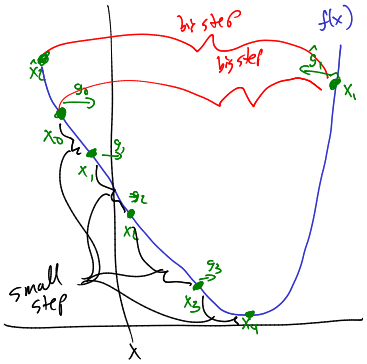



Figure 6.7: good and bad step sizes

**Theorem 7** (Gradient Descent Convergence). *Under suitable conditions*[1]*, for an appropriately chosen constant step size (i.e., $\eta_1 = \eta_2, \cdots = \eta$), the* **convergence rate** *of gradient descent is $\mathcal{O}(1/k)$. More specifically, letting $\boldsymbol{z}^*$ be the global minimum of $\mathcal{F}$, we have: $\mathcal{F}(\boldsymbol{z}^{(k)}) - \mathcal{F}(\boldsymbol{z}^*) \leq \frac{2||\boldsymbol{z}^{(0)} - \boldsymbol{z}^*||^2}{\eta k)}$.*

[1] Specifically the function to be optimized needs to be **strongly convex**. This is true for all our problems, provided $\lambda > 0$. For $\lambda = 0$ the rate could be as bad as $\mathcal{O}(1/\sqrt{k})$.

The proof of this theorem is a bit complicated because it makes heavy use of some linear algebra. The key is to set the learning rate to $1/L$, where $L$ is the maximum **curvature** of the function that is being optimized. The curvature is simply the "size" of the second derivative. Functions with high curvature have gradients that change quickly, which means that you need to take small steps to avoid overstepping the optimum.

? A naive reading of this theorem seems to say that you should choose huge values of $\eta$. It should be obvious that this cannot be right. What is missing?

This convergence result suggests a simple approach to deciding when to stop optimizing: wait until the objective function stops changing by much. An alternative is to wait until the *parameters* stop changing by much. A final example is to do what you did for perceptron: early stopping. Every iteration, you can check the performance of the current model on some held-out data, and stop optimizing when performance plateaus.

## 6.5 From Gradients to Subgradients

As a good exercise, you should try deriving gradient descent update rules for the different loss functions and different regularizers you've learned about. However, if you do this, you might notice that *hinge loss* and the 1-norm regularizer are not differentiable everywhere! In

particular, the 1-norm is not differentiable around $w_d = 0$, and the hinge loss is not differentiable around $y\hat{y} = 1$.

The solution to this is to use **subgradient** optimization. One way to think about subgradients is just to not think about it: you essentially need to just ignore the fact that you forgot that your function wasn't differentiable, and just try to apply gradient descent anyway.

To be more concrete, consider the hinge function $f(z) = \max\{0, 1 - z\}$. This function is differentiable for $z > 1$ and differentiable for $z < 1$, but not differentiable at $z = 1$. You can derive this using differentiation by parts:

$$\frac{\partial}{\partial z} f(z) = \frac{\partial}{\partial z} \begin{cases} 0 & \text{if } z > 1 \\ 1 - z & \text{if } z < 1 \end{cases} \tag{6.19}$$

$$= \begin{cases} \frac{\partial}{\partial z} 0 & \text{if } z > 1 \\ \frac{\partial}{\partial z}(1 - z) & \text{if } z < 1 \end{cases} \tag{6.20}$$

$$= \begin{cases} 0 & \text{if } z \geq 1 \\ -1 & \text{if } z < 1 \end{cases} \tag{6.21}$$

Thus, the derivative is zero for $z < 1$ and $-1$ for $z > 1$, matching intuition from the Figure. At the non-differentiable point, $z = 1$, we can use a **subderivative**: a generalization of derivatives to non-differentiable functions. Intuitively, you can think of the derivative of $f$ at $z$ as the tangent line. Namely, it is the line that touches $f$ at $z$ that is always below $f$ (for convex functions). The subderivative, denoted $\partial f$, is the *set* of all such lines. At differentiable positions, this set consists just of the actual derivative. At non-differentiable positions, this contains all slopes that define lines that always lie under the function and make contact at the operating point. This is shown pictorally in Figure 6.8, where example subderivatives are shown for the hinge loss function. In the particular case of hinge loss, any value between 0 and $-1$ is a valid subderivative at $z = 0$. In fact, the subderivative is always a closed set of the form $[a, b]$, where $a$ and $b$ can be derived by looking at limits from the left and right.

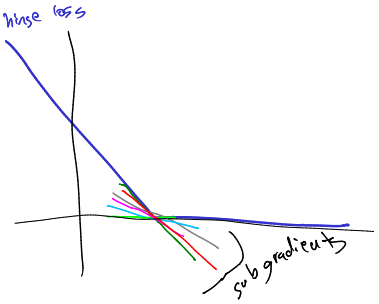



Figure 6.8: hinge loss with sub

This gives you a way of computing derivative-like things for non-differentiable functions. Take hinge loss as an example. For a given example $n$, the subgradient of hinge loss can be computed as:

$$\partial_{\boldsymbol{w}} \max\{0, 1 - y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\} \tag{6.22}$$

$$= \partial_{\boldsymbol{w}} \begin{cases} 0 & \text{if } y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) > 1 \\ y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) & \text{otherwise} \end{cases} \tag{6.23}$$

$$= \begin{cases} \partial_{\boldsymbol{w}} 0 & \text{if } y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) > 1 \\ \partial_{\boldsymbol{w}} y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) & \text{otherwise} \end{cases} \tag{6.24}$$

$$= \begin{cases} \boldsymbol{0} & \text{if } y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) > 1 \\ y_n \boldsymbol{x}_n & \text{otherwise} \end{cases} \tag{6.25}$$

**Algorithm 23** HINGEREGULARIZEDGD($\mathbf{D}$, $\lambda$, *MaxIter*)

```
w ← ⟨0, 0, ... 0⟩   ,   b ← 0                    // initialize weights and bias
for iter = 1 ... MaxIter do
    g ← ⟨0, 0, ... 0⟩   ,   g ← 0                // initialize gradient of weights and bias
    for all (x,y) ∈ D do
        if y(w · x + b) ≤ 1 then
            g ← g + y x                          // update weight gradient
            g ← g + y                            // update bias derivative
        end if
    end for
    g ← g − λw                                   // add in regularization term
    w ← w + ηg                                   // update weights
    b ← b + ηg                                   // update bias
end for
return w, b
```

**MATH REVIEW | MATRIX MULTIPLICATION AND INVERSION**

...

Figure 6.9:

If you plug this subgradient form into Algorithm 6.4, you obtain Algorithm 6.5. This is the **subgradient descent** for regularized hinge loss (with a 2-norm regularizer).

## 6.6 Closed-form Optimization for Squared Loss

Although gradient descent is a good, generic optimization algorithm, there are cases when you can do better. An example is the case of a 2-norm regularizer and squared error loss function. For this, you can actually obtain a *closed form* solution for the optimal weights. However, to obtain this, you need to rewrite the optimization problem in terms of matrix operations. For simplicity, we will only consider the *unbiased* version, but the extension is Exercise **??**. This is precisely the **linear regression** setting.

You can think of the training data as a large matrix $\mathbf{X}$ of size $N \times D$, where $X_{n,d}$ is the value of the $d$th feature on the $n$th example. You can think of the labels as a column ("tall") vector $\boldsymbol{Y}$ of dimension $N$. Finally, you can think of the weights as a column vector $\boldsymbol{w}$ of size $D$. Thus, the matrix-vector product $\boldsymbol{a} = \mathbf{X}\boldsymbol{w}$ has dimension $N$. In particular:

$$a_n = [\mathbf{X}\boldsymbol{w}]_n = \sum_d \mathbf{X}_{n,d} w_d \tag{6.26}$$

This means, in particular, that $\boldsymbol{a}$ is actually the predictions of the model. Instead of calling this $\boldsymbol{a}$, we will call it $\hat{\boldsymbol{Y}}$. The squared error

says that we should minimize $\frac{1}{2}\sum_n (\hat{Y}_n - Y_n)^2$, which can be written in vector form as a minimization of $\frac{1}{2}\left|\left|\hat{\boldsymbol{Y}} - \boldsymbol{Y}\right|\right|^2$.

This can be expanded visually as:

$$\underbrace{\begin{bmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,D} \\ x_{2,1} & x_{2,2} & \dots & x_{2,D} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N,1} & x_{N,2} & \dots & x_{N,D} \end{bmatrix}}_{\mathbf{X}} \underbrace{\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_D \end{bmatrix}}_{\boldsymbol{w}} = \underbrace{\begin{bmatrix} \sum_d x_{1,d} w_d \\ \sum_d x_{2,d} w_d \\ \vdots \\ \sum_d x_{N,d} w_d \end{bmatrix}}_{\hat{\boldsymbol{Y}}} \approx \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}}_{\boldsymbol{Y}} \tag{6.27}$$

So, compactly, our optimization problem can be written as:

$$\min_{\boldsymbol{w}} \quad \mathcal{L}(\boldsymbol{w}) = \frac{1}{2}\left|\left|\mathbf{X}\boldsymbol{w} - \boldsymbol{Y}\right|\right|^2 + \frac{\lambda}{2}\left|\left|\boldsymbol{w}\right|\right|^2 \tag{6.28}$$

If you recall from calculus, you can minimize a function by setting its derivative to zero. We start with the weights $\boldsymbol{w}$ and take gradients:

$$\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}) = \mathbf{X}^\top (\mathbf{X}\boldsymbol{w} - \boldsymbol{Y}) + \lambda \boldsymbol{w} \tag{6.29}$$

$$= \mathbf{X}^\top \mathbf{X}\boldsymbol{w} - \mathbf{X}^\top \boldsymbol{Y} + \lambda \boldsymbol{w} \tag{6.30}$$

$$= \left(\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}\right) \boldsymbol{w} - \mathbf{X}^\top \boldsymbol{Y} \tag{6.31}$$

We can equate this to zero and solve, yielding:

$$\left(\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}\right) \boldsymbol{w} - \mathbf{X}^\top \boldsymbol{Y} = 0 \tag{6.32}$$

$$\iff \left(\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_D\right) \boldsymbol{w} = \mathbf{X}^\top \boldsymbol{Y} \tag{6.33}$$

$$\iff \boldsymbol{w} = \left(\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_D\right)^{-1} \mathbf{X}^\top \boldsymbol{Y} \tag{6.34}$$

Thus, the *optimal* solution of the weights can be computed by a few matrix multiplications and a matrix inversion. As a sanity check, you can make sure that the dimensions match. The matrix $\mathbf{X}^\top \mathbf{X}$ has dimension $D \times D$, and therefore so does the inverse term. The inverse is $D \times D$ and $\mathbf{X}^\top$ is $D \times N$, so that product is $D \times N$. Multiplying through by the $N \times 1$ vector $\boldsymbol{Y}$ yields a $D \times 1$ vector, which is precisely what we want for the weights.

Note that this gives an *exact solution*, modulo numerical innacuracies with computing matrix inverses. In contrast, gradient descent will give you progressively better solutions and will "eventually" converge to the optimum at a rate of $1/k$. This means that if you want an answer that's within an accuracy of $\epsilon = 10^{-4}$, you will need something on the order of one thousand steps.

The question is whether getting this exact solution is always more efficient. To run gradient descent for one step will take $\mathcal{O}(ND)$ time, with a relatively small constant. You will have to run $K$ iterations,

? Verify that the squared error can actually be written as this vector norm.

? For those who are keen on linear algebra, you might be worried that the matrix you must invert might not be invertible. Is this actually a problem?

yielding an overall runtime of $\mathcal{O}(KND)$. On the other hand, the closed form solution requires constructing $\mathbf{X}^\top\mathbf{X}$, which takes $\mathcal{O}(D^2N)$ time. The inversion take $\mathcal{O}(D^3)$ time using standard matrix inversion routines. The final multiplications take $\mathcal{O}(ND)$ time. Thus, the overall runtime is on the order $\mathcal{O}(D^3 + D^2N)$. In most standard cases (though this is becoming less true over time), $N > D$, so this is dominated by $\mathcal{O}(D^2N)$.

Thus, the overall question is whether you will need to run more than $D$-many iterations of gradient descent. If so, then the matrix inversion will be (roughly) faster. Otherwise, gradient descent will be (roughly) faster. For low- and medium-dimensional problems (say, $D \leq 100$), it is probably faster to do the closed form solution via matrix inversion. For high dimensional problems ($D \geq 10,000$), it is probably faster to do gradient descent. For things in the middle, it's hard to say for sure.

## 6.7 Support Vector Machines

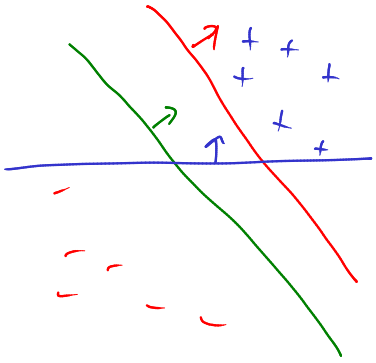


Figure 6.10: picture of data points with three hyperplanes, RGB with G the best

At the beginning of this chapter, you may have looked at the convex surrogate loss functions and asked yourself: where did these come from?! They are all derived from different underlying principles, which essentially correspond to different inductive biases.

Let's start by thinking back to the original goal of linear classifiers: to find a hyperplane that separates the positive training examples from the negative ones. Figure 6.10 shows some data and three potential hyperplanes: red, green and blue. Which one do you like best?

Most likely you chose the green hyperplane. And most likely you chose it because it was furthest away from the closest training points. In other words, it had a large **margin**. The desire for hyperplanes with large margins is a perfect example of an inductive bias. The data does not tell us which of the three hyperplanes is best: we have to choose one using some other source of information.

Following this line of thinking leads us to the **support vector machine** (SVM). This is simply a way of setting up an optimization problem that attempts to find a separating hyperplane with as large a margin as possible. It is written as a **constrained optimization problem**:

$$\begin{aligned} \min_{\boldsymbol{w},b} \quad & \frac{1}{\gamma(\boldsymbol{w},b)} && (6.35)\\ \text{subj. to} \quad & y_n\left(\boldsymbol{w}\cdot\boldsymbol{x}_n + b\right) \geq 1 && (\forall n) \end{aligned}$$

In this optimization, you are trying to find parameters that maximize the margin, denoted $\gamma$, (i.e., minimize the reciprocal of the margin)

subject to the constraint that *all* training examples are correctly classified.

The "odd" thing about this optimization problem is that we require the classification of each point to be greater than *one* rather than simply greater than *zero*. However, the problem doesn't fundamentally change if you replace the "1" with any other positive constant (see Exercise **??**). As shown in Figure 6.11, the constant one can be interpreted visually as ensuring that there is a non-trivial margin between the positive points and negative points.

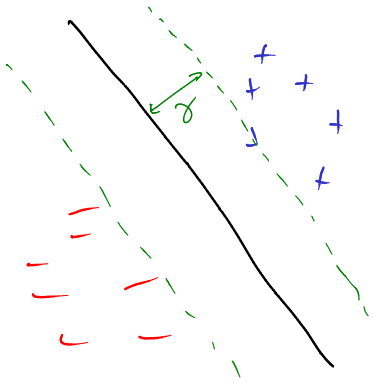



Figure 6.11: hyperplane with margins on sides

The difficulty with the optimization problem in Eq (**??**) is what happens with data that is not linearly separable. In that case, there *is no* set of parameters $\boldsymbol{w}, b$ that can simultaneously satisfy all the constraints. In optimization terms, you would say that the **feasible region** is *empty*. (The feasible region is simply the set of all parameters that satify the constraints.) For this reason, this is refered to as the **hard-margin SVM**, because enforcing the margin is a hard constraint. The question is: how to modify this optimization problem so that it can handle inseparable data.

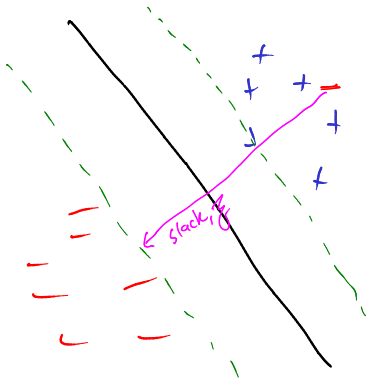



Figure 6.12: one bad point with slack

The key idea is the use of **slack parameters**. The intuition behind slack parameters is the following. Suppose we find a set of parameters $\boldsymbol{w}, b$ that do a really good job on 9999 data points. The points are perfectly classifed and you achieve a large margin. But there's one pesky data point left that cannot be put on the proper side of the margin: perhaps it is noisy. (See Figure 6.12.) You want to be able to pretend that you can "move" that point across the hyperplane on to the proper side. You will have to pay a little bit to do so, but as long as you aren't moving a *lot* of points around, it should be a good idea to do this. In this picture, the amount that you move the point is denoted $\xi$ (xi).

By introducing one slack parameter for each training example, and penalizing yourself for having to use slack, you can create an objective function like the following, **soft-margin SVM**:

$$\min_{\boldsymbol{w},b,\boldsymbol{\xi}} \quad \underbrace{\frac{1}{\gamma(\boldsymbol{w},b)}}_{\text{large margin}} + \underbrace{C\sum_n \xi_n}_{\text{small slack}} \tag{6.36}$$

$$\text{subj. to} \quad y_n\left(\boldsymbol{w}\cdot\boldsymbol{x}_n + b\right) \geq 1 - \xi_n \qquad (\forall n)$$

$$\xi_n \geq 0 \qquad (\forall n)$$

The goal of this objective function is to ensure that all points are correctly classified (the first constraint). But if a point $n$ cannot be correctly classified, then you can set the slack $\xi_n$ to something greater than zero to "move" it in the correct direction. However, for all non-zero slacks, you have to pay in the objective function proportional to the amount of slack. The hyperparameter $C > 0$ controls overfitting

versus underfitting. The second constraint simply says that you must not have negative slack.

> ? What values of $C$ will lead to overfitting? What values will lead to underfitting?

One major advantage of the soft-margin SVM over the original hard-margin SVM is that the feasible region is *never empty*. That is, there is always going to be some solution, regardless of whether your training data is linearly separable or not.

It's one thing to write down an optimization problem. It's another thing to try to solve it. There are a very large number of ways to optimize SVMs, essentially because they are such a popular learning model. Here, we will talk just about one, very simple way. More complex methods will be discussed later in this book once you have a bit more background.

> ? Suppose I give you a data set. Without even looking at the data, construct for me a feasible solution to the soft-margin SVM. What is the value of the objective for this solution?

To make progress, you need to be able to measure the size of the margin. Suppose someone gives you parameters $\boldsymbol{w}, b$ that optimize the hard-margin SVM. We wish to measure the size of the margin. The first observation is that the hyperplane will lie *exactly* halfway between the nearest positive point and nearest negative point. If not, the margin could be made bigger by simply sliding it one way or the other by adjusting the bias $b$.

By this observation, there is some positive example that that lies exactly 1 unit from the hyperplane. Call it $\boldsymbol{x}^+$, so that $\boldsymbol{w} \cdot \boldsymbol{x}^+ + b = 1$. Similarly, there is some negative example, $\boldsymbol{x}^-$, that lies exactly on the other side of the margin: for which $\boldsymbol{w} \cdot \boldsymbol{x}^- + b = -1$. These two points, $\boldsymbol{x}^+$ and $\boldsymbol{x}^-$ give us a way to measure the size of the margin. As shown in Figure 6.11, we can measure the size of the margin by looking at the difference between the lengths of projections of $\boldsymbol{x}^+$ and $\boldsymbol{x}^-$ onto the hyperplane. Since projection requires a normalized vector, we can measure the distances as:

$$d^+ = \frac{1}{||\boldsymbol{w}||} \boldsymbol{w} \cdot \boldsymbol{x}^+ + b - 1 \tag{6.37}$$

$$d^- = -\frac{1}{||\boldsymbol{w}||} \boldsymbol{w} \cdot \boldsymbol{x}^- - b + 1 \tag{6.38}$$

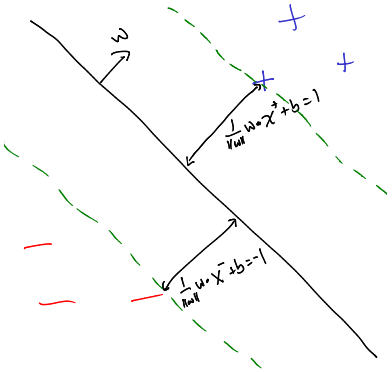


Figure 6.13: copy of figure from p5 of cs544 svm tutorial

We can then compute the margin by algebra:

$$\gamma = \frac{1}{2}\left[d^+ - d^-\right] \tag{6.39}$$

$$= \frac{1}{2}\left[\frac{1}{||\boldsymbol{w}||} \boldsymbol{w} \cdot \boldsymbol{x}^+ + b - 1 - \frac{1}{||\boldsymbol{w}||} \boldsymbol{w} \cdot \boldsymbol{x}^- - b + 1\right] \tag{6.40}$$

$$= \frac{1}{2}\left[\frac{1}{||\boldsymbol{w}||} \boldsymbol{w} \cdot \boldsymbol{x}^+ - \frac{1}{||\boldsymbol{w}||} \boldsymbol{w} \cdot \boldsymbol{x}^-\right] \tag{6.41}$$

$$= \frac{1}{2}\left[\frac{1}{||\boldsymbol{w}||}(+1) - \frac{1}{||\boldsymbol{w}||}(-1)\right] \tag{6.42}$$

$$= \frac{1}{||\boldsymbol{w}||} \tag{6.43}$$

This is a remarkable conclusion: the size of the margin is inversely proportional to the norm of the weight vector. Thus, **maximizing the margin is equivalent to minimizing $||\boldsymbol{w}||$!** This serves as an additional justification of the 2-norm regularizer: having small weights means having large margins!

However, our goal wasn't to justify the regularizer: it was to understand hinge loss. So let us go back to the soft-margin SVM and plug in our new knowledge about margins:

$$\begin{aligned} \min_{\boldsymbol{w},b,\boldsymbol{\xi}} \quad & \underbrace{\frac{1}{2}||\boldsymbol{w}||^2}_{\text{large margin}} + \underbrace{C\sum_n \xi_n}_{\text{small slack}} & (6.44)\\ \text{subj. to} \quad & y_n\left(\boldsymbol{w}\cdot\boldsymbol{x}_n + b\right) \geq 1 - \xi_n & (\forall n)\\ & \xi_n \geq 0 & (\forall n) \end{aligned}$$

Now, let's play a thought experiment. Suppose someone handed you a solution to this optimization problem that consisted of weights ($\boldsymbol{w}$) and a bias ($b$), but they forgot to give you the slacks. Could you recover the slacks from the information you have?

In fact, the answer is yes! For simplicity, let's consider positive examples. Suppose that you look at some positive example $\boldsymbol{x}_n$. You need to figure out what the slack, $\xi_n$, would have been. There are two cases. Either $\boldsymbol{w}\cdot\boldsymbol{x}_n + b$ is at least 1 or it is not. If it's large enough, then you want to set $\xi_n = 0$. Why? It cannot be less than zero by the second constraint. Moreover, if you set it greater than zero, you will "pay" unnecessarily in the objective. So in this case, $\xi_n = 0$. Next, suppose that $\boldsymbol{w}\cdot\boldsymbol{x}_n + b = 0.2$, so it is not big enough. In order to satisfy the first constraint, you'll need to set $\xi_n \geq 0.8$. But because of the objective, you'll not want to set it any larger than necessary, so you'll set $\xi_n = 0.8$ exactly.

Following this argument through for both positive and negative points, if someone gives you solutions for $\boldsymbol{w}, b$, you can automatically compute the optimal $\xi$ variables as:

$$\xi_n = \begin{cases} 0 & \text{if } y_n(\boldsymbol{w}\cdot\boldsymbol{x}_n + b) \geq 1 \\ 1 - y_n(\boldsymbol{w}\cdot\boldsymbol{x}_n + b) & \text{otherwise} \end{cases} \tag{6.45}$$

In other words, the optimal value for a slack variable is *exactly* the hinge loss on the corresponding example! Thus, we can write the SVM objective as an *unconstrained* optimization problem:

$$\min_{\boldsymbol{w},b} \quad \underbrace{\frac{1}{2}||\boldsymbol{w}||^2}_{\text{large margin}} + \underbrace{C\sum_n \ell^{(\text{hin})}(y_n, \boldsymbol{w}\cdot\boldsymbol{x}_n + b)}_{\text{small slack}} \tag{6.46}$$

Multiplying this objective through by $\lambda/C$, we obtain exactly the regularized objective from Eq (6.8) with hinge loss as the loss function and the 2-norm as the regularizer!

TODO: justify in term of one dimensional projections!

## 6.8 Exercises

**Exercise 6.1. TODO...**

# 7 | Probabilistic Modeling

–

**Learning Objectives:**
- Define the generative story for a naive Bayes classifier.
- Derive relative frequency as the solution to a constrained optimization problem.
- Compare and contrast generative, conditional and discriminative learning.
- Explain when generative models are likely to fail.
- Derive logistic loss with an $\ell_2$ regularizer from a probabilistic perspective.

Dependencies:

Many of the models and algorithms you have learned about thus far are relatively *disconnected*. There is an alternative view of machine learning that unites and generalizes much of what you have already learned. This is the **probabilistic modeling** framework, in which you will explicitly think of learning as a problem of **statistical inference**.

In this chapter, you will learn about two flavors of probabilistic models: generative and conditional. You will see that many of the approaches (both supervised and unsupervised) we have seen already can be cast as probabilistic models. Through this new view, you will be able to develop learning algorithms that have inductive biases closer to what you, as a designer, believe. Moreover, the two chapters that follow will make heavy use of the probabilistic modeling approach to open doors to other learning problems.

## 7.1 Classification by Density Estimation

Our underlying assumption for the majority of this book is that learning problems are characterized by some unknown probability distribution $\mathcal{D}$ over input/output pairs $(x, y) \in \mathcal{X} \times \mathcal{Y}$. Suppose that someone *told* you what $\mathcal{D}$ was. In particular, they gave you a Python function computeD that took two inputs, $x$ and $y$, and returned the probability of that $x, y$ pair under $\mathcal{D}$. If you had access to such a function, classification becomes simple. We can define the **Bayes optimal classifier** as the classifier that, for any test input $\hat{x}$, simply returns the $\hat{y}$ that maximizes computeD$(\hat{x}, \hat{y})$, or, more formally:

$$f^{(\mathrm{BO})}(\hat{x}) = \arg\max_{\hat{y} \in \mathcal{Y}} \mathcal{D}(\hat{x}, \hat{y}) \tag{7.1}$$

This classifier is optimal in one specific sense: of *all possible* classifiers, it achieves the smallest zero/one error.

**Theorem 8** (Bayes Optimal Classifier)**.** *The Bayes Optimal Classifier $f^{(BO)}$ achieves minimal zero/one error of any deterministic classifier.*

**MATH REVIEW | RULES OF PROBABILITY**

chain rule, marginalization and Bayes' rule

Figure 7.1:

This theorem assumes that you are comparing against *deterministic* classifiers. You can actually prove a stronger result that $f^{(\text{BO})}$ is optimal for randomized classifiers as well, but the proof is a bit messier. However, the intuition is the same: for a given $x$, $f^{(\text{BO})}$ chooses the label with highest probability, thus *minimizing* the probability that it makes an error.

*Proof of Theorem 8.* Consider some other classifier $g$ that claims to be better than $f$. Then, there must be some $x$ on which $g(x) \neq f(x)$. Fix such an $x$. Now, the probability that $f$ makes an error on this particular $x$ is $1 - \mathcal{D}(x, f^{(\text{BO})}(x))$ and the probability that $g$ makes an error on this $x$ is $1 - \mathcal{D}(x, g(x))$. But $f^{(\text{BO})}$ was chosen in such a way to *maximize* $\mathcal{D}(x, f^{(\text{BO})}(x))$, so this *must* be greater than $\mathcal{D}(x, g(x))$. Thus, the probability that $f$ errs on this particular $x$ is smaller than the probability that $g$ errs on it. This applies to any $x$ for which $f(x) \neq g(x)$ and therefore $f$ achieves smaller zero/one error than any $g$. □

The **Bayes error rate** (or **Bayes optimal error rate**) is the error rate of the Bayes optimal classifier. It is the best error rate you can ever hope to achieve on this classification problem (under zero/one loss).

The take-home message is that if someone gave you access to the data distribution, forming an *optimal* classifier would be trivial. Unfortunately, no one gave you this distribution, but this analysis suggests that good way to build a classifier is to try to *estimate* $\mathcal{D}$. In other words, you try to learn a distribution $\hat{\mathcal{D}}$, which you hope to very similar to $\mathcal{D}$, and then use this distribution for classification. Just as in the preceding chapters, you can try to form your estimate of $\mathcal{D}$ based on a finite training set.

The most direct way that you can attempt to construct such a probability distribution is to select a *family* of parametric distributions. For instance, a Gaussian (or Normal) distribution is parametric: it's parameters are its mean and covariance. The job of learning is then to infer which parameters are "best" as far as the observed training data is concerned, as well as whatever inductive bias you bring. A key assumption that you will need to make is that the training data you have access to is drawn **independently** from $\mathcal{D}$. In particular, as you draw examples $(x_1, y_1) \sim \mathcal{D}$ then $(x_2, y_2) \sim \mathcal{D}$ and so on, the $n$th draw $(x_n, y_n)$ is drawn from $\mathcal{D}$ and *does not* otherwise depend

on the previous $n-1$ samples. This assumption is usually false, but is also usually sufficiently close to being true to be useful. Together with the assumption that all the training data is drawn from the same distribution $\mathcal{D}$ leads to the **i.i.d. assumption** or **independently and identically distributed** assumption. This is a key assumption in almost all of machine learning.

## 7.2 Statistical Estimation

Suppose you need to model a coin that is possibly biased (you can think of this as modeling the *label* in a binary classification problem), and that you observe data `HHTH` (where `H` means a flip came up heads and `T` means it came up tails). You can assume that all the flips came from the same coin, and that each flip was independent (hence, the data was i.i.d.). Further, you may choose to believe that the coin has a fixed probability $\beta$ of coming up heads (and hence $1-\beta$ of coming up tails). Thus, the parameter of your model is simply the scalar $\beta$.

? Describe a case in which at least one of the assumptions we are making about the coin flip is false.

The most basic computation you might perform is **maximum likelihood estimation**: namely, select the paramter $\beta$ the maximizes the probability of the data under that parameter. In order to do so, you need to compute the probability of the data:

$$p_\beta(D) = p_\beta(\texttt{HHTH}) \qquad \text{definition of } D \qquad (7.2)$$
$$= p_\beta(\texttt{H})p_\beta(\texttt{H})p_\beta(\texttt{T})p_\beta(\texttt{H}) \qquad \text{data is independent} \qquad (7.3)$$
$$= \beta\beta(1-\beta)\beta \qquad (7.4)$$
$$= \beta^3(1-\beta) \qquad (7.5)$$
$$= \beta^3 - \beta^4 \qquad (7.6)$$

Thus, if you want the parameter $\beta$ that maximizes the probability of the data, you can take the derivative of $\beta^3 - \beta^4$ with respect to $\beta$, set it equal to zero and solve for $\beta$:

$$\frac{\partial}{\partial\beta}\left[\beta^3 - \beta^4\right] = 3\beta^2 - 4\beta^3 \qquad (7.7)$$
$$4\beta^3 = 3\beta^2 \qquad (7.8)$$
$$\iff 4\beta = 3 \qquad (7.9)$$
$$\iff \beta = \frac{3}{4} \qquad (7.10)$$

Thus, the maximum likelihood $\beta$ is 0.75, which is probably what you would have selected by intuition. You can solve this problem more generally as follows. If you have $H$-many heads and $T$-many tails, the probability of your data sequence is $\beta^H(1-\beta)^T$. You can try to take the derivative of this with respect to $\beta$ and follow the same recipe, but all of the products make things difficult. A more

friendly solution is to work with the **log likelihood** or **log probability** instead. The log likelihood of this data sequence is $H \log \beta + T \log(1-\beta)$. Differentiating with respect to $\beta$, you get $H/\beta - T/(1-\beta)$. To solve, you obtain $H/\beta = T/(1-\beta)$ so $H(1-\beta) = T\beta$. Thus $H - H\beta = T\beta$ and so $H = (H+T)\beta$, finally yeilding that $\beta = H/(H+T)$ or, simply, the fraction of observed data that came up heads. In this case, the maximum likelihood estimate is nothing but the relative frequency of observing heads!

? How do you know that the solution of $\beta = H/(H+T)$ is actually a maximum?

Now, suppose that instead of flipping a coin, you're rolling a $K$-sided die (for instance, to pick the label for a multiclass classification problem). You might model this by saying that there are parameters $\theta_1, \theta_2, \ldots, \theta_K$ specifying, respectively, the probabilities that any given side comes up on a role. Since these are themselves probabilities, each $\theta_k$ should be at least zero, and the sum of the $\theta_k$s should be one. Given a data set that consists of $x_1$ rolls of 1, $x_2$ rolls of 2 and so on, the probability of this data is $\prod_k \theta_k^{x_k}$, yielding a log probability of $\sum_k x_k \log \theta_k$. If you pick some particular parameter, say $\theta_3$, the derivative of this with respect to $\theta_3$ is $x_3/\theta_3$, which you want to equate to zero. This leads to... $\theta_3 \to \infty$.

This is obviously "wrong." From the mathematical formulation, it's correct: in fact, setting all of the $\theta_k$s to $\infty$ *does* maximize $\prod_k \theta_k^{x_k}$ for any (non-negative) $x_k$s. The problem is that you need to constrain the $\theta$s to sum to one. In particular, you have a constraint that $\sum_k \theta_k = 1$ that you forgot to enforce. A convenient way to enforce such constraints is through the technique of **Lagrange multipliers**. To make this problem consistent with standard minimization problems, it is convenient to minimize negative log probabilities, instead of maximizing log probabilities. Thus, the *constrainted* optimization problem is:

$$\begin{aligned} \min_{\boldsymbol{\theta}} \quad & -\sum_k x_k \log \theta_k \\ \text{subj. to} \quad & \sum_k \theta_k - 1 = 0 \end{aligned} \tag{7.11}$$

The Lagrange multiplier approach involves adding a new variable $\lambda$ to the problem (called the **Lagrange variable**) corresponding to the constraint, and to use that to move the constraint into the objective. The result, in this case, is:

$$\max_{\lambda} \min_{\boldsymbol{\theta}} \quad -\sum_k x_k \log \theta_k - \lambda \left( \sum_k \theta_k - 1 \right) \tag{7.12}$$

Turning a constrained optimization problem into it's corresponding **Lagrangian** is straightforward. The mystical aspect is why it works. In this case, the idea is as follows. Think of $\lambda$ as an adversary: $\lambda$

is trying to maximize this function (you're trying to minimize it). If you pick some parameters $\boldsymbol{\theta}$ that actually satisfy the constraint, then the green term in Eq (**??**) goes to zero, and therefore $\lambda$ does not matter: the adversary cannot do anything. On the other hand, if the constraint is even *slightly* unsatisfied, then $\lambda$ can tend toward $+\infty$ or $-\infty$ to blow up the objective. So, in order to have a non-infinite objective value, the optimizer *must* find values of $\boldsymbol{\theta}$ that satisfy the constraint.

If we solve the *inner* optimization of Eq (**??**) by differentiating with respect to $\theta_1$, we get $x_1/\theta_1 = \lambda$, yielding $\theta_1 = x_1/\lambda$. In general, the solution is $\theta_k = x_k/\lambda$. Remembering that the goal of $\lambda$ is to enforce the sums-to-one constraint, we can set $\lambda = \sum_k x_k$ and verify that this is a solution. Thus, our optimal $\theta_k = x_k / \sum_k x_k$, which again completely corresponds to intuition.

## 7.3 Naive Bayes Models

Now, consider the binary classification problem. You are looking for a *parameterized* probability distribution that can describe the training data you have. To be concrete, your task might be to predict whether a movie review is positive or negative (label) based on what words (features) appear in that review. Thus, the probability for a *single* data point can be written as:

$$p_{\boldsymbol{\theta}}((y, \boldsymbol{x})) = p_{\boldsymbol{\theta}}(y, x_1, x_2, \ldots, x_D) \tag{7.13}$$

The challenge in working with a probability distribution like Eq (7.13) is that it's a distribution over a *lot* of variables. You can try to simplify it by applying the **chain rule** of probabilities:

$$\begin{aligned} p_{\boldsymbol{\theta}}(x_1, x_2, \ldots, x_D, y) &= p_{\boldsymbol{\theta}}(y) p_{\boldsymbol{\theta}}(x_1 \mid y) p_{\boldsymbol{\theta}}(x_2 \mid y, x_1) p_{\boldsymbol{\theta}}(x_3 \mid y, x_1, x_2) \\ &\quad \cdots p_{\boldsymbol{\theta}}(x_D \mid y, x_1, x_2, \ldots, x_{D-1}) && (7.14) \\ &= p_{\boldsymbol{\theta}}(y) \prod_d p_{\boldsymbol{\theta}}(x_d \mid y, x_1, \ldots, x_{d-1}) && (7.15) \end{aligned}$$

At this point, this equality is *exact* for any probability distribution. However, it might be difficult to craft a probability distribution for the 10000th feature, given the previous 9999. Even if you could, it might be difficult to accurately estimate it. At this point, you can make assumptions. A classic assumption, called the **naive Bayes assumption**, is that *the features are independent, conditioned on the label.* In the movie review example, this is saying that *once you know that it's a positive review*, the probability that the word "excellent" appears is independent of whether "amazing" also appeared. (Note that this does *not* imply that these words are independent when you

don't know the label—they most certainly are not.) Formally this assumption states that:

$$\textbf{Assumption:} \quad p(x_d \mid y, x_{d'}) = p(x_d \mid y) \quad , \quad \forall d \neq d' \tag{7.16}$$

Under this assumption, you can simplify Eq (7.15) to:

$$p_{\theta}((y, \boldsymbol{x})) = p_{\theta}(y) \prod_d p_{\theta}(x_d \mid y) \qquad \text{naive Bayes assumption} \tag{7.17}$$

At this point, you can start parameterizing $p$. Suppose, for now, that your labels are binary *and* your features are also binary. In this case, you could model the label as a biased coin, with probability of heads (eg., positive review) given by $\theta_0$. Then, for each label, you can imagine having one (biased) coin for each feature. So if there are $D$-many features, you'll have $1 + 2D$ total coins: one for the label (call it $\theta_0$) and one for each label/feature combination (call these $\boldsymbol{\theta}_{+1}$ and as $\boldsymbol{\theta}_{-1}$). In the movie review example, we might expect $\theta_0 \approx 0.4$ (forty percent of movie reviews are positive) and also that $\theta_{+1}$ might give high probability to words like "excellent" and "amazing" and "good" and $\boldsymbol{\theta}_{-1}$ might give high probability to words like "terrible" and "boring" and "hate". You can rewrite the probability of a single example as follows, eventually leading to the log probability of the entire data set:

$$p_{\theta}((y, \boldsymbol{x})) = p_{\theta}(y) \prod_d p_{\theta}(x_d \mid y) \qquad \text{naive Bayes assumption} \tag{7.18}$$

$$= \left(\theta_0^{[y=+1]} (1 - \theta_0)^{[y=-1]}\right) \prod_d \theta_{(y),d}^{[x_d=1]} (1 - \theta_{(y),d})^{[x_d=0]} \qquad \text{model assumptions} \tag{7.19}$$

Solving for $\theta_0$ is identical to solving for the biased coin case from before: it is just the relative frequency of positive labels in your data (because $\theta_0$ doesn't depend on $\boldsymbol{x}$ at all). For the other parameters, you can repeat the same exercise as before for each of the $2D$ coins independently. This yields:

$$\hat{\theta}_0 = \frac{1}{N} \sum_n [y_n = +1] \tag{7.20}$$

$$\hat{\theta}_{(+1),d} = \frac{\sum_n [y_n = +1 \wedge x_{n,d} = 1]}{\sum_n [y_n = +1]} \tag{7.21}$$

$$\hat{\theta}_{(-1),d} = \frac{\sum_n [y_n = -1 \wedge x_{n,d} = 1]}{\sum_n [y_n = -1]} \tag{7.22}$$

In the case that the features are *not* binary, you need to choose a different model for $p(x_d \mid y)$. The model we chose here is the **Bernouilli distribution**, which is effectively a distribution over independent

**MATH REVIEW | COMMON PROBABILITY DISTRIBUTIONS**

remove people about discrete, bernoulli, binomial, multinomial and gaussian distributions

Figure 7.2:

coin flips. For other types of data, other distributions become more appropriate. The die example from before corresponds to a **discrete distribution**. If the data is continuous, you might choose to use a **Gaussian distribution** (aka **Normal distribution**). The choice of distribution is a form of **inductive bias** by which you can inject your knowledge of the problem into the learning algorithm.

## 7.4 Prediction

Consider the *predictions* made by the naive Bayes model with Bernoulli features in Eq (7.18). You can better understand this model by considering its decision boundary. In the case of probabilistic models, the decision boundary is the set of inputs for which the likelihood of $y = +1$ is precisely 50%. Or, in other words, the set of inputs $\boldsymbol{x}$ for which $p(y = +1 \mid \boldsymbol{x})/p(y = -1 \mid \boldsymbol{x}) = 1$. In order to do this, the first thing to notice is that $p(y \mid \boldsymbol{x}) = p(y, \boldsymbol{x})/p(\boldsymbol{x})$. In the ratio, the $p(\boldsymbol{x})$ terms cancel, leaving $p(y = +1, \boldsymbol{x})/p(y = -1, \boldsymbol{x})$. Instead of computing this ratio, it is easier to compute the **log-likelihood ratio** (or LLR), $\log p(y = +1, \boldsymbol{x}) - \log p(y = -1, \boldsymbol{x})$, computed below:

$$\text{LLR} = \log\left[\theta_0 \prod_d \theta_{(+1),d}^{[x_d=1]}(1-\theta_{(+1),d})^{[x_d=0]}\right] - \log\left[(1-\theta_0)\prod_d \theta_{(-1),d}^{[x_d=1]}(1-\theta_{(-1),d})^{[x_d=0]}\right] \quad \text{model assumptions} \tag{7.23}$$

$$= \log\theta_0 - \log(1-\theta_0) + \sum_d [x_d = 1]\left(\log\theta_{(+1),d} - \log\theta_{(-1),d}\right) + \sum_d [x_d = 0]\left(\log(1-\theta_{(+1),d}) - \log(1-\theta_{(-1),d})\right) \quad \text{take logs and rearrange} \tag{7.24}$$

$$= \sum_d x_d \log\frac{\theta_{(+1),d}}{\theta_{(-1),d}} + \sum_d (1-x_d)\log\frac{1-\theta_{(+1),d}}{1-\theta_{(-1),d}} + \log\frac{\theta_0}{1-\theta_0} \quad \text{simplify log terms} \tag{7.25}$$

$$= \sum_d x_d\left[\log\frac{\theta_{(+1),d}}{\theta_{(-1),d}} - \log\frac{1-\theta_{(+1),d}}{1-\theta_{(-1),d}}\right] + \sum_d \log\frac{1-\theta_{(+1),d}}{1-\theta_{(-1),d}} + \log\frac{\theta_0}{1-\theta_0} \quad \text{group } x\text{-terms} \tag{7.26}$$

$$= \boldsymbol{x}\cdot\boldsymbol{w} + b \tag{7.27}$$

$$w_d = \log \frac{\theta_{(+1),d}(1-\theta_{(-1),d})}{\theta_{(-1),d}(1-\theta_{(+1),d})} \quad , \quad b = \sum_d \log \frac{1-\theta_{(+1),d}}{1-\theta_{(-1),d}} + \log \frac{\theta_0}{1-\theta_0} \tag{7.28}$$

The result of the algebra is that the naive Bayes model has precisely the form of a linear model! Thus, like perceptron and many of the other models you've previous studied, the decision boundary is linear.

TODO: MBR

## 7.5 Generative Stories

A useful way to develop probabilistic models is to tell a **generative story**. This is a *fictional* story that explains how you believe your training data came into existence. To make things interesting, consider a multiclass classification problem, with continuous features modeled by independent Gaussians. Since the label can take values $1 \dots K$, you can use a discrete distribution (die roll) to model it (as opposed to the Bernoilli distribution from before):

1. For each example $n = 1 \dots N$:
   (a) Choose a label $y_n \sim \mathcal{D}isc(\boldsymbol{\theta})$
   (b) For each feature $d = 1 \dots D$:
      i. Choose feature value $x_{n,d} \sim \mathcal{N}or(\mu_{y_n,d}, \sigma^2_{y_n,d})$

This generative story can be directly translated into a likelihood function by replacing the "for each"s with products:

$$p(D) = \overbrace{\prod_n \underbrace{\theta_{y_n}}_{\text{choose label}} \underbrace{\prod_d \frac{1}{\sqrt{2\pi\sigma^2_{y_n,d}}} \exp\left[-\frac{1}{2\sigma^2_{y_n,d}}(x_{n,d} - \mu_{y_n,d})^2\right]}_{\text{choose feature value}}}^{\text{for each example}} \underbrace{\phantom{\prod_d \frac{1}{\sqrt{2\pi\sigma^2_{y_n,d}}} \exp\left[-\frac{1}{2\sigma^2_{y_n,d}}(x_{n,d} - \mu_{y_n,d})^2\right]}}_{\text{for each feature}} \tag{7.29}$$

You can take logs to arrive at the log-likelihood:

$$\log p(D) = \sum_n \left[\log \theta_{y_n} + \sum_d -\frac{1}{2}\log(\sigma^2_{y_n,d}) - \frac{1}{2\sigma^2_{y_n,d}}(x_{n,d} - \mu_{y_n,d})^2\right] + const \tag{7.30}$$

To optimize for $\boldsymbol{\theta}$, you need to add a "sums to one" constraint as before. This leads to the previous solution where the $\theta_k$s are proportional to the number of examples with label $k$. In the case of the $\mu$s

you can take a derivative with respect to, say $\mu_{k,i}$ and obtain:

$$\frac{\partial \log p(D)}{\partial \mu_{k,i}} = \frac{\partial}{\partial \mu_{k,i}} - \sum_n \sum_d \frac{1}{2\sigma^2_{y_n,d}} (x_{n,d} - \mu_{y_n,d})^2 \qquad \text{ignore irrelevant terms} \tag{7.31}$$

$$= \frac{\partial}{\partial \mu_{k,i}} - \sum_{n:y_n=k} \frac{1}{2\sigma^2_{k,d}} (x_{n,i} - \mu_{k,i})^2 \qquad \text{ignore irrelevant terms} \tag{7.32}$$

$$= \sum_{n:y_n=k} \frac{1}{\sigma^2_{k,d}} (x_{n,i} - \mu_{k,i}) \qquad \text{take derivative} \tag{7.33}$$

Setting this equal to zero and solving yields:

$$\mu_{k,i} = \frac{\sum_{n:y_n=k} x_{n,i}}{\sum_{n:y_n=k} 1} \tag{7.34}$$

Namely, the sample mean of the $i$th feature of the data points that fall in class $k$. A similar analysis for $\sigma^2_{k,i}$ yields:

$$\frac{\partial \log p(D)}{\partial \sigma^2_{k,i}} = \frac{\partial}{\partial \sigma^2_{k,i}} - \sum_{y:y_n=k} \left[ \frac{1}{2} \log(\sigma^2_{k,i}) + \frac{1}{2\sigma^2_{k,i}} (x_{n,i} - \mu_{k,i})^2 \right] \qquad \text{ignore irrelevant terms} \tag{7.35}$$

$$= - \sum_{y:y_n=k} \left[ \frac{1}{2\sigma^2_{k,i}} - \frac{1}{2(\sigma^2_{k,i})^2} (x_{n,i} - \mu_{k,i})^2 \right] \qquad \text{take derivative} \tag{7.36}$$

$$= \frac{1}{2\sigma^4_{k,i}} \sum_{y:y_n=k} \left[ (x_{n,i} - \mu k, i)^2 - \sigma^2_{k,i} \right] \qquad \text{simplify} \tag{7.37}$$

You can now set this equal to zero and solve, yielding:

$$\sigma^2_{k,i} = \frac{\sum_{n:y_n=k} (x_{n,i} - \mu_{k,i})^2}{\sum_{n:y_n=k} 1} \tag{7.38}$$

Which is just the sample variance of feature $i$ for class $k$.

? What would the estimate be if you decided that, for a given class $k$, all features had equal variance? What if you assumed feature $i$ had equal variance for each class? Under what circumstances might it be a good idea to make such assumptions?

## 7.6 Conditional Models

In the foregoing examples, the task was formulated as attempting to model the **joint** distribution of $(\boldsymbol{x}, y)$ pairs. This may seem wasteful: at prediction time, all you care about is $p(y \mid \boldsymbol{x})$, so why not model it directly?

Starting with the case of regression is actually somewhat simpler than starting with classification in this case. Suppose you "believe"

that the relationship between the real value $y$ and the vector $\boldsymbol{x}$ should be linear. That is, you expect that $y = \boldsymbol{w} \cdot \boldsymbol{x} + b$ *should hold* for some parameters $(\boldsymbol{w}, b)$. Of course, the data that you get does not exactly obey this: that's fine, you can think of deviations from $y = \boldsymbol{w} \cdot \boldsymbol{x} + b$ as *noise*. To form a probabilistic model, you must assume some distribution over noise; a convenient choice is zero-mean Gaussian noise. This leads to the following generative story:

1. For each example $n = 1 \dots N$:
   (a) Compute $t_n = \boldsymbol{w} \cdot \boldsymbol{x}_n + b$
   (b) Choose noise $e_n \sim \mathcal{N}or(0, \sigma^2)$
   (c) Return $y_n = t_n + e_n$

In this story, the variable $t_n$ stands for "target." It is the noiseless variable that you do not get to observe. Similarly $e_n$ is the error (noise) on example $n$. The value that you actually get to observe is $y_n = t_n + e_n$. See Figure 7.3.

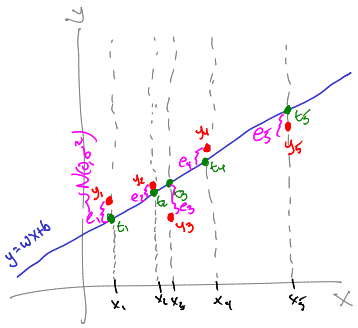


Figure 7.3: pictorial view of targets versus labels

A basic property of the Gaussian distribution is additivity. Namely, that if $a \sim \mathcal{N}or(\mu, \sigma^2)$ and $b = a + c$, then $b \sim \mathcal{N}or(\mu + c, \sigma^2)$. Given this, from the generative story above, you can derive a shorter generative story:

1. For each example $n = 1 \dots N$:
   (a) Choose $y_n \sim \mathcal{N}or(\boldsymbol{w} \cdot \boldsymbol{x}_n + b, \sigma^2)$

Reading off the log likelihood of a dataset from this generative story, you obtain:

$$\log p(D) = \sum_n \left[ -\frac{1}{2} \log(\sigma^2) - \frac{1}{2\sigma^2} (\boldsymbol{w} \cdot \boldsymbol{x}_n + b - y_n)^2 \right] \quad \text{model assumptions} \tag{7.39}$$

$$= -\frac{1}{2\sigma^2} \sum_n (\boldsymbol{w} \cdot \boldsymbol{x}_n + b - y_n)^2 + const \quad \text{remove constants} \tag{7.40}$$

This is precisely the linear regression model you encountered in Section 6.6! To minimizing the *negative* log probability, you need only solve for the regression coefficients $\boldsymbol{w}, b$ as before.

In the case of binary classification, using a Gaussian noise model does not make sense. Switching to a Bernoulli model, which describes binary outcomes, makes more sense. The only remaining difficulty is that the parameter of a Bernoulli is a value between zero and one (the probability of "heads") so your model must produce such values. A classic approach is to produce a real-valued target, as before, and then transform this target into a value between zero and

one, so that $-\infty$ maps to 0 and $+\infty$ maps to 1. A function that does this is the logistic function[1], defined below and plotted in Figure **??**:

[1] Also called the **sigmoid function** because of it's "S"-shape.

$$\text{Logistic function:} \quad \sigma(z) = \frac{1}{1+\exp[-z]} = \frac{\exp z}{1+\exp z} \tag{7.41}$$

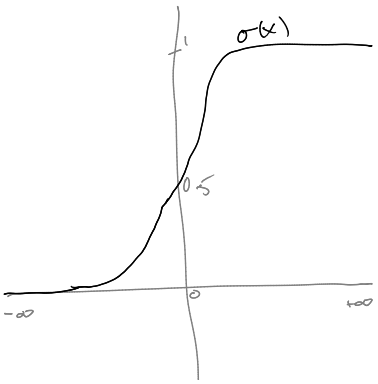



Figure 7.4: sketch of logistic function

The logistic function has several nice properties that you can verify for yourself: $\sigma(-z) = 1 - \sigma(z)$ and $\partial\sigma/\partial z = z\sigma^2(z)$.

Using the logistic function, you can write down a generative story for binary classification:

1. For each example $n = 1 \dots N$:
   (a) Compute $t_n = \sigma(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)$
   (b) Compute $z_n \sim \mathcal{B}er(t_n)$
   (c) Return $y_n = 2z_n - 1$ (to make it $\pm 1$)

The log-likelihood for this model is:

$$\log p(D) = \sum_n \Big[ [y_n = +1] \log \sigma(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) + [y_n = -1] \log \sigma(-\boldsymbol{w} \cdot \boldsymbol{x}_n + b) \Big] \qquad \text{model and properties of } \sigma \tag{7.42}$$

$$= \sum_n \log \sigma(y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)) \qquad \text{join terms} \tag{7.43}$$

$$= -\sum_n \log[1 + \exp(-y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b))] \qquad \text{definition of } \sigma \tag{7.44}$$

$$= -\sum_n \ell^{(\log)}(y_n, \boldsymbol{w} \cdot \boldsymbol{x}_n + b) \qquad \text{definition of } \ell^{(\log)} \tag{7.45}$$

As you can see, the log-likelihood is *precisely* the negative of (a scaled version of) the logistic loss from Chapter 6. This model is the **logistic regression** model, and this is where logisitic loss originally derived from.

TODO: conditional versus joint

## 7.7 Regularization via Priors

In the foregoing discussion, parameters of the model were selected according to the maximum likelihood criteria: find the parameters $\boldsymbol{\theta}$ that maximize $p_{\boldsymbol{\theta}}(D)$. The trouble with this approach is easy to see even in a simple coin flipping example. If you flip a coin twice and it comes up heads both times, the maximum likelihood estimate

for the bias of the coin is 100%: it will *always* come up heads. This is true even if you had only flipped it once! If course if you had flipped it one million times and it had come up heads every time, *then* you might find this to be a reasonable solution.

This is clearly undesirable behavior, especially since data is expensive in a machine learning setting. One solution (there are others!) is to seek parameters that balance a tradeoff between the likelihood of the data and some prior belief you have about what values of those parameters are likely. Taking the case of the logistic regression, you might a priori believe that small values of $\boldsymbol{w}$ are more likely than large values, and choose to represent this as a Gaussian prior on each component of $\boldsymbol{w}$.

The **maximum a posteriori** principle is a method for incoporating both data and prior beliefs to obtain a more balanced parameter estimate. In abstract terms, consider a probabilistic model over data $D$ that is parameterized by parameters $\theta$. If you think of the parameters as just another random variable, then you can write this model as $p(D \mid \theta)$, and maximum likelihood amounts to choosing $\theta$ to maximize $p(D \mid \theta)$. However, you might instead with to maximize the probability of the *parameters*, given the data. Namely, maximize $p(\theta \mid D)$. This term is known as the **posterior** distribution on $\theta$, and can be computed by Bayes' rule:

$$\underbrace{p(\theta \mid D)}_{\text{posterior}} = \frac{\overbrace{p(\theta)}^{\text{prior}}\overbrace{p(D \mid \theta)}^{\text{likelihood}}}{\underbrace{p(D)}_{\text{evidence}}} \quad \text{, where} \quad p(D) = \int \mathrm{d}\theta p(\theta) p(D \mid \theta) \tag{7.46}$$

This reads: the **posterior** is equal to the **prior** times the **likelihood** divided by the **evidence**.[2] The evidence is a scary-looking term (it has an integral!) but note that from the perspective of seeking parameters $\theta$ than maximize the posterior, the evidence is just a constant (it does not depend on $\theta$) and therefore can be ignored.

[2] The evidence is sometimes called the **marginal likelihood**.

Returning to the logistic regression example with Gaussian priors on the weights, the **log posterior** looks like:

$$\log p(\theta \mid D) = -\sum_n \ell^{(\log)}(y_n, \boldsymbol{w} \cdot \boldsymbol{x}_n + b) - \sum_d \frac{1}{2\sigma^2} w_d^2 + const \qquad \text{model definition} \tag{7.47}$$

$$= -\sum_n \ell^{(\log)}(y_n, \boldsymbol{w} \cdot \boldsymbol{x}_n + b) - \frac{1}{2\sigma^2} ||\boldsymbol{w}||^2 \tag{7.48}$$

and therefore reduces to a regularized logistic function, with a squared 2-norm regularizer on the weights. (A 1-norm regularizer

can be obtained by using a Laplace prior on $w$ rather than a Gaussian prior on $w$.)

## 7.8 Exercises

**Exercise 7.1. TODO…**

# 8 | Neural Networks

**Learning Objectives:**

- Explain the biological inspiration for multi-layer neural networks.
- Construct a two-layer network that can solve the XOR problem.
- Implement the back-propogation algorithm for training multi-layer networks.
- Explain the trade-off between depth and breadth in network structure.
- Contrast neural networks with radial basis functions with $k$-nearest neighbor learning.

Dependencies:

The first learning models you learned about (decision trees and nearest neighbor models) created complex, **non-linear** decision boundaries. We moved from there to the perceptron, perhaps the most classic linear model. At this point, we will move *back* to non-linear learning models, but using all that we have learned about linear learning thus far.

This chapter presents an extension of perceptron learning to non-linear decision boundaries, taking the biological inspiration of neurons even further. In the perceptron, we thought of the input data point (eg., an image) as being directly connected to an output (eg., label). This is often called a **single-layer network** because there is one layer of weights. Now, instead of directly connecting the inputs to the outputs, we will insert a layer of "hidden" nodes, moving from a single-layer network to a **multi-layer network**. But introducing a non-linearity at inner layers, this will give us non-linear decision boundaires. In fact, such networks are able to express almost any function we want, not just linear functions. The trade-off for this flexibility is increased complexity in parameter tuning and model design.

## 8.1 Bio-inspired Multi-Layer Networks

One of the major weaknesses of linear models, like perceptron and the regularized linear models from the previous chapter, is that they are linear! Namely, they are unable to learn arbitrary decision boundaries. In contrast, decision trees and KNN *could* learn arbitrarily complicated decision boundaries.

One approach to doing this is to chain together a collection of perceptrons to build more complex **neural networks**. An example of a **two-layer network** is shown in Figure 8.1. Here, you can see five inputs (features) that are fed into two **hidden units**. These hidden units are then fed in to a single **output unit**. Each edge in this figure corresponds to a different weight. (Even though it looks like there are three layers, this is called a two-layer network because we don't count

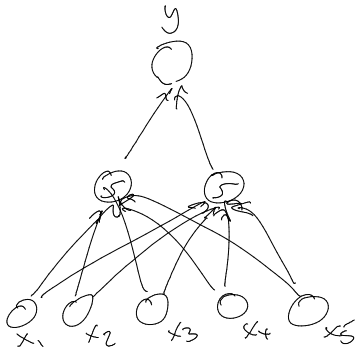



Figure 8.1: picture of a two-layer network with 5 inputs and two hidden units

the inputs as a real layer. That is, it's two layers of *trained* weights.)

Prediction with a neural network is a straightforward generalization of prediction with a perceptron. First you compute activations of the nodes in the hidden unit based on the inputs and the input weights. Then you compute activations of the output unit given the hidden unit activations and the second layer of weights.

The only major difference between this computation and the perceptron computation is that the hidden units compute a *non-linear* function of their inputs. This is usually called the **activation function** or **link function**. More formally, if $w_{i,d}$ is the weights on the edge connecting input $d$ to hidden unit $i$, then the activation of hidden unit $i$ is computed as:

$$h_i = f(\boldsymbol{w}_i \cdot \boldsymbol{x}) \tag{8.1}$$

Where $f$ is the link function and $\boldsymbol{w}_i$ refers to the vector of weights feeding in to node $i$.

One example link function is the **sign** function. That is, if the incoming signal is negative, the activation is $-1$. Otherwise the activation is $+1$. This is a potentially useful activiation function, but you might already have guessed the problem with it: it is non-differentiable.

EXPLAIN BIAS!!!

A more popular link function is the **hyperbolic tangent** function, tanh. A comparison between the sign function and the tanh function is in Figure 8.2. As you can see, it is a reasonable approximation to the sign function, but is convenient in that it is differentiable.[1] Because it looks like an "S" and because the Greek character for "S" is "Sigma," such functions are usually called **sigmoid** functions.

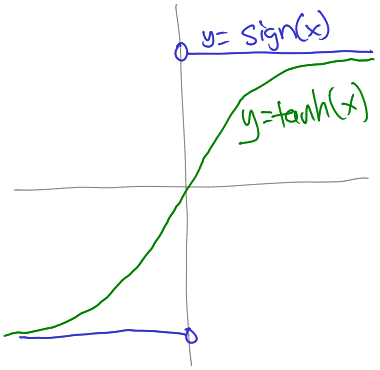



Figure 8.2: picture of sign versus tanh

[1] It's derivative is just $1 - \tanh^2(x)$.

Assuming for now that we are using tanh as the link function, the overall prediction made by a two-layer network can be computed using Algorithm 8.1. This function takes a matrix of weights $\mathbf{W}$ corresponding to the first layer weights and a vector of weights $\boldsymbol{v}$ corresponding to the second layer. You can write this entire computation out in one line as:

$$\hat{y} = \sum_i v_i \tanh(\boldsymbol{w}_i \cdot \hat{\boldsymbol{x}}) \tag{8.2}$$

$$= \boldsymbol{v} \cdot \tanh(\mathbf{W}\hat{\boldsymbol{x}}) \tag{8.3}$$

Where the second line is short hand assuming that tanh can take a vector as input and product a vector as output.

? Is it necessary to use a link function at all? What would happen if you just used the identify function as a link?

**Algorithm 24** TwoLayerNetworkPredict($\mathbf{W}$, $\boldsymbol{v}$, $\hat{\boldsymbol{x}}$)

```
for i = 1 to number of hidden units do
    h_i ← tanh(w_i · x̂)          // compute activation of hidden unit i
end for
return v · h                      // compute output unit
```

The claim is that two-layer neural networks are more expressive than single layer networks (i.e., perceptrons). To see this, you can construct a very small two-layer network for solving the XOR problem. For simplicity, suppose that the data set consists of four data points, given in Table 8.1. The classification rule is that $y = +1$ if an only if $x_1 = x_2$, where the features are just $\pm 1$.

| $y$ | $x_0$ | $x_1$ | $x_2$ |
|---|---|---|---|
| +1 | +1 | +1 | +1 |
| +1 | +1 | -1 | -1 |
| -1 | +1 | +1 | -1 |
| -1 | +1 | -1 | +1 |

Table 8.1: Small XOR data set.

You can solve this problem using a two layer network with two hidden units. The key idea is to make the first hidden unit compute an "or" function: $x_1 \vee x_2$. The second hidden unit can compute an "and" function: $x_1 \wedge x_2$. The the output can combine these into a single prediction that mimics XOR. Once you have the first hidden unit activate for "or" and the second for "and," you need only set the output weights as $-2$ and $+1$, respectively.

? Verify that these output weights will actually give you XOR.

To achieve the "or" behavior, you can start by setting the bias to $-0.5$ and the weights for the two "real" features as both being 1. You can check for yourself that this will do the "right thing" if the link function were the sign function. Of course it's not, it's tanh. To get tanh to mimic sign, you need to make the dot product either really really large or really really small. You can accomplish this by setting the bias to $-500,000$ and both of the two weights to $1,000,000$. Now, the activation of this unit will be just slightly above $-1$ for $\boldsymbol{x} = \langle -1, -1 \rangle$ and just slightly below $+1$ for the other three examples.

? This shows how to create an "or" function. How can you create an "and" function?

At this point you've seen that one-layer networks (aka perceptrons) can represent any linear function and only linear functions. You've also seen that two-layer networks can represent non-linear functions like XOR. A natural question is: do you get additional representational power by moving beyond two layers? The answer is partially provided in the following Theorem, due originally to George Cybenko for one particular type of link function, and extended later by Kurt Hornik to arbitrary link functions.

**Theorem 9** (Two-Layer Networks are Universal Function Approximators)**.** *Let $F$ be a continuous function on a bounded subset of $D$-dimensional space. Then there exists a two-layer neural network $\hat{F}$ with a finite number of hidden units that approximate $F$ arbitrarily well. Namely, for all $\boldsymbol{x}$ in the domain of $F$, $\left|F(\boldsymbol{x}) - \hat{F}(\boldsymbol{x})\right| < \epsilon$.*

Or, in colloquial terms "two-layer networks can approximate any

function."

This is a remarkable theorem. Practically, it says that if you give me a function $F$ and some error tolerance parameter $\epsilon$, I can construct a two layer network that computes $F$. In a sense, it says that going from one layer to two layers completely changes the representational capacity of your model.

When working with two-layer networks, the key question is: how many hidden units should I have? If your data is $D$ dimensional and you have $K$ hidden units, then the total number of parameters is $(D+2)K$. (The first $+1$ is from the bias, the second is from the second layer of weights.) Following on from the heuristic that you should have one to two examples for each parameter you are trying to estimate, this suggests a method for choosing the number of hidden units as roughly $\lfloor \frac{N}{D} \rfloor$. In other words, if you have tons and tons of examples, you can safely have lots of hidden units. If you only have a few examples, you should probably restrict the number of hidden units in your network.

The number of units is both a form of inductive bias and a form of regularization. In both view, the number of hidden units controls how complex your function will be. Lots of hidden units $\Rightarrow$ very complicated function. Figure **??** shows training and test error for neural networks trained with different numbers of hidden units. As the number increases, training performance continues to get better. But at some point, test performance gets worse because the network has overfit the data.

## 8.2 *The Back-propagation Algorithm*

The back-propagation algorithm is a classic approach to training neural networks. Although it was not originally seen this way, based on what you know from the last chapter, you can summarize back-propagation as:

$$\text{back-propagation} = \text{gradient descent} + \text{chain rule} \tag{8.4}$$

More specifically, the set up is *exactly* the same as before. You are going to optimize the weights in the network to minimize some objective function. The only difference is that the predictor is no longer linear (i.e., $\hat{y} = \boldsymbol{w} \cdot \boldsymbol{x} + b$) but now non-linear (i.e., $\boldsymbol{v} \cdot \tanh(\mathbf{W}\hat{\boldsymbol{x}})$). The only question is how to do gradient descent on this more complicated objective.

For now, we will ignore the idea of regularization. This is for two reasons. The first is that you already know how to deal with regularization, so everything you've learned before applies. The second is that *historically*, neural networks have not been regularized. Instead,

people have used **early stopping** as a method for controlling overfitting. Presently, it's not obvious which is a better solution: both are valid options.

To be completely explicit, we will focus on optimizing squared error. Again, this is mostly for historic reasons. You could easily replace squared error with your loss function of choice. Our overall objective is:

$$\min_{\mathbf{W},v} \quad \sum_n \frac{1}{2} \left( y_n - \sum_i v_i f(\boldsymbol{w}_i \cdot \boldsymbol{x}_n) \right)^2 \tag{8.5}$$

Here, $f$ is some link function like tanh.

The easy case is to differentiate this with respect to $\boldsymbol{v}$: the weights for the output unit. Without even doing any math, you should be able to guess what this looks like. The way to think about it is that from $\boldsymbol{v}$s perspective, it is just a linear model, attempting to minimize squared error. The only "funny" thing is that its inputs are the activations $\boldsymbol{h}$ rather than the examples $\boldsymbol{x}$. So the gradient with respect to $\boldsymbol{v}$ is just as for the linear case.

To make things notationally more convenient, let $e_n$ denote the *error* on the $n$th example (i.e., the blue term above), and let $\boldsymbol{h}_n$ denote the vector of hidden unit activations on that example. Then:

$$\nabla_v = -\sum_n e_n \boldsymbol{h}_n \tag{8.6}$$

This is exactly like the linear case. One way of interpreting this is: how would the output weights have to change to make the prediction better? This is an easy question to answer because they can easily measure how their changes affect the output.

The more complicated aspect to deal with is the weights corresponding to the *first* layer. The reason this is difficult is because the weights in the first layer aren't necessarily trying to produce specific values, say 0 or 5 or $-2.1$. They are simply trying to produce activations that get fed to the output layer. So the change they want to make depends crucially on how the output layer interprets them.

Thankfully, the chain rule of calculus saves us. Ignoring the sum over data points, we can compute:

$$\mathcal{L}(\mathbf{W}) = \frac{1}{2} \left( y - \sum_i v_i f(\boldsymbol{w}_i \cdot \boldsymbol{x}) \right)^2 \tag{8.7}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}_i} = \frac{\partial \mathcal{L}}{\partial f_i} \frac{\partial f_i}{\partial \boldsymbol{w}_i} \tag{8.8}$$

$$\frac{\partial \mathcal{L}}{\partial f_i} = -\left( y - \sum_i v_i f(\boldsymbol{w}_i \cdot \boldsymbol{x}) \right) v_i = -e v_i \tag{8.9}$$

$$\frac{\partial f_i}{\partial \boldsymbol{w}_i} = f'(\boldsymbol{w}_i \cdot \boldsymbol{x}) \boldsymbol{x} \tag{8.10}$$

**Algorithm 25** TwoLayerNetworkTrain($\mathbf{D}$, $\eta$, $K$, *MaxIter*)

```
W ← D×K matrix of small random values          // initialize input layer weights
v ← K-vector of small random values            // initialize output layer weights
for iter = 1 ... MaxIter do
    G ← D×K matrix of zeros                    // initialize input layer gradient
    g ← K-vector of zeros                      // initialize output layer gradient
    for all (x,y) ∈ D do
        for i = 1 to K do
            a_i ← w_i · x̂
            h_i ← tanh(a_i)                    // compute activation of hidden unit i
        end for
        ŷ ← v · h                              // compute output unit
        e ← y − ŷ                              // compute error
        g ← g − eh                             // update gradient for output layer
        for i = 1 to K do
            G_i ← G_i − e v_i (1 − tanh^2(a_i)) x   // update gradient for input layer
        end for
    end for
    W ← W − ηG                                 // update input layer weights
    v ← v − ηg                                 // update output layer weights
end for
return W, v
```

Putting this together, we get that the gradient with respect to $\boldsymbol{w}_i$ is:

$$\nabla_{\boldsymbol{w}_i} = -e v_i f'(\boldsymbol{w}_i \cdot \boldsymbol{x}) \boldsymbol{x} \tag{8.11}$$

Intuitively you can make sense of this. If the overall error of the predictor ($e$) is small, you want to make small steps. If $v_i$ is small for hidden unit $i$, then this means that the output is not particularly sensitive to the activation of the $i$th hidden unit. Thus, its gradient should be small. If $v_i$ flips sign, the gradient at $\boldsymbol{w}_i$ should also flip signs. The name **back-propagation** comes from the fact that you propagate gradients backward through the network, starting at the end.

The complete instantiation of gradient descent for a two layer network with $K$ hidden units is sketched in Algorithm 8.2. Note that this really is *exactly* a gradient descent algorithm; the only different is that the computation of the gradients of the input layer is moderately complicated.

As a bit of practical advice, implementing the back-propagation algorithm can be a bit tricky. Sign errors often abound. A useful trick is first to keep $\mathbf{W}$ fixed and work on just training $\boldsymbol{v}$. Then keep $\boldsymbol{v}$ fixed and work on training $\mathbf{W}$. Then put them together.

? What would happen to this algorithm if you wanted to optimize exponential loss instead of squared error? What if you wanted to add in weight regularization?

? If you like matrix calculus, derive the same algorithm starting from Eq (8.3).

## 8.3 Initialization and Convergence of Neural Networks

Based on what you know about linear models, you might be tempted to initialize all the weights in a neural network to zero. You might also have noticed that in Algorithm **??**, this is not what's done: they're initialized to small random values. The question is why?

The answer is because an initialization of $\mathbf{W} = \mathbf{0}$ and $\boldsymbol{v} = \mathbf{0}$ will lead to "uninteresting" solutions. In other words, if you initialize the model in this way, it will eventually get stuck in a bad local optimum. To see this, first realize that on any example $\boldsymbol{x}$, the activation $h_i$ of the hidden units will all be zero since $\mathbf{W} = \mathbf{0}$. This means that on the first iteration, the gradient on the output weights ($\boldsymbol{v}$) will be zero, so they will stay put. Furthermore, the gradient $w_{1,d}$ for the $d$th feature on the $i$th unit will be *exactly* the same as the gradient $w_{2,d}$ for the same feature on the second unit. This means that the weight matrix, after a gradient step, will change in *exactly the same way* for every hidden unit. Thinking through this example for iterations $2\ldots$, the values of the hidden units will *always* be exactly the same, which means that the weights feeding in to any of the hidden units will be exactly the same. Eventually the model will converge, but it will converge to a solution that does not take advantage of having access to the hidden units.

This shows that neural networks are *sensitive* to their initialization. In particular, the function that they optimize is **non-convex**, meaning that it might have plentiful local optima. (One of which is the trivial local optimum described in the preceding paragraph.) In a sense, neural networks *must* have local optima. Suppose you have a two layer network with two hidden units that's been optimized. You have weights $\boldsymbol{w}_1$ from inputs to the first hidden unit, weights $\boldsymbol{w}_2$ from inputs to the second hidden unit and weights $(v_1, v_2)$ from the hidden units to the output. If I give you back another network with $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ swapped, and $v_1$ and $v_2$ swapped, the network computes *exactly* the same thing, but with a markedly different weight structure. This phenomena is known as **symmetric modes** ("mode" referring to an optima) meaning that there are symmetries in the weight space. It would be one thing if there were lots of modes and they were all symmetric: then finding one of them would be as good as finding any other. Unfortunately there are additional local optima that are *not* global optima.

Random initialization of the weights of a network is a way to address *both* of these problems. By initializing a network with small random weights (say, uniform between $-0.1$ and $0.1$), the network is unlikely to fall into the trivial, symmetric local optimum. Moreover, by training a collection of networks, each with a different random

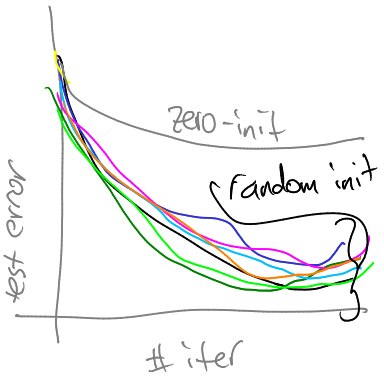



Figure 8.3: convergence of randomly initialized networks

initialization, you can often obtain better solutions that with just one initialization. In other words, you can train ten networks with different random seeds, and then pick the one that does best on held-out data. Figure 8.3 shows prototypical *test-set* performance for ten networks with different random initialization, plus an eleventh plot for the trivial symmetric network initialized with zeros.

One of the typical complaints about neural networks is that they are finicky. In particular, they have a rather large number of knobs to tune:

1. The number of layers
2. The number of hidden units per layer
3. The gradient descent learning rate $\eta$
4. The initialization
5. The stopping iteration or weight regularization

The last of these is minor (early stopping is an easy regularization method that does not require much effort to tune), but the others are somewhat significant. Even for two layer networks, having to choose the number of hidden units, and then get the learning rate and initialization "right" can take a bit of work. Clearly it can be automated, but nonetheless it takes time.

Another difficulty of neural networks is that their weights can be difficult to interpret. You've seen that, for linear networks, you can often interpret high weights as indicative of positive examples and low weights as indicative of negative examples. In multilayer networks, it becomes very difficult to try to understand what the different hidden units are doing.

## 8.4 Beyond Two Layers

The definition of neural networks and the back-propagation algorithm can be generalized beyond two layers to any arbitrary directed acyclic graph. In practice, it is most common to use a layered network like that shown in Figure 8.4 unless one has a very strong reason (aka inductive bias) to do something different. However, the view as a directed graph sheds a different sort of insight on the back-propagation algorithm.

Suppose that your network structure is stored in some directed acyclic graph, like that in Figure 8.5. We index nodes in this graph as $u, v$. The activation *before* applying non-linearity at a node is $a_u$ and after non-linearity is $h_u$. The graph has a single sink, which is the output node $y$ with activation $a_y$ (no non-linearity is performed

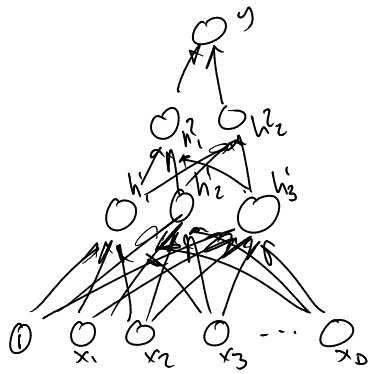



Figure 8.4: multi-layer network

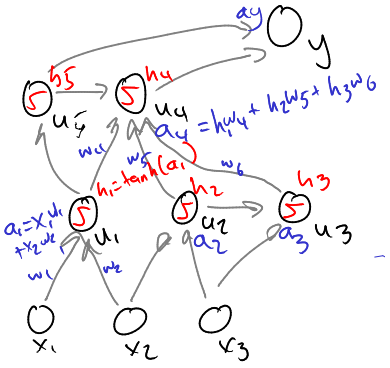



Figure 8.5: DAG network

**Algorithm 26** ForwardPropagation($x$)

```
for all input nodes u do
    h_u ← corresponding feature of x
end for
for all nodes v in the network whose parent's are computed do
    a_v ← Σ_{u∈par(v)} w_{(u,v)} h_u
    h_v ← tanh(a_v)
end for
return a_y
```

**Algorithm 27** BackPropagation($x$, $y$)

```
run ForwardPropagation(x) to compute activations
e_y ← y − a_y                                   // compute overall network error
for all nodes v in the network whose error e_v is computed do
    for all u ∈ par(v) do
        g_{u,v} ← −e_v h_u                      // compute gradient of this edge
        e_u ← e_u + e_v w_{u,v} (1 − tanh²(a_u))  // compute the "error" of the parent node
    end for
end for
return all gradients g_e
```

on the output unit). The graph has $D$-many inputs (i.e., nodes with no parent), whose activations $h_u$ are given by an input example. An edge $(u, v)$ is from a parent to a child (i.e., from an input to a hidden unit, or from a hidden unit to the sink). Each edge has a weight $w_{u,v}$. We say that $par(u)$ is the set of parents of $u$.

There are two relevant algorithms: forward-propagation and back-propagation. Forward-propagation tells you how to compute the activation of the sink $y$ given the inputs. Back-propagation computes derivatives of the edge weights for a given input.

The key aspect of the **forward-propagation** algorithm is to iteratively compute activations, going deeper and deeper in the DAG. Once the activations of all the parents of a node $u$ have been computed, you can compute the activation of node $u$. This is spelled out in Algorithm 8.4. This is also explained pictorially in Figure 8.6.

Back-propagation (see Algorithm 8.4) does the opposite: it computes gradients top-down in the network. The key idea is to compute an *error* for each node in the network. The error at the output unit is the "true error." For any input unit, the error is the amount of gradient that we see coming from our children (i.e., higher in the network). These errors are computed backwards in the network (hence the name **back-propagation**) along with the gradients themselves. This is also explained pictorially in Figure 8.7.

Given the back-propagation algorithm, you can directly run gradient descent, using it as a subroutine for computing the gradients.

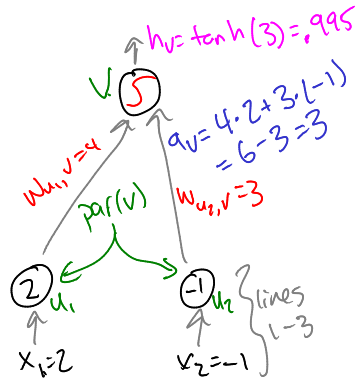



Figure 8.6: picture of forward prop

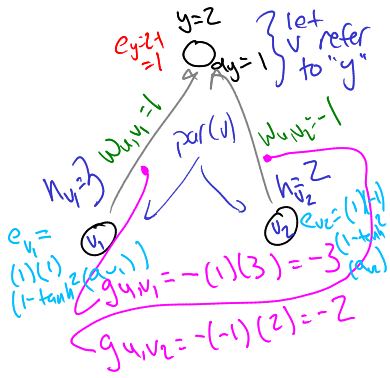



Figure 8.7: picture of back prop

## 8.5 Breadth versus Depth

At this point, you've seen how to train two-layer networks and how to train arbitrary networks. You've also seen a theorem that says that two-layer networks are universal function approximators. This begs the question: if two-layer networks are so great, why do we care about deeper networks?

To understand the answer, we can borrow some ideas from CS theory, namely the idea of **circuit complexity**. The goal is to show that there are functions for which it might be a "good idea" to use a deep network. In other words, there are functions that will require a huge number of hidden units if you force the network to be shallow, but can be done in a small number of units if you allow it to be deep. The example that we'll use is the **parity function** which, ironically enough, is just a generalization of the XOR problem. The function is defined over binary inputs as:

$$parity(\boldsymbol{x}) = \sum_d \boldsymbol{x}_d \mod 2 \tag{8.12}$$

$$= \begin{cases} 1 & \text{if the number of 1s in } \boldsymbol{x} \text{ is odd} \\ 0 & \text{if the number of 1s in } \boldsymbol{x} \text{ is even} \end{cases} \tag{8.13}$$

It is easy to define a circuit of depth $\mathcal{O}(\log_2 D)$ with $\mathcal{O}(D)$-many gates for computing the parity function. Each gate is an XOR, arranged in a complete binary tree, as shown in Figure 8.8. (If you want to disallow XOR as a gate, you can fix this by allowing the depth to be doubled and replacing each XOR with an AND, OR and NOT combination, like you did at the beginning of this chapter.)

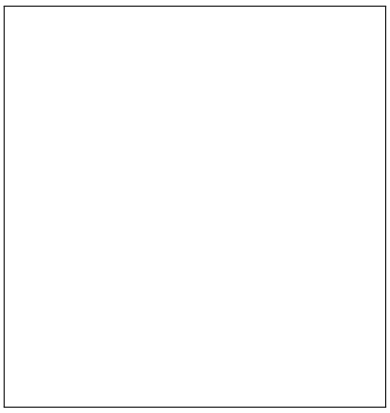


Figure 8.8: `nnet:paritydeep`: deep function for computing parity

This shows that if you are allowed to be deep, you can construct a circuit with that computes parity using a number of hidden units that is linear in the dimensionality. So can you do the same with shallow circuits? The answer is no. It's a famous result of circuit complexity that parity requires exponentially many gates to compute in constant depth. The formal theorem is below:

**Theorem 10** (Parity Function Complexity)**.** *Any circuit of depth $K < \log_2 D$ that computes the parity function of $D$ input bits must contain $\mathcal{O}e^D$ gates.*

This is a very famous result because it shows that constant-depth circuits are less powerful that deep circuits. Although a neural network isn't exactly the same as a circuit, the is generally believed that the same result holds for neural networks. At the very least, this gives a strong indication that depth might be an important consideration in neural networks.

One way of thinking about the issue of breadth versus depth has to do with the number of *parameters* that need to be estimated. By

? What is it about neural networks that makes it so that the theorem about circuits does not apply directly?

the heuristic that you need roughly one or two examples for every parameter, a deep model could potentially require exponentially fewer examples to train than a shallow model!

This now flips the question: if deep is potentially so much better, why doesn't everyone use deep networks? There are at least two answers. First, it makes the **architecture selection** problem more significant. Namely, when you use a two-layer network, the only hyperparameter to choose is how many hidden units should go in the middle layer. When you choose a deep network, you need to choose how many layers, and what is the width of all those layers. This can be somewhat daunting.

A second issue has to do with training deep models with back-propagation. In general, as back-propagation works its way down through the model, the sizes of the gradients shrink. You can work this out mathematically, but the intuition is simpler. If you are the beginning of a very deep network, changing one single weight is unlikely to have a significant effect on the output, since it has to go through so many other units before getting there. This directly implies that the derivatives are small. This, in turn, means that back-propagation essentially never moves far from its initialization when run on very deep networks.

? While these small derivatives might make training difficult, they might be *good* for other reasons: what reasons?

Finding good ways to train deep networks is an active research area. There are two general strategies. The first is to attempt to initialize the weights better, often by a **layer-wise** initialization strategy. This can be often done using unlabeled data. After this initialization, back-propagation can be run to tweak the weights for whatever classification problem you care about. A second approach is to use a more complex optimization procedure, rather than gradient descent. You will learn about some such procedures later in this book.

## 8.6 Basis Functions

At this point, we've seen that: (a) neural networks can mimic linear functions and (b) they can learn more complex functions. A reasonable question is whether they can mimic a *K*NN classifier, and whether they can do it efficiently (i.e., with not-too-many hidden units).

A natural way to train a neural network to mimic a *K*NN classifier is to replace the sigmoid link function with a **radial basis function** (RBF). In a **sigmoid network** (i.e., a network with sigmoid links), the hidden units were computed as $h_i = \tanh(\boldsymbol{w}_i, \boldsymbol{x}\cdot)$. In an **RBF network**, the hidden units are computed as:

$$h_i = \exp\left[-\gamma_i ||\boldsymbol{w}_i - \boldsymbol{x}||^2\right] \tag{8.14}$$

In other words, the hidden units behave like little Gaussian "bumps" centered around locations specified by the vectors $\boldsymbol{w}_i$. A one-dimensional example is shown in Figure 8.9. The parameter $\gamma_i$ specifies the *width* of the Gaussian bump. If $\gamma_i$ is large, then only data points that are really close to $\boldsymbol{w}_i$ have non-zero activations. To distinguish sigmoid networks from RBF networks, the hidden units are typically drawn with sigmoids or with Gaussian bumps, as in Figure 8.10.

Training RBF networks involves finding good values for the Gassian widths, $\gamma_i$, the centers of the Gaussian bumps, $\boldsymbol{w}_i$ and the connections between the Gaussian bumps and the output unit, $\boldsymbol{v}$. This can all be done using back-propagation. The gradient terms for $\boldsymbol{v}$ remain unchanged from before, the the derivates for the other variables differ (see Exercise **??**).

One of the big questions with RBF networks is: where should the Gaussian bumps be centered? One can, of course, apply back-propagation to attempt to find the centers. Another option is to specify them ahead of time. For instance, one potential approach is to have one RBF unit per data point, centered on that data point. If you carefully choose the $\gamma$s and $\boldsymbol{v}$s, you can obtain something that looks nearly identical to distance-weighted *K*NN by doing so. This has the added advantage that you can go futher, and use back-propagation to *learn* good Gaussian widths ($\gamma$) and "voting" factors ($\boldsymbol{v}$) for the nearest neighbor algorithm.

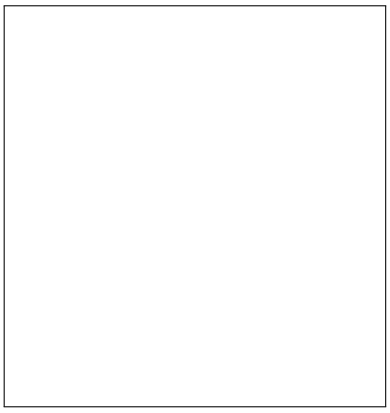


Figure 8.9: `nnet:rbfpicture`: a one-D picture of RBF bumps

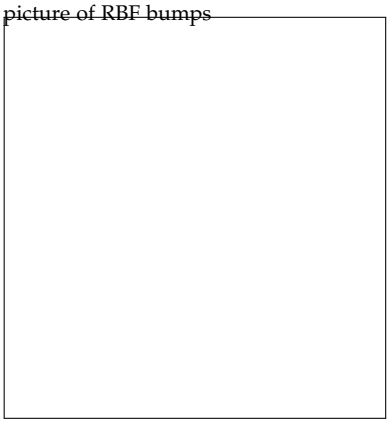


Figure 8.10: `nnet:unitsymbols`: picture of nnet with sigmoid/rbf units

## 8.7 *Exercises*

**Exercise 8.1. TODO...**

? Consider an RBF network with one hidden unit per training point, centered at that point. What bad thing might happen if you use back-propagation to estimate the $\gamma$s and $\boldsymbol{v}$ on this data if you're not careful? How could you be careful?

# 9 | Kernel Methods

**Learning Objectives:**

- Explain how kernels generalize both feature combinations and basis functions.
- Contrast dot products with kernel products.
- Implement kernelized perceptron.
- Derive a kernelized version of regularized least squares regression.
- Implement a kernelized version of the perceptron.
- Derive the dual formulation of the support vector machine.

Dependencies:

Linear models are great because they are easy to understand and easy to optimize. They suffer because they can only learn very simple decision boundaries. Neural networks can learn more complex decision boundaries, but lose the nice convexity properties of many linear models.

One way of getting a linear model to behave non-linearly is to transform the input. For instance, by adding feature pairs as additional inputs. Learning a linear model on such a representation is convex, but is computationally prohibitive in all but very low dimensional spaces. You might ask: instead of *explicitly* expanding the feature space, is it possible to stay with our original data representation and do all the feature blow up *implicitly*? Surprisingly, the answer is often "yes" and the family of techniques that makes this possible are known as **kernel** approaches.

## 9.1 From Feature Combinations to Kernels

In Section 4.4, you learned one method for increasing the expressive power of linear models: explode the feature space. For instance, a "quadratic" feature explosion might map a feature vector $\boldsymbol{x} = \langle x_1, x_2, x_3, \ldots, x_D \rangle$ to an expanded version denoted $\phi(\boldsymbol{x})$:

$$\begin{aligned}\phi(\boldsymbol{x}) = \langle & 1, 2x_1, 2x_2, 2x_3, \ldots, 2x_D, \\ & x_1^2, x_1x_2, x_1x_3, \ldots, x_1x_D, \\ & x_2x_1, x_2^2, x_2x_3, \ldots, x_2x_D, \\ & x_3x_1, x_3x_2, x_3^2, \ldots, x_2x_D, \\ & \ldots, \\ & x_Dx_1, x_Dx_2, x_Dx_3, \ldots, x_D^2 \rangle \end{aligned} \tag{9.1}$$

(Note that there are repetitions here, but hopefully most learning algorithms can deal well with redundant features; in particular, the $2x_1$ terms are due to collapsing some repetitions.)

You could then train a classifier on this expanded feature space. There are two primary concerns in doing so. The first is computational: if your learning algorithm scales linearly in the number of features, then you've just squared the amount of computation you need to perform; you've also squared the amount of memory you'll need. The second is statistical: if you go by the heuristic that you should have about two examples for every feature, then you will now need quadratically many training examples in order to avoid overfitting.

This chapter is all about dealing with the *computational* issue. It will turn out in Chapter **??** that you can also deal with the statistical issue: for now, you can just hope that regularization will be sufficient to attenuate overfitting.

The key insight in kernel-based learning is that you can *rewrite* many linear models in a way that doesn't require you to ever *explicitly* compute $\phi(\boldsymbol{x})$. To start with, you can think of this purely as a computational "trick" that enables you to use the power of a quadratic feature mapping without actually having to compute and store the mapped vectors. Later, you will see that it's actually quite a bit deeper. Most algorithms we discuss involve a product of the form $\boldsymbol{w} \cdot \phi(\boldsymbol{x})$, after performing the feature mapping. The goal is to rewrite these algorithms so that they only ever depend on dot products between two examples, say $\boldsymbol{x}$ and $\boldsymbol{z}$; namely, they depend on $\phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{z})$. To understand why this is helpful, consider the quadratic expansion from above, and the dot-product between two vectors. You get:

$$\begin{aligned}
\phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{z}) &= 1 + x_1 z_1 + x_2 z_2 + \cdots + x_D z_D + x_1^2 z_1^2 + \cdots + x_1 x_D z_1 z_D + \\
&\quad \cdots + x_D x_1 z_D z_1 + x_D x_2 z_D z_2 + \cdots + x_D^2 z_D^2 && (9.2)\\
&= 1 + 2\sum_d x_d z_d + \sum_d \sum_e x_d x_e z_d z_e && (9.3)\\
&= 1 + 2\boldsymbol{x} \cdot \boldsymbol{z} + (\boldsymbol{x} \cdot \boldsymbol{z})^2 && (9.4)\\
&= (1 + \boldsymbol{x} \cdot \boldsymbol{z})^2 && (9.5)
\end{aligned}$$

Thus, you can compute $\phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{z})$ in exactly the same amount of time as you can compute $\boldsymbol{x} \cdot \boldsymbol{z}$ (plus the time it takes to perform an addition and a multiply, about 0.02 nanoseconds on a circa 2011 processor).

The rest of the practical challenge is to rewrite your algorithms so that they only depend on dot products between examples and not on any explicit weight vectors.

## 9.2 Kernelized Perceptron

Consider the original perceptron algorithm from Chapter 3, repeated in Algorithm 9.2 using linear algebra notation and using feature expansion notation $\phi(\boldsymbol{x})$. In this algorithm, there are two places

**Algorithm 28** PERCEPTRONTRAIN(**D**, *MaxIter*)

```
1:  w ← 0, b ← 0                                   // initialize weights and bias
2:  for iter = 1 ... MaxIter do
3:      for all (x,y) ∈ D do
4:          a ← w · φ(x) + b                       // compute activation for this example
5:          if ya ≤ 0 then
6:              w ← w + y φ(x)                     // update weights
7:              b ← b + y                          // update bias
8:          end if
9:      end for
10: end for
11: return w, b
```

**MATH REVIEW | SPANS AND NULL SPACES**

reminder: if $\mathcal{U} = \{\boldsymbol{u}_i\}_i$ is a set of vectors in $\mathbb{R}^D$, then the span of $\mathcal{U}$ is the set of vectors that can be written as linear combinations of $\boldsymbol{u}_i$s; namely: $span(\mathcal{U}) = \{\sum_i a_i \boldsymbol{u}_i : a_1 \in \mathbb{R}, \ldots, a_I \in \mathbb{R}\}$.
the null space of $\mathcal{U}$ is everything that's left: $\mathbb{R}^D \backslash span(\mathcal{U})$.
TODO pictures

Figure 9.1:

where $\phi(\boldsymbol{x})$ is used explicitly. The first is in computing the activation (line 4) and the second is in updating the weights (line 6). The goal is to remove the explicit dependence of this algorithm on $\phi$ and on the weight vector.

To do so, you can observe that at any point in the algorithm, the weight vector $\boldsymbol{w}$ can be written as a linear combination of expanded training data. In particular, at any point, $\boldsymbol{w} = \sum_n \alpha_n \phi(\boldsymbol{x}_n)$ for some parameters $\alpha$. Initially, $\boldsymbol{w} = \boldsymbol{0}$ so choosing $\boldsymbol{\alpha} = 0$ yields this. If the first update occurs on the $n$th training example, then the resolution weight vector is simply $y_n \phi(\boldsymbol{x}_n)$, which is equivalent to setting $\alpha_n = y_n$. If the second update occurs on the $m$th training example, then all you need to do is update $\alpha_m \leftarrow \alpha_m + y_m$. This is true, even if you make multiple passes over the data. This observation leads to the following **representer theorem**, which states that the weight vector of the perceptron lies in the **span** of the training data.

**Theorem 11** (Perceptron Representer Theorem)**.** *During a run of the perceptron algorithm, the weight vector $\boldsymbol{w}$ is always in the span of the (assumed non-empty) training data, $\phi(\boldsymbol{x}_1), \ldots, \phi(\boldsymbol{x}_N)$.*

*Proof of Theorem 11.* By induction. Base case: the span of any non-empty set contains the zero vector, which is the initial weight vector. Inductive case: suppose that the theorem is true before the $k$th update, and suppose that the $k$th update happens on example $n$. By the inductive hypothesis, you can write $\boldsymbol{w} = \sum_i \alpha_i \phi(\boldsymbol{x}_i)$ before

**Algorithm 29** KERNELIZEDPERCEPTRONTRAIN(**D**, *MaxIter*)

```
1:  α ← 0, b ← 0                                   // initialize coefficients and bias
2:  for iter = 1 ... MaxIter do
3:      for all (x_n, y_n) ∈ D do
4:          a ← Σ_m α_m φ(x_m) · φ(x_n) + b        // compute activation for this example
5:          if y_n a ≤ 0 then
6:              α_n ← α_n + y_n                    // update coefficients
7:              b ← b + y                          // update bias
8:          end if
9:      end for
10: end for
11: return α, b
```

the update. The new weight vector is $[\sum_i \alpha_i \phi(\boldsymbol{x}_i)] + y_n \phi(\boldsymbol{x}_n) = \sum_i (\alpha_i + y_n[i = n])\phi(\boldsymbol{x}_i)$, which is still in the span of the training data. □

Now that you know that you can always write $\boldsymbol{w} = \sum_n \alpha_n \phi(\boldsymbol{x}_n)$ for some $\alpha_i$s, you can additionall compute the activations (line 4) as:

$$\boldsymbol{w} \cdot \phi(\boldsymbol{x}) + b = \left( \sum_n \alpha_n \phi(\boldsymbol{x}_n) \right) \cdot \phi(\boldsymbol{x}) + b \qquad \text{definition of } \boldsymbol{w} \tag{9.6}$$

$$= \sum_n \alpha_n \Big[ \phi(\boldsymbol{x}_n) \cdot \phi(\boldsymbol{x}) \Big] + b \qquad \text{dot products are linear} \tag{9.7}$$

This now depends only on dot-products between data points, and never explicitly requires a weight vector. You can now rewrite the entire perceptron algorithm so that it never refers explicitly to the weights and only ever depends on pairwise dot products between examples. This is shown in Algorithm 9.2.

The advantage to this "kernelized" algorithm is that you can perform feature expansions like the quadratic feature expansion from the introduction for "free." For example, for exactly the same cost as the quadratic features, you can use a **cubic feature map**, computed as $\phi(\ddot{\boldsymbol{x}})\phi(\boldsymbol{z}) = (1 + \boldsymbol{x} \cdot \boldsymbol{z})^3$, which corresponds to three-way interactions between variables. (And, in general, you can do so for any polynomial degree $p$ at the same computational complexity.)

## 9.3 *Kernelized K-means*

For a complete change of pace, consider the $K$-means algorithm from Section **??**. This algorithm is for *clustering* where there is no notion of "training labels." Instead, you want to partition the data into coherent clusters. For data in $\mathbb{R}^D$, it involves randomly initializing $K$-many

cluster means $\boldsymbol{\mu}^{(1)}, \ldots, \boldsymbol{\mu}^{(K)}$. The algorithm then alternates between the following two steps until convergence, with $\boldsymbol{x}$ replaced by $\phi(\boldsymbol{x})$ since that is the eventual goal:

1. For each example $n$, set cluster label $z_n = \arg\min_k \left|\left|\phi(\boldsymbol{x}_n) - \boldsymbol{\mu}^{(k)}\right|\right|^2$.
2. For each cluster $k$, update $\boldsymbol{\mu}^{(k)} = \frac{1}{N_k} \sum_{n:z_n=k} \phi(\boldsymbol{x}_n)$, where $N_k$ is the number of $n$ with $z_n = k$.

The question is whether you can perform these steps without explicitly computing $\phi(\boldsymbol{x}_n)$. The **representer theorem** is more straightforward here than in the perceptron. The mean of a set of data is, almost by definition, in the span of that data (choose the $a_i$s all to be equal to $1/N$). Thus, so long as you *initialize* the means in the span of the data, you are guaranteed always to have the means in the span of the data. Given this, you know that you can write each mean as an expansion of the data; say that $\boldsymbol{\mu}^{(k)} = \sum_n \alpha_n^{(k)} \phi(\boldsymbol{x}_n)$ for some parameters $\alpha_n^{(k)}$ (there are $N{\times}K$-many such parameters).

Given this expansion, in order to execute step (1), you need to compute norms. This can be done as follows:

$$z_n = \arg\min_k \left|\left|\phi(\boldsymbol{x}_n) - \boldsymbol{\mu}^{(k)}\right|\right|^2 \qquad \text{definition of } z_n \tag{9.8}$$

$$= \arg\min_k \left|\left|\phi(\boldsymbol{x}_n) - \sum_m \alpha_m^{(k)} \phi(\boldsymbol{x}_m)\right|\right|^2 \qquad \text{definition of } \boldsymbol{\mu}^{(k)} \tag{9.9}$$

$$= \arg\min_k ||\phi(\boldsymbol{x}_n)||^2 + \left|\left|\sum_m \alpha_m^{(k)} \phi(\boldsymbol{x}_m)\right|\right|^2 + \phi(\boldsymbol{x}_n) \cdot \left[\sum_m \alpha_m^{(k)} \phi(\boldsymbol{x}_m)\right] \qquad \text{expand quadratic term} \tag{9.10}$$

$$= \arg\min_k \sum_m \sum_{m'} \alpha_m^{(k)} \alpha_{m'}^{(k)} \phi(\boldsymbol{x}_m) \cdot \phi(\boldsymbol{x}_{m'}) + \sum_m \alpha_m^{(k)} \phi(\boldsymbol{x}_m) \cdot \phi(\boldsymbol{x}_n) + const \qquad \text{linearity and constant} \tag{9.11}$$

This computation can replace the assignments in step (1) of $K$-means. The mean updates are more direct in step (2):

$$\boldsymbol{\mu}^{(k)} = \frac{1}{N_k} \sum_{n:z_n=k} \phi(\boldsymbol{x}_n) \quad \Longleftrightarrow \quad \alpha_n^{(k)} = \begin{cases} \frac{1}{N_k} & \text{if } z_n = k \\ 0 & \text{otherwise} \end{cases} \tag{9.12}$$

## 9.4 What Makes a Kernel

A **kernel** is just a form of generalized dot product. You can also think of it as simply shorthand for $\phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{z})$, which is commonly written $K^{\phi}(\boldsymbol{x}, \boldsymbol{z})$. Or, when $\phi$ is clear from context, simply $K(\boldsymbol{x}, \boldsymbol{z})$.

This is often refered to as the kernel product between $\boldsymbol{x}$ and $\boldsymbol{z}$ (under the mapping $\phi$).

In this view, what you've seen in the preceding two sections is that you can rewrite both the perceptron algorithm and the $K$-means algorithm so that *they only ever depend on kernel products between data points, and never on the actual datapoints themselves*. This is a very powerful notion, as it has enabled the development of a large number of non-linear algorithms essentially "for free" (by applying the so-called **kernel trick**, that you've just seen twice).

This raises an interesting question. If you have rewritten these algorithms so that they only depend on the data through a function $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, can you stick *any* function $K$ in these algorithms, or are there some $K$ that are "forbidden?" In one sense, you "could" use any $K$, but the real question is: for what types of functions $K$ do these algorithms retain the properties that we expect them to have (like convergence, optimality, etc.)?

One way to answer this question is to say that $K(\cdot, \cdot)$ is a valid kernel if it corresponds to the inner product between two vectors. That is, $K$ is valid if there exists a function $\phi$ such that $K(\boldsymbol{x}, \boldsymbol{z}) = \phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{z})$. This is a direct definition and it should be clear that if $K$ satisfies this, then the algorithms go through as expected (because this is how we derived them).

You've already seen the general class of **polynomial kernels**, which have the form:

$$K_d^{(\text{poly})}(\boldsymbol{x}, \boldsymbol{z}) = \Big(1 + \boldsymbol{x} \cdot \boldsymbol{z}\Big)^d \tag{9.13}$$

where $d$ is a hyperparameter of the kernel. These kernels correspond to polynomial feature expansions.

There is an alternative characterization of a valid kernel function that is more mathematical. It states that $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a kernel if $K$ is **positive semi-definite** (or, in shorthand, **psd**). This property is also sometimes called **Mercer's condition**. In this context, this means the *for all* functions $f$ that are square integrable (i.e., $\int f(\boldsymbol{x})^2 \mathrm{d}\boldsymbol{x} < \infty$), other than the zero function, the following property holds:

$$\iint f(\boldsymbol{x}) K(\boldsymbol{x}, \boldsymbol{z}) f(\boldsymbol{z}) \mathrm{d}\boldsymbol{x} \mathrm{d}\boldsymbol{z} > 0 \tag{9.14}$$

This likely seems like it came out of nowhere. Unfortunately, the connection is well beyond the scope of this book, but is covered well is external sources. For now, simply take it as a given that this is an equivalent requirement. (For those so inclined, the appendix of this book gives a proof, but it requires a bit of knowledge of function spaces to understand.)

The question is: why is this alternative characterization useful? It is useful because it gives you an alternative way to construct kernel

functions. For instance, using it you can easily prove the following, which would be difficult from the definition of kernels as inner products after feature mappings.

**Theorem 12** (Kernel Addition). *If $K_1$ and $K_2$ are kernels, the $K$ defined by $K(\boldsymbol{x}, \boldsymbol{z}) = K_1(\boldsymbol{x}, \boldsymbol{z}) + K_2(\boldsymbol{x}, \boldsymbol{z})$ is also a kernel.*

*Proof of Theorem 12.* You need to verify the positive semi-definite property on $K$. You can do this as follows:

$$\iint f(\boldsymbol{x}) K(\boldsymbol{x}, \boldsymbol{z}) f(\boldsymbol{z}) \mathrm{d}\boldsymbol{x}\mathrm{d}\boldsymbol{z} = \iint f(\boldsymbol{x}) \left[K_1(\boldsymbol{x}, \boldsymbol{z}) + K_2(\boldsymbol{x}, \boldsymbol{z})\right] f(\boldsymbol{z}) \mathrm{d}\boldsymbol{x}\mathrm{d}\boldsymbol{z} \quad \text{definition of } K \tag{9.15}$$

$$= \iint f(\boldsymbol{x}) K_1(\boldsymbol{x}, \boldsymbol{z}) f(\boldsymbol{z}) \mathrm{d}\boldsymbol{x}\mathrm{d}\boldsymbol{z} + \iint f(\boldsymbol{x}) K_2(\boldsymbol{x}, \boldsymbol{z}) f(\boldsymbol{z}) \mathrm{d}\boldsymbol{x}\mathrm{d}\boldsymbol{z} \quad \text{distributive rule} \tag{9.16}$$

$$> 0 + 0 \quad K_1 \text{ and } K_2 \text{ are psd} \tag{9.17}$$

□

More generally, any positive linear combination of kernels is still a kernel. Specifically, if $K_1, \ldots, K_M$ are all kernels, and $\alpha_1, \ldots, \alpha_M \geq 0$, then $K(\boldsymbol{x}, \boldsymbol{z}) = \sum_m \alpha_m K_m(\boldsymbol{x}, \boldsymbol{z})$ is also a kernel.

You can also use this property to show that the following **Gaussian kernel** (also called the **RBF kernel**) is also psd:

$$K_\gamma^{(\mathrm{RBF})}(\boldsymbol{x}, \boldsymbol{z}) = \exp\left[-\gamma \left|\left|\boldsymbol{x} - \boldsymbol{z}\right|\right|^2\right] \tag{9.18}$$

Here $\gamma$ is a hyperparameter that controls the width of this Gaussian-like bumps. To gain an intuition for what the RBF kernel is doing, consider what prediction looks like in the perceptron:

$$f(\boldsymbol{x}) = \sum_n \alpha_n K(\boldsymbol{x}_n, \boldsymbol{x}) + b \tag{9.19}$$

$$= \sum_n \alpha_n \exp\left[-\gamma \left|\left|\boldsymbol{x}_n - \boldsymbol{z}\right|\right|^2\right] \tag{9.20}$$

In this computation, each training example is getting to "vote" on the label of the test point $\boldsymbol{x}$. The amount of "vote" that the $n$th training example gets is proportional to the negative exponential of the distance between the test point and itself. This is very much like an RBF neural network, in which there is a Gaussian "bump" at each training example, with variance $1/(2\gamma)$, and where the $\alpha_n$s act as the weights connecting these RBF bumps to the output.

Showing that this kernel is positive definite is a bit of an exercise in analysis (particularly, integration by parts), but otherwise not difficult. Again, the proof is provided in the appendix.

So far, you have seen two bsaic classes of kernels: polynomial kernels ($K(\boldsymbol{x}, vz) = (1 + \boldsymbol{x} \cdot \boldsymbol{z})^d$), which includes the linear kernel ($K(\boldsymbol{x}, \boldsymbol{z}) = \boldsymbol{x} \cdot \boldsymbol{z}$) and RBF kernels ($K(\boldsymbol{x}, \boldsymbol{z}) = \exp[-\gamma ||\boldsymbol{x} - \boldsymbol{z}||^2]$). The former have a direct connection to feature expansion; the latter to RBF networks. You also know how to combine kernels to get new kernels by addition. In fact, you can do more than that: the product of two kernels is also a kernel.

As far as a "library of kernels" goes, there are many. Polynomial and RBF are by far the most popular. A commonly used, but technically *invalid* kernel, is the hyperbolic-tangent kernel, which mimics the behavior of a two-layer neural network. It is defined as:

$$K^{(\tanh)} = \tanh(1 + \boldsymbol{x} \cdot \boldsymbol{z}) \qquad \text{Warning: not psd} \qquad (9.21)$$

A final example, which is not very common, but is nonetheless interesting, is the all-subsets kernel. Suppose that your $D$ features are all *binary*: all take values 0 or 1. Let $A \subseteq \{1, 2, \dots D\}$ be a subset of features, and let $f_A(\boldsymbol{x}) = \bigwedge_{d \in A} x_d$ be the conjunction of all the features in $A$. Let $\phi(\boldsymbol{x})$ be a feature vector over *all* such $A$s, so that there are $2^D$ features in the vector $\phi$. You can compute the kernel associated with this feature mapping as:

$$K^{(\text{subs})}(\boldsymbol{x}, \boldsymbol{z}) = \prod_d \Big(1 + x_d z_d\Big) \qquad (9.22)$$

Verifying the relationship between this kernel and the all-subsets feature mapping is left as an exercise (but closely resembles the expansion for the quadratic kernel).

## 9.5 Support Vector Machines

Kernelization predated support vector machines, but SVMs are definitely the model that popularized the idea. Recall the definition of the soft-margin SVM from Chapter 6.7 and in particular the optimization problem (6.36), which attempts to balance a large margin (small $||\boldsymbol{w}||^2$) with a small loss (small $\xi_n$s, where $\xi_n$ is the **slack** on the $n$th training example). This problem is repeated below:

$$\begin{aligned} \min_{\boldsymbol{w}, b, \boldsymbol{\xi}} \quad & \frac{1}{2} ||\boldsymbol{w}||^2 + C \sum_n \xi_n & (9.23) \\ \text{subj. to} \quad & y_n (\boldsymbol{w} \cdot \boldsymbol{x}_n + b) \geq 1 - \xi_n & (\forall n) \\ & \xi_n \geq 0 & (\forall n) \end{aligned}$$

Previously, you optimized this by explicitly computing the slack variables $\xi_n$, given a solution to the decision boundary, $\boldsymbol{w}$ and $b$. However, you are now an expert with using Lagrange multipliers

to optimize constrained problems! The overall *goal* is going to be to rewrite the SVM optimization problem in a way that it no longer explicitly depends on the weights $\boldsymbol{w}$ and only depends on the examples $\boldsymbol{x}_n$ through kernel products.

There are $2N$ constraints in this optimization, one for each slack constraint and one for the requirement that the slacks are non-negative. Unlike the last time, these constraints are now *inequalities*, which require a slightly different solution. First, you rewrite all the inequalities so that they read as *something* $\geq 0$ and then add corresponding Lagrange multipliers. The main difference is that the Lagrange multipliers are now constrained to be non-negative, and their sign in the augmented objective function matters.

The second set of constraints is already in the proper form; the first set can be rewritten as $y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) - 1 + \xi_n \geq 0$. You're now ready to construct the Lagrangian, using multipliers $\alpha_n$ for the first set of constraints and $\beta_n$ for the second set.

$$\mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \frac{1}{2}||\boldsymbol{w}||^2 + C\sum_n \xi_n - \sum_n \beta_n \xi_n \tag{9.24}$$

$$- \sum_n \alpha_n \left[ y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) - 1 + \xi_n \right] \tag{9.25}$$

The *new* optimization problem is:

$$\min_{\boldsymbol{w},b,\boldsymbol{\xi}} \max_{\boldsymbol{\alpha}\geq 0} \max_{\boldsymbol{\beta}\geq 0} \mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) \tag{9.26}$$

The intuition is exactly the same as before. If you are able to find a solution that satisfies the constraints (eg., the purple term is properly non-negative), then the $\beta_n$s cannot do anything to "hurt" the solution. On the other hand, if the purple term *is* negative, then the corresponding $\beta_n$ can go to $+\infty$, breaking the solution.

You can solve this problem by taking gradients. This is a bit tedious, but and important step to realize how everything fits together. Since your goal is to remove the dependence on $\boldsymbol{w}$, the first step is to take a gradient with respect to $\boldsymbol{w}$, set it equal to zero, and solve for $\boldsymbol{w}$ in terms of the other variables.

$$\nabla_{\boldsymbol{w}} \mathcal{L} = \boldsymbol{w} - \sum_n \alpha_n y_n \boldsymbol{x}_n = 0 \quad \iff \quad \boldsymbol{w} = \sum_n \alpha_n y_n \boldsymbol{x}_n \tag{9.27}$$

At this point, you should immediately recognize a similarity to the kernelized perceptron: the optimal weight vector takes *exactly* the same form in both algorithms.

You can now take this new expression for $\boldsymbol{w}$ and plug it back in to the expression for $\mathcal{L}$, thus removing $\boldsymbol{w}$ from consideration. To avoid subscript overloading, you should replace the $n$ in the expression for

$\boldsymbol{w}$ with, say, $m$. This yields:

$$\mathcal{L}(b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \frac{1}{2}\left|\left|\sum_m \alpha_m y_m \boldsymbol{x}_m\right|\right|^2 + C\sum_n \xi_n - \sum_n \beta_n \xi_n \tag{9.28}$$

$$- \sum_n \alpha_n \left[ y_n \left( \left[ \sum_m \alpha_m y_m \boldsymbol{x}_m \right] \cdot \boldsymbol{x}_n + b \right) - 1 + \xi_n \right] \tag{9.29}$$

At this point, it's convenient to rewrite these terms; be sure you understand where the following comes from:

$$\mathcal{L}(b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \frac{1}{2}\sum_n \sum_m \alpha_n \alpha_m y_n y_m \boldsymbol{x}_n \cdot \boldsymbol{x}_m + \sum_n (C - \beta_n)\xi_n \tag{9.30}$$

$$- \sum_n \sum_m \alpha_n \alpha_m y_n y_m \boldsymbol{x}_n \cdot \boldsymbol{x}_m - \sum_n \alpha_n \left( y_n b - 1 + \xi_n \right) \tag{9.31}$$

$$= -\frac{1}{2}\sum_n \sum_m \alpha_n \alpha_m y_n y_m \boldsymbol{x}_n \cdot \boldsymbol{x}_m + \sum_n (C - \beta_n)\xi_n \tag{9.32}$$

$$-b\sum_n \alpha_n y_n - \sum_n \alpha_n (\xi_n - 1) \tag{9.33}$$

Things are starting to look good: you've successfully removed the dependence on $\boldsymbol{w}$, *and* everything is now written in terms of dot products between input vectors! This might still be a difficult problem to solve, so you need to continue and attempt to remove the remaining variables $b$ and $\boldsymbol{\xi}$.

The derivative with respect to $b$ is:

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_n \alpha_n y_n = 0 \tag{9.34}$$

This doesn't allow you to *substitute* $b$ with something (as you did with $\boldsymbol{w}$), but it does mean that the fourth term ($b\sum_n \alpha_n y_n$) goes to zero at the optimum.

The last of the original variables is $\xi_n$; the derivatives in this case look like:

$$\frac{\partial \mathcal{L}}{\partial \xi_n} = C - \beta_n - \alpha_n \quad \Longleftrightarrow \quad C - \beta_n = \alpha_n \tag{9.35}$$

Again, this doesn't allow you to substitute, but it does mean that you can rewrite the second term, which as $\sum_n (C - \beta_n)\xi_n$ as $\sum_n \alpha_n \xi_n$. This then cancels with (most of) the final term. However, you need to be careful to remember something. When we optimize, both $\alpha_n$ and $\beta_n$ are constrained to be non-negative. What this means is that since we are dropping $\beta$ from the optimization, we need to ensure that $\alpha_n \leq C$, otherwise the corresponding $\beta$ will need to be negative, which is not

allowed. You finally wind up with the following, where $\boldsymbol{x}_n \cdot \boldsymbol{x}_m$ has been replaced by $K(\boldsymbol{x}_n, \boldsymbol{x}_m)$:

$$\mathcal{L}(\boldsymbol{\alpha}) = \sum_n \alpha_n - \frac{1}{2} \sum_n \sum_m \alpha_n \alpha_m y_n y_m K(\boldsymbol{x}_n, \boldsymbol{x}_m) \tag{9.36}$$

If you are comfortable with matrix notation, this has a very compact form. Let $\mathbf{1}$ denote the $N$-dimensional vector of all 1s, let $\boldsymbol{y}$ denote the vector of labels and let $\mathbf{G}$ be the $N \times N$ matrix, where $\mathbf{G}_{n,m} = y_n y_m K(\boldsymbol{x}_n, \boldsymbol{x}_m)$, then this has the following form:

$$\mathcal{L}(\boldsymbol{\alpha}) = \boldsymbol{\alpha}^\top \mathbf{1} - \frac{1}{2} \boldsymbol{\alpha}^\top \mathbf{G} \boldsymbol{\alpha} \tag{9.37}$$

The resulting optimization problem is to *maximize* $\mathcal{L}(\boldsymbol{\alpha})$ as a function of $\boldsymbol{\alpha}$, subject to the constraint that the $\alpha_n$s are all non-negative and less than $C$ (because of the constraint added when removing the $\beta$ variables). Thus, your problem is:

$$\min_{\boldsymbol{\alpha}} \quad -\mathcal{L}(\boldsymbol{\alpha}) = \frac{1}{2} \sum_n \sum_m \alpha_n \alpha_m y_n y_m K(\boldsymbol{x}_n, \boldsymbol{x}_m) - \sum_n \alpha_n \tag{9.38}$$

$$\text{subj. to} \quad 0 \leq \alpha_n \leq C \qquad (\forall n)$$

One way to solve this problem is gradient descent on $\alpha$. The only complication is making sure that the $\alpha$s satisfy the constraints. In this case, you can use a **projected gradient** algorithm: after each gradient update, you adjust your parameters to satisfy the constraints by *projecting* them into the feasible region. In this case, the projection is trivial: if, after a gradient step, any $\alpha_n < 0$, simply set it to 0; if any $\alpha_n > C$, set it to $C$.

## 9.6 Understanding Support Vector Machines

The prior discussion involved quite a bit of math to derive a representation of the support vector machine in terms of the Lagrange variables. This mapping is actually sufficiently standard that everything in it has a name. The original problem variables $(\boldsymbol{w}, b, \boldsymbol{\xi})$ are called the **primal variables**; the Lagrange variables are called the **dual variables**. The optimization problem that results after removing all of the primal variables is called the **dual problem**.

A succinct way of saying what you've done is: you found that after converting the SVM into its dual, it is possible to kernelize.

To understand SVMs, a first step is to peek into the dual formulation, Eq (9.38). The objective has two terms: the first depends on the data, and the second depends only on the dual variables. The first thing to notice is that, because of the second term, the $\alpha$s "want" to

get as large as possible. The constraint ensures that they cannot exceed $C$, which means that the general tendency is for the $\alpha$s to grow as close to $C$ as possible.

To further understand the dual optimization problem, it is useful to think of the kernel as being a measure of *similarity* between two data points. This analogy is most clear in the case of RBF kernels, but even in the case of linear kernels, if your examples all have unit norm, then their dot product is still a measure of similarity. Since you can write the prediction function as $f(\hat{\boldsymbol{x}}) = \text{sign}(\sum_n \alpha_n y_n K(\boldsymbol{x}_n, \hat{\boldsymbol{x}}))$, it is natural to think of $\alpha_n$ as the "importance" of training example $n$, where $\alpha_n = 0$ means that it is not used at all at test time.

Consider two data points that have the same label; namely, $y_n = y_m$. This means that $y_n y_m = +1$ and the objective function has a term that looks like $\alpha_n \alpha_m K(\boldsymbol{x}_n, \boldsymbol{x}_m)$. Since the goal is to make this term small, then one of two things has to happen: either $K$ has to be small, or $\alpha_n \alpha_m$ has to be small. If $K$ is already small, then this doesn't affect the setting of the corresponding $\alpha$s. But if $K$ is large, then this *strongly* encourages at least one of $\alpha_n$ or $\alpha_m$ to go to zero. So if you have two data points that are very similar *and* have the same label, at least one of the corresponding $\alpha$s will be small. This makes intuitive sense: if you have two data points that are basically the same (both in the $\boldsymbol{x}$ and $y$ sense) then you only need to "keep" one of them around.

Suppose that you have two data points with different labels: $y_n y_m = -1$. Again, if $K(\boldsymbol{x}_n, \boldsymbol{x}_m)$ is small, nothing happens. But if it is large, then the corresponding $\alpha$s are encouraged to be as large as possible. In other words, if you have two similar examples with different labels, you are strongly encouraged to keep the corresponding $\alpha$s as large as $C$.

An alternative way of understanding the SVM dual problem is geometrically. Remember that the whole point of introducing the variable $\alpha_n$ was to ensure that the $n$th training example was correctly classified, modulo slack. More formally, the goal of $\alpha_n$ is to ensure that $y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) - 1 + \xi_n \geq 0$. Suppose that this constraint it *not* satisfied. There is an important result in optimization theory, called the **Karush-Kuhn-Tucker conditions** (or **KKT conditions**, for short) that states that at the optimum, the product of the Lagrange multiplier for a constraint, and the value of that constraint, will equal zero. In this case, this says that at the optimum, you have:

$$\alpha_n \Big[ y_n \left( \boldsymbol{w} \cdot \boldsymbol{x}_n + b \right) - 1 + \xi_n \Big] = 0 \tag{9.39}$$

In order for this to be true, it means that (at least) one of the following must be true:

$$\alpha_n = 0 \qquad or \qquad y_n \left( \boldsymbol{w} \cdot \boldsymbol{x}_n + b \right) - 1 + \xi_n = 0 \tag{9.40}$$

A reasonable question to ask is: under what circumstances will $\alpha_n$ be *non-zero?* From the KKT conditions, you can discern that $\alpha_n$ can be non-zero *only* when the constraint holds *exactly*; namely, that $y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) - 1 + \xi_n = 0$. When does that constraint hold *exactly*? It holds exactly only for those points *precisely* on the margin of the hyperplane.

In other words, the *only* training examples for which $\alpha_n \neq 0$ are those that lie precisely 1 unit away from the maximum margin decision boundary! (Or those that are "moved" there by the corresponding slack.) These points are called the **support vectors** because they "support" the decision boundary. In general, the number of support vectors is far smaller than the number of training examples, and therefore you naturally end up with a solution that only uses a subset of the training data.

From the first discussion, you know that the points that wind up being support vectors are exactly those that are "confusable" in the sense that you have to examples that are nearby, but have different labels. This is a completely in line with the previous discussion. If you have a decision boundary, it will pass between these "confusable" points, and therefore they will end up being part of the set of support vectors.

## 9.7 Exercises

**Exercise 9.1. TODO...**

# 10 | Learning Theory

> For nothing ought to be posited without a reason given, unless it is self-evident or known by experience or proved by the authority of Sacred Scripture. – William of Occam, c. 1320

**Learning Objectives:**

- Explain why inductive bias is necessary.
- Define the PAC model and explain why both the "P" and "A" are necessary.
- Explain the relationship between complexity measures and regularizers.
- Identify the role of complexity in generalization.
- Formalize the relationship between margins and complexity.

Dependencies:

By now, you are an expert at building learning algorithms. You probably understand how they work, intuitively. And you understand why they should generalize. However, there are several basic questions you might want to know the answer to. Is learning always possible? How many training examples will I need to do a good job learning? Is my test performance going to be much worse than my training performance? The key idea that underlies all these answer is that *simple functions generalize well.*

The amazing thing is that you can actually prove strong results that address the above questions. In this chapter, you will learn some of the most important results in learning theory that attempt to answer these questions. The goal of this chapter is not theory for theory's sake, but rather as a way to better understand why learning models work, and how to use this theory to build better algorithms. As a concrete example, we will see how 2-norm regularization provably leads to better generalization performance, thus justifying our common practice!

## 10.1 The Role of Theory

In contrast to the quote at the start of this chapter, a practitioner friend once said "I would happily give up a few percent performance for an algorithm that I can understand." Both perspectives are completely valid, and are actually not contradictory. The second statement is presupposing that theory helps you understand, which hopefully you'll find to be the case in this chapter.

Theory can serve two roles. It can justify and help understand why common practice works. This is the "theory after" view. It can also serve to suggest new algorithms and approaches that turn out to work well in practice. This is the "theory before" view. Often, it turns out to be a mix. Practitioners discover something that works surprisingly well. Theorists figure out why it works and prove something about it. And in the process, they make it better or find new algo-

rithms that more directly exploit whatever property it is that made the theory go through.

Theory can also help you understand what's possible and *what's not possible*. One of the first things we'll see is that, in general, machine learning can not work. Of course it *does* work, so this means that we need to think harder about what it means for learning algorithms to work. By understanding what's not possible, you can focus our energy on things that are.

Probably the biggest *practical* success story for theoretical machine learning is the theory of **boosting**, which you won't actually see in this chapter. (You'll have to wait for Chapter 11.) Boosting is a very simple style of algorithm that came out of theoretical machine learning, and has proven to be incredibly successful in practice. So much so that it is one of the de facto algorithms to run when someone gives you a new data set. In fact, in 2004, Yoav Freund and Rob Schapire won the ACM's Paris Kanellakis Award for their boosting algorithm AdaBoost. This award is given for theoretical accomplishments that have had a significant and demonstrable effect on the practice of computing.[1]

[1] In 2008, Corinna Cortes and Vladimir Vapnik won it for support vector machines.

## 10.2 Induction is Impossible

One nice thing about theory is that it forces you to be *precise* about what you are trying to do. You've already seen a formal definition of binary classification in Chapter 5. But let's take a step back and re-analyze what it means to learn to do binary classification.

From an *algorithmic* perspective, a natural question is whether there is an "ultimate" learning algorithm, $\mathcal{A}^{\text{awesome}}$, that solves the Binary Classification problem above. In other words, have you been wasting your time learning about *K*NN and Perceptron and decision trees, when $\mathcal{A}^{\text{awesome}}$ is out there.

What would such an ultimate learning algorithm do? You would like it to take in a data set $D$ and produce a function $f$. No matter what $D$ looks like, this function $f$ should get perfect classification on all future examples drawn from the same distribution that produced $D$.

A little bit of introspection should demonstrate that this is impossible. For instance, there might be label noise in our distribution. As a very simple example, let $\mathcal{X} = \{-1, +1\}$ (i.e., a one-dimensional, binary distribution. Define the data distribution as:

$$\mathcal{D}(\langle +1 \rangle, +1) = 0.4 \qquad \mathcal{D}(\langle -1 \rangle, -1) = 0.4 \tag{10.1}$$

$$\mathcal{D}(\langle +1 \rangle, -1) = 0.1 \qquad \mathcal{D}(\langle -1 \rangle, +1) = 0.1 \tag{10.2}$$

In other words, 80% of data points in this distrubtion have $x = y$

and 20% don't. No matter what function your learning algorithm produces, there's no way that it can do better than 20% error on this data.

Given this, it seems hopeless to have an algorithm $\mathcal{A}^{\text{awesome}}$ that always achieves an error rate of zero. The best that we can hope is that the error rate is not "too large."

Unfortunately, simply weakening our requirement on the error rate is not enough to make learning possible. The second source of difficulty comes from the fact that the only access we have to the data distribution is through sampling. In particular, when trying to learn about a distribution like that in 10.1, you only get to see data points *drawn* from that distribution. You know that "eventually" you will see enough data points that your sample is representative of the distribution, but it might not happen immediately. For instance, even though a fair coin will come up heads only with probability 1/2, it's completely plausible that in a sequence of four coin flips you never see a tails, or perhaps only see one tails.

So the second thing that we have to give up is the hope that $\mathcal{A}^{\text{awesome}}$ will *always* work. In particular, if we happen to get a lousy sample of data from $\mathcal{D}$, we need to allow $\mathcal{A}^{\text{awesome}}$ to do something completely unreasonable.

Thus, we cannot hope that $\mathcal{A}^{\text{awesome}}$ will do perfectly, every time. We cannot even hope that it will do pretty well, all of the time. Nor can we hope that it will do perfectly, most of the time. The best *best* we can reasonably hope of $\mathcal{A}^{\text{awesome}}$ is that it it will do pretty well, most of the time.

? It's clear that if your algorithm produces a *deterministic* function that it cannot do better than 20% error. What if it produces a stochastic (aka randomized) function?

## 10.3 Probably Approximately Correct Learning

**Probably Approximately Correct** (**PAC**) learning is a formalism of inductive learning based on the realization that the best we can hope of an algorithm is that it does a good job (i.e., is approximately correct), most of the time (i.e., it is *probably* appoximately correct).[2]

Consider a hypothetical learning algorithm. You run it on ten different binary classification data sets. For each one, it comes back with functions $f_1, f_2, \ldots, f_{10}$. For some reason, whenever you run $f_4$ on a test point, it crashes your computer. For the other learned functions, their performance on test data is always at most 5% error. If this situtation is guaranteed to happen, then this hypothetical learning algorithm is a PAC learning algorithm. It satisfies "probably" because it only failed in one out of ten cases, and it's "approximate" because it achieved low, but non-zero, error on the remainder of the cases.

This leads to the formal definition of an $(\epsilon, \delta)$ PAC-learning algorithm. In this definition, $\epsilon$ plays the role of measuring accuracy (in

[2] Leslie Valiant invented the notion of PAC learning in 1984. In 2011, he received the Turing Award, the highest honor in computing for his work in learning theory, computational complexity and parallel systems.

the previous example, $\epsilon = 0.05$) and $\delta$ plays the role of measuring failure (in the previous, $\delta = 0.1$).

**Definitions 1.** *An algorithm $\mathcal{A}$ is an $(\epsilon, \delta)$-PAC learning algorithm if,* for all *distributions $\mathcal{D}$: given samples from $\mathcal{D}$, the probability that it returns a "bad function" is at most $\delta$; where a "bad" function is one with test error rate more than $\epsilon$ on $\mathcal{D}$.*

There are two notions of *efficiency* that matter in PAC learning. The first is the usual notion of *computational complexity*. You would prefer an algorithm that runs quickly to one that takes forever. The second is the notion of **sample complexity**: the number of examples required for your algorithm to achieve its goals. Note that the goal of both of these measure of complexity is to bound how much of a scarse resource your algorithm uses. In the computational case, the resource is CPU cycles. In the sample case, the resource is labeled examples.

**Definition:** An algorithm $\mathcal{A}$ is an efficient $(\epsilon, \delta)$-PAC learning algorithm if it is an $(\epsilon, \delta)$-PAC learning algorithm whose runtime is polynomial in $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$.

In other words, suppose that you want your algorithm to achieve 4% error rate rather than 5%. The runtime required to do so should no go up by an exponential factor.

## 10.4 PAC Learning of Conjunctions

To get a better sense of PAC learning, we will start with a completely irrelevant and uninteresting example. The purpose of this example is *only* to help understand how PAC learning works.

The setting is learning conjunctions. Your data points are binary vectors, for instance $\boldsymbol{x} = \langle 0, 1, 1, 0, 1 \rangle$. Someone guarantees for you that there is some boolean conjunction that defines the true labeling of this data. For instance, $x_1 \wedge \neg x_2 \wedge x_5$ ("or" is not allowed). In formal terms, we often call the true underlying classification function the **concept**. So this is saying that the concept you are trying to learn is a conjunction. In this case, the boolean function would assign a negative label to the example above.

Since you know that the concept you are trying to learn is a conjunction, it makes sense that you would represent your function as a conjunction as well. For historical reasons, the function that you learn is often called a **hypothesis** and is often denoted $h$. However, in keeping with the other notation in this book, we will continue to denote it $f$.

Formally, the set up is as follows. There is some distribution $\mathcal{D}^X$ over binary data points (vectors) $\boldsymbol{x} = \langle x_1, x_2, \ldots, x_D \rangle$. There is a fixed

concept conjunction $c$ that we are trying to learn. There is no noise, so for any example $\boldsymbol{x}$, its true label is simply $y = c(\boldsymbol{x})$.

What is a reasonable algorithm in this case? Suppose that you observe the example in Table 10.1. From the first example, we know that the true formula cannot include the term $x_1$. If it did, this example would have to be negative, which it is not. By the same reasoning, it cannot include $x_2$. By analogous reasoning, it also can neither include the term $\neg x_3$ nor the term $\neg x_4$.

| $y$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ |
|---|---|---|---|---|
| +1 | 0 | 0 | 1 | 1 |
| +1 | 0 | 1 | 1 | 1 |
| -1 | 1 | 1 | 0 | 1 |

Table 10.1: Data set for learning conjunctions.

This suggests the algorithm in Algorithm 10.4, colloquially the "Throw Out Bad Terms" algorithm. In this algorith, you begin with a function that includes all possible $2D$ terms. Note that this function will initially classify everything as negative. You then process each example in sequence. On a negative example, you do nothing. On a positive example, you throw out terms from $f$ that contradict the given positive example.

? Verify that Algorithm 10.4 maintains an *invariant* that it always errs on the side of classifying examples negative and never errs the other way.

If you run this algorithm on the data in Table 10.1, the sequence of $f$s that you cycle through are:

$$f^0(\boldsymbol{x}) = x_1 \wedge \neg x_1 \wedge x_2 \wedge \neg x_2 \wedge x_3 \wedge \neg x_3 \wedge x_4 \wedge \neg x_4 \tag{10.3}$$
$$f^1(\boldsymbol{x}) = \neg x_1 \wedge \neg x_2 \wedge x_3 \wedge x_4 \tag{10.4}$$
$$f^2(\boldsymbol{x}) = \neg x_1 \wedge x_3 \wedge x_4 \tag{10.5}$$
$$f^3(\boldsymbol{x}) = \neg x_1 \wedge x_3 \wedge x_4 \tag{10.6}$$

The first thing to notice about this algorithm is that after processing an example, it is guaranteed to classify that example correctly. This observation *requires* that there is no noise in the data.

The second thing to notice is that it's very computationally efficient. Given a data set of $N$ examples in $D$ dimensions, it takes $\mathcal{O}(ND)$ time to process the data. This is linear in the size of the data set.

However, in order to be an efficient $(\epsilon, \delta)$-PAC learning algorithm, you need to be able to get a bound on the **sample complexity** of this algorithm. Sure, you know that its run time is linear in the number of example $N$. But *how many* examples $N$ do you need to see in order to guarantee that it achieves an error rate of at most $\epsilon$ (in all but $\delta$-many cases)? Perhaps $N$ has to be gigantic (like $2^{2^{D/\epsilon}}$) to (probably) guarantee a small error.

The goal is to prove that the number of samples $N$ required to (probably) achieve a small error is not-too-big. The general proof technique for this has essentially the same flavor as almost every PAC learning proof around. First, you define a "bad thing." In this case, a "bad thing" is that there is some term (say $\neg x_8$) that should have been thrown out, but wasn't. Then you say: well, bad things happen. Then you notice that if this bad thing happened, you must not have

**Algorithm 30** BINARYCONJUNCTIONTRAIN(**D**)

```
f ← x_1 ∧ ¬x_1 ∧ x_2 ∧ ¬x_2 ∧ ··· ∧ x_D ∧ ¬x_D          // initialize function
for all positive examples (x,+1) in D do
  for d = 1 ... D do
    if x_d = 0 then
      f ← f without term "x_d"
    else
      f ← f without term "¬x_d"
    end if
  end for
end for
return f
```

seen any positive training examples with $x_8 = 0$. So example with $x_8 = 0$ must have low probability (otherwise you would have seen them). So bad things must not be that common.

**Theorem 13.** *With probability at least $(1 - \delta)$: Algorithm 10.4 requires at most $N = \ldots$ examples to achieve an error rate $\leq \epsilon$.*

*Proof of Theorem 13.* Let $c$ be the concept you are trying to learn and let $\mathcal{D}$ be the distribution that generates the data.

A learned function $f$ can make a mistake if it contains *any* term $t$ that is not in $c$. There are initially $2D$ many terms in $f$, and any (or all!) of them might not be in $c$. We want to ensure that the probability that $f$ makes an error is at most $\epsilon$. It is sufficient to ensure that

For a term $t$ (eg., $\neg x_5$), we say that $t$ "negates" an example $\boldsymbol{x}$ if $t(\boldsymbol{x}) = 0$. Call a term $t$ "bad" if (a) it does not appear in $c$ and (b) has probability at least $\epsilon/2D$ of appearing (with respect to the unknown distribution $\mathcal{D}$ over data points).

First, we show that if we have no bad terms left in $f$, then $f$ has an error rate at most $\epsilon$.

We know that $f$ contains *at most* $2D$ terms, since is begins with $2D$ terms and throws them out.

The algorithm begins with $2D$ terms (one for each variable and one for each negated variable). Note that $f$ will only make one type of error: it can call positive examples negative, but can never call a negative example positive. Let $c$ be the true concept (true boolean formula) and call a term "bad" if it does not appear in $c$. A specific bad term (eg., $\neg x_5$) will cause $f$ to err only on positive examples that contain a corresponding bad value (eg., $x_5 = 1$). TODO... finish this □

What we've shown in this theorem is that: *if* the true underlying concept is a boolean conjunction, *and* there is no noise, *then* the "Throw Out Bad Terms" algorithm needs $N \leq \ldots$ examples in order

to learn a boolean conjunction that is $(1-\delta)$-likely to achieve an error of at most $\epsilon$. That is to say, that the **sample complexity** of "Throw Out Bad Terms" is .... Moreover, since the algorithm's runtime is linear in $N$, it is an efficient PAC learning algorithm.

## 10.5 Occam's Razor: Simple Solutions Generalize

The previous example of boolean conjunctions is mostly just a warm-up exercise to understand PAC-style proofs in a concrete setting. In this section, you get to generalize the above argument to a much larger range of learning problems. We will still assume that there is no noise, because it makes the analysis much simpler. (Don't worry: noise will be added eventually.)

William of Occam (c. 1288 – c. 1348) was an English friar and philosopher is is most famous for what later became known as Occam's razor and popularized by Bertrand Russell. The principle basically states that you should only assume as much as you need. Or, more verbosely, "if one can explain a phenomenon without assuming this or that hypothetical entity, then there is no ground for assuming it i.e. that one should always opt for an explanation in terms of the fewest possible number of causes, factors, or variables." What Occam actually wrote is the quote that began this chapter.

In a machine learning context, a reasonable paraphrase is "simple solutions generalize well." In other words, you have $10,000$ features you could be looking at. If you're able to explain your predictions using just 5 of them, or using all $10,000$ of them, then you should just use the 5.

The Occam's razor theorem states that this is a good idea, theoretically. It essentially states that if you are learning some unknown concept, and if you are able to fit your training data perfectly, but you don't need to resort to a huge class of possible functions to do so, then your learned function will generalize well. It's an amazing theorem, due partly to the simplicity of its proof. In some ways, the proof is actually *easier* than the proof of the boolean conjunctions, though it follows the same basic argument.

In order to state the theorem explicitly, you need to be able to think about a **hypothesis class**. This is the set of possible hypotheses that your algorithm searches through to find the "best" one. In the case of the boolean conjunctions example, the hypothesis class, $\mathcal{H}$, is the set of all boolean formulae over $D$-many variables. In the case of a perceptron, your hypothesis class is the set of all possible linear classifiers. The hypothesis class for boolean conjunctions is *finite*; the hypothesis class for linear classifiers is *infinite*. For Occam's razor, we can only work with finite hypothesis classes.

**Theorem 14** (Occam's Bound). *Suppose $\mathcal{A}$ is an algorithm that learns a function $f$ from some* finite *hypothesis class $\mathcal{H}$. Suppose the learned function always gets zero error on the training data. Then, the sample complexity of $f$ is at most* $\log |\mathcal{H}|$.

TODO COMMENTS

*Proof of Theorem 14.* TODO □

This theorem applies directly to the "Throw Out Bad Terms" algorithm, since (a) the hypothesis class is finite and (b) the learned function always achieves zero error on the training data. To apply Occam's Bound, you need only compute the size of the hypothesis class $\mathcal{H}$ of boolean conjunctions. You can compute this by noticing that there are a total of $2D$ possible terms in any formula in $\mathcal{H}$. Moreover, each term may or may not be in a formula. So there are $2^{2D} = 4^D$ possible formulae; thus, $|\mathcal{H}| = 4^D$. Applying Occam's Bound, we see that the sample complexity of this algorithm is $N \leq \ldots$.

Of course, Occam's Bound is general enough to capture other learning algorithms as well. In particular, it can capture decision trees! In the no-noise setting, a decision tree will always fit the training data perfectly. The only remaining difficulty is to compute the size of the hypothesis class of a decision tree learner.

For simplicity's sake, suppose that our decision tree algorithm always learns complete trees: i.e., every branch from root to leaf is length $D$. So the number of split points in the tree (i.e., places where a feature is queried) is $2^{D-1}$. (See Figure 10.1.) Each split point needs to be assigned a feature: there $D$-many choices here. This gives $D2^{D-1}$ trees. The last thing is that there are $2^D$ leaves of the tree, each of which can take two possible values, depending on whether this leaf is classified as $+1$ or $-1$: this is $2{\times}2^D = 2^{D+1}$ possibilities. Putting this all togeter gives a total number of trees $|\mathcal{H}| = D2^{D-1}2^{D+1} = D2^{2D} = D4^D$. Applying Occam's Bound, we see that *TODO* examples is enough to learn a decision tree!

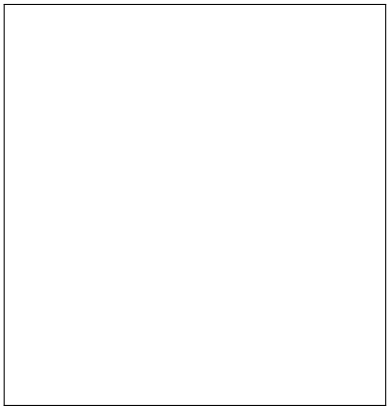


Figure 10.1: `thy:dt`: picture of full decision tree

## 10.6 Complexity of Infinite Hypothesis Spaces

Occam's Bound is a fantastic result for learning over *finite* hypothesis spaces. Unfortunately, it is completely useless when $|\mathcal{H}| = \infty$. This is because the proof works by using each of the $N$ training examples to "throw out" bad hypotheses until only a small number are left. But if $|\mathcal{H}| = \infty$, and you're throwing out a finite number at each step, there will always be an infinite number remaining.

This means that, if you want to establish sample complexity results for infinite hypothesis spaces, you need some new way of measuring

their "size" or "complexity." A prototypical way of doing this is to measure the complexity of a hypothesis class as the *number of different things it can do*.

As a silly example, consider boolean conjunctions again. Your input is a vector of binary features. However, instead of representing your hypothesis as a boolean conjunction, you choose to represent it as a conjunction of *inequalities*. That is, instead of writing $x_1 \wedge \neg x_2 \wedge x_5$, you write $[x_1 > 0.2] \wedge [x_2 < 0.77] \wedge [x_5 < \pi/4]$. In this representation, for each feature, you need to choose an inequality ($<$ or $>$) and a threshold. Since the thresholds can be arbitrary real values, there are now infinitely many possibilities: $|\mathcal{H}| = 2^D \times \infty = \infty$. However, you can immediately recognize that on binary features, there really is no difference between $[x_2 < 0.77]$ and $[x_2 < 0.12]$ and any other number of infinitely many possibilities. In other words, *even though there are infinitely many hypotheses, there are only finitely many behaviors.*

The **Vapnik-Chernovenkis dimension** (or **VC dimension**) is a classic measure of complexity of infinite hypothesis classes based on this intuition[3]. The VC dimension is a very classification-oriented notion of complexity. The idea is to look at a finite set of unlabeled examples, such as those in Figure 10.2. The question is: no matter how these points were labeled, would we be able to find a hypothesis that correctly classifies them. The idea is that as you add more points, being able to represent an arbitrary labeling becomes harder and harder. For instance, regardless of how the three points are labeled, you can find a linear classifier that agrees with that classification. However, for the four points, there exists a labeling for which you cannot find a perfect classifier. The VC dimension is the *maximum* number of points for which you can always find such a classifier.

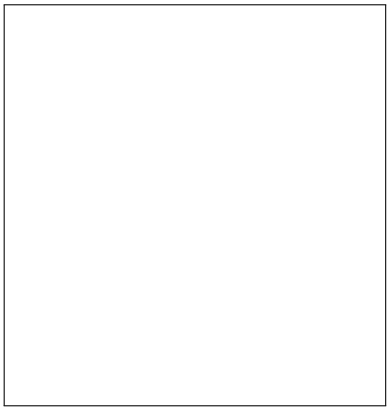


Figure 10.2: `thy:vcex`: figure with three and four examples

[3] Yes, this is the same Vapnik who is credited with the creation of the support vector machine.

? What is that labeling? What is it's name?

You can think of VC dimension as a game between you and an adversary. To play this game, *you* choose $K$ unlabeled points however you want. Then your adversary looks at those $K$ points and assigns binary labels to them them however he wants. You must then find a hypothesis (classifier) that agrees with his labeling. You win if you can find such a hypothesis; he wins if you cannot. The VC dimension of your hypothesis class is the *maximum* number of points $K$ so that you can always win this game. This leads to the following formal definition, where you can interpret there exists as your move and for all as adversary's move.

**Definitions 2.** *For data drawn from some space $\mathcal{X}$, the VC dimension of a hypothesis space $\mathcal{H}$ over $\mathcal{X}$ is the* maximal *$K$ such that: there exists a set $X \subseteq \mathcal{X}$ of size $|X| = K$, such that for all binary labelings of $X$, there exists a function $f \in \mathcal{H}$ that matches this labeling.*

In general, it is much easier to show that the VC dimension is at least some value; it is much harder to show that it is at most some value. For example, following on the example from Figure 10.2, the image of three points (plus a little argumentation) is enough to show that the VC dimension of linear classifiers in two dimension is *at least three*.

To show that the VC dimension is *exactly three* it suffices to show that you *cannot* find a set of four points such that you win this game against the adversary. This is much more difficult. In the proof that the VC dimension is at least three, you simply need to provide an example of three points, and then work through the small number of possible labelings of that data. To show that it is at most three, you need to argue that no matter what set of four point you pick, you cannot win the game.

VC

margins

small norms

## 10.7 Learning with Noise

## 10.8 Agnostic Learning

## 10.9 Error versus Regret

Despite the fact that there's no way to get better than 20% error on this distribution, it would be nice to say that you can still learn something from it. For instance, the predictor that always guesses $y = x$ seems like the "right" thing to do. Based on this observation, maybe we can rephrase the goal of learning as to find a function that does as well as the distribution allows. In other words, on this data, you would hope to get 20% error. On some other distribution, you would hope to get $X$% error, where $X$% is the best you could do.

This notion of "best you could do" is sufficiently important that it has a name: the **Bayes error rate**. This is the error rate of the best possible classifier, the so-called **Bayes optimal classifier**. If you knew the underlying distribution $\mathcal{D}$, you could actually *write down* the exact Bayes optimal classifier explicitly. (This is why learning is uninteresting in the case that you know $\mathcal{D}$.) It simply has the form:

$$f^{\text{Bayes}}(\boldsymbol{x}) = \begin{cases} +1 & \text{if } \mathcal{D}(\boldsymbol{x}, +1) > \mathcal{D}(\boldsymbol{x}, -1) \\ -1 & \text{otherwise} \end{cases} \tag{10.7}$$

The Bayes optimal error rate is the error rate that this (hypothetical) classifier achieves:

$$\epsilon^{\text{Bayes}} = \mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}}\left[y \neq f^{\text{Bayes}}(\boldsymbol{x})\right] \tag{10.8}$$

## 10.10 Exercises

**Exercise 10.1. TODO...**

# 11 | Ensemble Methods

–

**Learning Objectives:**

- Implement bagging and explain how it reduces variance in a predictor.
- Explain the difference between a weak learner and a strong learner.
- Derive the AdaBoost algorithm.
- Understand the relationship between boosting decision stumps and linear classification.

Dependencies:

Groups of people can often make better decisions than individuals, especially when group members each come in with their own biases. The same is true in machine learning. Ensemble methods are learning models that achieve performance by combining the opinions of multiple learners. In doing so, you can often get away with using much *simpler* learners and still achieve great performance. Moreover, ensembles are inherantly parallel, which can make them much more efficient at training and test time, if you have access to multiple processors.

In this chapter, you will learn about various ways of combining **base learners** into **ensembles**. One of the shocking results we will see is that you can take a learning model that only ever does slightly better than chance, and turn it into an arbitrarily good learning model, though a technique known as **boosting**. You will also learn how ensembles can decrease the variance of predictors as well as perform regularization.

## 11.1 Voting Multiple Classifiers

All of the learning algorithms you have seen so far are deterministic. If you train a decision tree multiple times on the same data set, you will always get the same tree back. In order to get an effect out of voting multiple classifiers, they need to differ. There are two primary ways to get variability. You can either change the learning algorithm or change the data set.

Building an emsemble by training different classifiers is the most straightforward approach. As in single-model learning, you are given a data set (say, for classification). Instead of learning a single classifier (eg., a decision tree) on this data set, you learn multiple different classifiers. For instance, you might train a decision tree, a perceptron, a KNN, and multiple neural networks with different architectures. Call these classifiers $f_1, \ldots, f_M$. At test time, you can make a prediction by *voting*. On a test example $\hat{x}$, you compute $\hat{y}_1 = f_1(\hat{x}), \ldots,$

$\hat{y}_M = f_M(\hat{x})$. If there are more +1s in the list $\langle y_1, \ldots, y_M$ then you predict +1; otherwise you predict −1.

The main advantage of ensembles of different classifiers is that it is unlikely that all classifiers will make the same mistake. In fact, as long as every error is made by a minority of the classifiers, you will achieve optimal classification! Unfortunately, the inductive biases of different learning algorithms are highly correlated. This means that different algorithms are prone to similar types of errors. In particular, ensembles tend to reduce the **variance** of classifiers. So if you have a classification algorithm that tends to be very sensitive to small changes in the training data, ensembles are likely to be useful.

? Which of the classifiers you've learned about so far have high variance?

Note that the voting scheme naturally extends to multiclass classification. However, it does not make sense in the contexts of regression, ranking or collective classification. This is because you will rarely see the same exact output predicted twice by two different regression models (or ranking models or collective classification models). For regression, a simple solution is to take the *mean* or *median* prediction from the different models. For ranking and collective classification, different approaches are required.

Instead of training different types of classifiers on the same data set, you can train a single type of classifier (eg., decision tree) on multiple data sets. The question is: where do these multiple data sets come from, since you're only given one at training time?

One option is to fragment your original data set. For instance, you could break it into 10 pieces and build decision trees on each of these pieces individually. Unfortunately, this means that each decision tree is trained on only a very small part of the entire data set and is likely to perform poorly.

A better solution is to use **bootstrap resampling**. This is a technique from the statistics literature based on the following observation. The data set we are given, $D$, is a sample drawn i.i.d. from an unknown distribution $\mathcal{D}$. If we draw a *new* data set $\tilde{D}$ by random sampling from $D$ with replacement[1], then $\tilde{D}$ is *also* a sample from $\mathcal{D}$. Figure 11.1 shows the process of bootstrap resampling of ten objects.

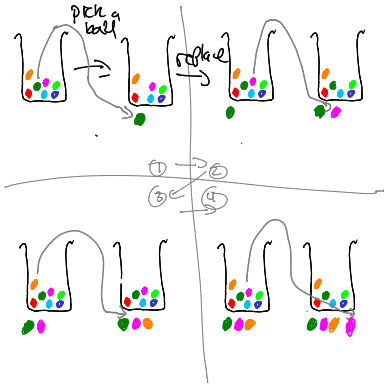


Figure 11.1: picture of sampling with replacement

[1] To sample with replacement, imagine putting all items from $D$ in a hat. To draw a single sample, pick an element at random from that hat, write it down, and then *put it back*.

Applying this idea to ensemble methods yields a technique known as **bagging**. You start with a single data set $D$ that contains $N$ training examples. From this single data set, you create $M$-many "bootstrapped training sets" $\tilde{D}_1, \ldots \tilde{D}_M$. Each of these bootstrapped sets also contains $N$ training examples, drawn randomly from $D$ with replacement. You can then train a decision tree (or other model) seperately on each of these data sets to obtain classifiers $f_1, \ldots, f_M$. As before, you can use these classifiers to vote on new test points.

Note that the bootstrapped data sets will be similar. However, they will not be *too* similar. For example, if $N$ is large then the number of

examples that are not present in any particular bootstrapped sample is relatively large. The probability that the first training example is not selected once is $(1 - 1/N)$. The probability that it is not selected at all is $(1 - 1/N)^N$. As $N \to \infty$, this tends to $1/e \approx 0.3679$. (Already for $N = 1000$ this is correct to four decimal points.) So only about 63% of the original training examples will be represented in any given bootstrapped set.

Since bagging tends to reduce *variance*, it provides an alternative approach to regularization. That is, even if each of the learned classifiers $f_1, \ldots, f_M$ are individually overfit, they are likely to be overfit to different things. Through voting, you are able to overcome a significant portion of this overfitting. Figure **??** shows this effect by comparing regularization via hyperparameters to regularization via bagging.

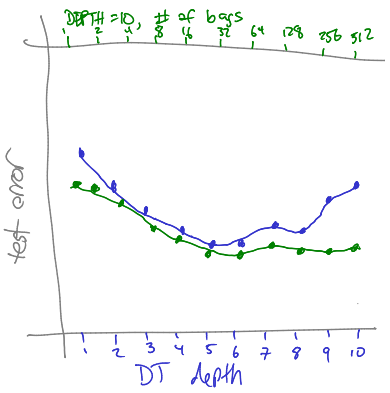



Figure 11.2: graph depicting overfitting using regularization versus bagging

## 11.2 Boosting Weak Learners

Boosting is the process of taking a crummy learning algorithm (technically called a **weak learner**) and turning it into a great learning algorithm (technically, a **strong learner**). Of all the ideas that originated in the theoretical machine learning community, boosting has had—perhaps—the greatest practical impact. The idea of boosting is reminiscent of what you (like me!) might have thought when you first learned about file compression. If I compress a file, and then re-compress it, and then re-compress it, eventually I'll end up with a final that's only one byte in size!

To be more formal, let's define a **strong learning algorithm** $\mathcal{L}$ as follows. When given a desired error rate $\epsilon$, a failure probability $\delta$ and access to "enough" labeled examples from some distribution $\mathcal{D}$, then, with high probability (at least $1 - \delta$), $\mathcal{L}$ learns a classifier $f$ that has error at most $\epsilon$. This is precisely the definition of **PAC** learning that you learned about in Chapter 10. Building a strong learning algorithm might be difficult. We can as if, instead, it is possible to build a **weak learning algorithm** $\mathcal{W}$ that only has to achieve an error rate of 49%, rather than some arbitrary user-defined parameter $\epsilon$. (49% is arbitrary: anything strictly less than 50% would be fine.)

Boosting is more of a "framework" than an algorithm. It's a framework for taking a weak learning algorithm $\mathcal{W}$ and turning it into a strong learning algorithm. The particular boosting algorithm discussed here is **AdaBoost**, short for "adaptive boosting algorithm." AdaBoost is famous because it was one of the first *practical* boosting algorithms: it runs in polynomial time and does not require you to define a large number of hyperparameters. It gets its name from the latter benefit: it automatically *adapts* to the data that you give it.

**Algorithm 31** AdaBoost($\mathcal{W}$, $\mathcal{D}$, $K$)

1: $\boldsymbol{d}^{(0)} \leftarrow \langle \frac{1}{N}, \frac{1}{N}, \ldots, \frac{1}{N} \rangle$ // Initialize uniform importance to each example
2: **for** $k = 1 \ldots K$ **do**
3: $f^{(k)} \leftarrow \mathcal{W}(\mathcal{D}, \boldsymbol{d}^{(k-1)})$ // Train $k$th classifier on weighted data
4: $\hat{y}_n \leftarrow f^{(k)}(\boldsymbol{x}_n), \forall n$ // Make predictions on training data
5: $\hat{\epsilon}^{(k)} \leftarrow \sum_n d_n^{(k-1)} [y_n \neq \hat{y}_n]$ // Compute weighted training error
6: $\alpha^{(k)} \leftarrow \frac{1}{2} \log \left( \frac{1-\hat{\epsilon}^{(k)}}{\hat{\epsilon}^{(k)}} \right)$ // Compute "adaptive" parameter
7: $d_n^{(k)} \leftarrow \frac{1}{Z} d_n^{(k-1)} \exp[-\alpha^{(k)} y_n \hat{y}_n], \forall n$ // Re-weight examples and normalize
8: **end for**
9: **return** $f(\hat{\boldsymbol{x}}) = \operatorname{sgn}\left[\sum_k \alpha^{(k)} f^{(k)}(\hat{\boldsymbol{x}})\right]$ // Return (weighted) voted classifier

The intuition behind AdaBoost is like studying for an exam by using a past exam. You take the past exam and grade yourself. The questions that you got right, you pay less attention to. Those that you got *wrong*, you study more. Then you take the exam again and repeat this process. You continually *down-weight* the importance of questions you routinely answer correctly and *up-weight* the importance of questions you routinely answer incorrectly. After going over the exam multiple times, you hope to have mastered everything.

The precise AdaBoost training algorithm is shown in Algorithm 11.2. The basic functioning of the algorithm is to maintain a *weight distribution* $\boldsymbol{d}$, over data points. A weak learner, $f^{(k)}$ is trained on this weighted data. (Note that we implicitly assume that our weak learner can accept weighted training data, a relatively mild assumption that is nearly always true.) The (weighted) error rate of $f^{(k)}$ is used to determine the *adaptive parameter* $\alpha$, which controls how "important" $f^{(k)}$ is. As long as the weak learner does, indeed, achieve $< 50\%$ error, then $\alpha$ will be greater than zero. As the error drops to zero, $\alpha$ grows without bound.

? What happens if the weak learning assumption is violated and $\hat{\epsilon}$ is equal to 50%? What if it is worse than 50%? What does this mean, in practice?

After the adaptive parameter is computed, the weight distibution is updated for the next iteration. As desired, examples that are correctly classified (for which $y_n \hat{y}_n = +1$) have their weight *decreased* multiplicatively. Examples that are incorrectly classified ($y_n \hat{y}_n = -1$) have their weight *increased* multiplicatively. The $Z$ term is a normralization constant to ensure that the sum of $\boldsymbol{d}$ is one (i.e., $\boldsymbol{d}$ can be interpreted as a distribution). The final classifier returned by AdaBoost is a weighted vote of the individual classifiers, with weights given by the adaptive parameters.

To better understand why $\alpha$ is defined as it is, suppose that our weak learner simply returns a *constant* function that returns the (weighted) majority class. So if the total weight of positive examples exceeds that of negative examples, $f(\boldsymbol{x}) = +1$ for all $\boldsymbol{x}$; otherwise $f(\boldsymbol{x}) = -1$ for all $\boldsymbol{x}$. To make the problem moderately interesting, suppose that in the original training set, there are 80 positive ex-

amples and 20 negative examples. In this case, $f^{(1)}(\boldsymbol{x}) = +1$. It's weighted error rate will be $\hat{\epsilon}^{(1)} = 0.2$ because it gets every negative example wrong. Computing, we get $\alpha^{(1)} = \frac{1}{2}\log 4$. Before normalization, we get the new weight for each positive (correct) example to be $1\exp[-\frac{1}{2}\log 4] = \frac{1}{2}$. The weight for each negative (incorrect) example becomes $1\exp[\frac{1}{2}\log 4] = 2$. We can compute $Z = 80{\times}\frac{1}{2} + 20{\times}2 = 80$. Therefore, after normalization, the weight distribution on any single positive example is $\frac{1}{160}$ and the weight on any negative example is $\frac{1}{40}$. However, since there are 80 positive examples and 20 negative examples, the *cumulative* weight on all positive examples is $80{\times}\frac{1}{160} = \frac{1}{2}$; the cumulative weight on all negative examples is $20{\times}\frac{1}{40} = \frac{1}{2}$. Thus, after a single boosting iteration, the data has become precisely evenly weighted. This guarantees that in the next iteration, our weak learner *must* do something more interesting than majority voting if it is to achieve an error rate *less than* 50%, as required.

? This example uses concrete numbers, but the same result holds no matter what the data distribution looks like nor how many examples there are. Write out the general case to see that you will still arrive at an even weighting after one iteration.

One of the major attractions of boosting is that it is perhaps easy to design computationally efficient weak learners. A very popular type of weak learner is a **shallow decision tree**: a decision tree with a small depth limit. Figure 11.3 shows test error rates for decision trees of different maximum depths (the different curves) run for differing numbers of boosting iterations (the x-axis). As you can see, if you are willing to boost for many iterations, very shallow trees are quite effective.

In fact, a very popular weak learner is a decision **decision stump**: a decision tree that can only ask *one* question. This may seem like a silly model (and, in fact, it is on it's own), but when combined with boosting, it becomes very effective. To understand why, suppose for a moment that our data consists only of binary features, so that any question that a decision tree might ask is of the form "is feature 5 on?" By concentrating on decision stumps, all weak functions must have the form $f(\boldsymbol{x}) = s(2x_d - 1)$, where $s \in \{\pm 1\}$ and $d$ indexes *some* feature.

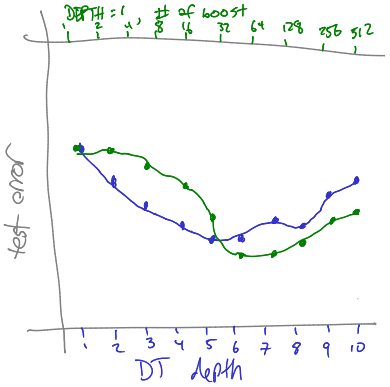



Figure 11.3: perf comparison of depth vs # boost

? Why do the functions have this form?

Now, consider the *final* form of a function learned by AdaBoost. We can expand it as follow, where we let $f_k$ denote the single feature selected by the $k$th decision stump and let $s_k$ denote its sign:

$$f(\boldsymbol{x}) = \operatorname{sgn}\left[\sum_k \alpha_k f^{(k)}(\boldsymbol{x})\right] \tag{11.1}$$

$$= \operatorname{sgn}\left[\sum_k \alpha_k s_k (2x_{f_k} - 1)\right] \tag{11.2}$$

$$= \operatorname{sgn}\left[\sum_k 2\alpha_k s_k x_{f_k} - \sum_k \alpha_k s_k\right] \tag{11.3}$$

$$= \operatorname{sgn}\left[\boldsymbol{w}\cdot\boldsymbol{x} + b\right] \tag{11.4}$$

**Algorithm 32** RANDOMFORESTTRAIN($\mathcal{D}$, *depth*, $K$)

**for** $k = 1 \dots K$ **do**
  $t^{(k)} \leftarrow$ complete binary tree of depth *depth* with random feature splits
  $f^{(k)} \leftarrow$ the function computed by $t^{(k)}$, with leaves filled in by $\mathcal{D}$
**end for**
**return** $f(\hat{\boldsymbol{x}}) = \text{sgn}\left[\sum_k f^{(k)}(\hat{\boldsymbol{x}})\right]$ // Return voted classifier

$$\text{where } w_d = \sum_{k:f_k=d} 2\alpha_k s_k \quad \text{and} \quad b = -\sum_k \alpha_k s_k \tag{11.5}$$

Thus, when working with decision stumps, AdaBoost actually provides an algorithm for learning **linear classifiers**! In fact, this connection has recently been strengthened: you can show that AdaBoost provides an algorithm for optimizing **exponential loss**. (However, this connection is beyond the scope of this book.)

As a further example, consider the case of boosting a **linear classifier**. In this case, if we let the $k$th weak classifier be parameterized by $\boldsymbol{w}^{(k)}$ and $b^{(k)}$, the overall predictor will have the form:

$$f(\boldsymbol{x}) = \text{sgn}\left[\sum_k \alpha_k \text{sgn}\left(\boldsymbol{w}^{(k)} \cdot \boldsymbol{x} + b^{(k)}\right)\right] \tag{11.6}$$

You can notice that this is nothing but a two-layer **neural network**, with $K$-many hidden units! Of course it's not a classifically trained neural network (once you learn $\boldsymbol{w}^{(k)}$ you never go back and update it), but the structure is identical.

## 11.3 *Random Ensembles*

One of the most computationally expensive aspects of ensembles of decision trees is training the decision trees. This is very fast for decision stumps, but for deeper trees it can be prohibitively expensive. The expensive part is choosing the tree structure. Once the tree structure is chosen, it is very cheap to fill in the leaves (i.e., the predictions of the trees) using the training data.

An efficient and surprisingly effective alternative is to use trees with fixed structures and random features. Collections of trees are called forests, and so classifiers built like this are called **random forests**. The random forest training algorithm, shown in Algorithm 11.3 is quite short. It takes three arguments: the data, a desired depth of the decision trees, and a number $K$ of total decision trees to build.

The algorithm generates each of the $K$ trees independently, which makes it very easy to parallelize. For each trees, it constructs a full binary tree of depth *depth*. The features used at the branches of this

tree are selected randomly, typically *with replacement*, meaning that the same feature can appear multiple times, even in one branch. The leaves of this tree, where predictions are made, are filled in based on the training data. This last step is the *only* point at which the training data is used. The resulting classifier is then just a voting of the $K$-many random trees.

The most amazing thing about this approach is that it actually works remarkably well. It tends to work best when all of the features are at least marginally relevant, since the number of features selected for any given tree is small. An intuitive reason that it works well is the following. Some of the trees will query on useless features. These trees will essentially make random predictions. But some of the trees will happen to query on good features and will make good predictions (because the leaves are estimated based on the training data). If you have enough trees, the random ones will wash out as noise, and only the good trees will have an effect on the final classification.

## 11.4 Exercises

**Exercise 11.1. TODO...**

# 12 | Efficient Learning

**Learning Objectives:**
- Understand and be able to implement stochastic gradient descent algorithms.
- Compare and contrast small versus large batch sizes in stochastic optimization.
- Derive subgradients for sparse regularizers.
- Implement feature hashing.

Dependencies:

So far, our focus has been on *models* of learning and basic algorithms for those models. We have not placed much emphasis on how to learn *quickly*. The basic techniques you learned about so far are enough to get learning algorithms running on tens or hundreds of thousands of examples. But if you want to build an algorithm for web page ranking, you will need to deal with millions or billions of examples, in hundreds of thousands of dimensions. The basic approaches you have seen so far are insufficient to achieve such a massive scale.

In this chapter, you will learn some techniques for scaling learning algorithms. This are useful even when you do not have billions of training examples, because it's always nice to have a program that runs quickly. You will see techniques for speeding up both model training and model prediction. The focus in this chapter is on linear models (for simplicity), but most of what you will learn applies more generally.

## 12.1 What Does it Mean to be Fast?

Everyone always wants fast algorithms. In the context of machine learning, this can mean many things. You might want fast training algorithms, or perhaps training algorithms that scale to very large data sets (for instance, ones that will not fit in main memory). You might want training algorithms that can be easily parallelized. Or, you might not care about training efficiency, since it is an offline process, and only care about how quickly your learned functions can make classification decisions.

It is important to separate out these desires. If you care about efficiency at training time, then what you are really asking for are more efficient learning algorithms. On the other hand, if you care about efficiency at test time, then you are asking for *models* that can be quickly evaluated.

One issue that is not covered in this chapter is parallel learning.

This is largely because it is currently not a well-understood area in machine learning. There are many aspects of parallelism that come into play, such as the speed of communication across the network, whether you have shared memory, etc. Right now, this the general, poor-man's approach to parallelization, is to employ ensembles.

## 12.2 Stochastic Optimization

During training of most learning algorithms, you consider the entire data set simultaneously. This is certainly true of gradient descent algorithms for regularized linear classifiers (recall Algorithm 6.4), in which you first compute a gradient over the entire training data (for simplicity, consider the unbiased case):

$$\boldsymbol{g} = \sum_n \nabla_{\boldsymbol{w}} \ell(y_n, \boldsymbol{w} \cdot \boldsymbol{x}_n) + \lambda \boldsymbol{w} \tag{12.1}$$

where $\ell(y, \hat{y})$ is some loss function. Then you update the weights by $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \boldsymbol{g}$. In this algorithm, in order to make a *single* update, you have to look at *every* training example.

When there are billions of training examples, it is a bit silly to look at every one before doing anything. Perhaps just on the basis of the first few examples, you can already start learning something!

Stochastic optimization involves thinking of your training data as a big distribution over examples. A draw from this distribution corresponds to picking some example (uniformly at random) from your data set. Viewed this way, the optimization problem becomes a **stochastic optimization** problem, because you are trying to optimize some function (say, a regularized linear classifier) over a probability distribution. You can derive this intepretation directly as follows:

$$\boldsymbol{w}^* = \arg\max_{\boldsymbol{w}} \sum_n \ell(y_n, \boldsymbol{w} \cdot \boldsymbol{x}_n) + R(\boldsymbol{w}) \qquad \text{definition} \tag{12.2}$$

$$= \arg\max_{\boldsymbol{w}} \sum_n \left[ \ell(y_n, \boldsymbol{w} \cdot \boldsymbol{x}_n) + \frac{1}{N} R(\boldsymbol{w}) \right] \qquad \text{move } R \text{ inside sum} \tag{12.3}$$

$$= \arg\max_{\boldsymbol{w}} \sum_n \left[ \frac{1}{N} \ell(y_n, \boldsymbol{w} \cdot \boldsymbol{x}_n) + \frac{1}{N^2} R(\boldsymbol{w}) \right] \qquad \text{divide through by } N \tag{12.4}$$

$$= \arg\max_{\boldsymbol{w}} \mathbb{E}_{(y,\boldsymbol{x})\sim D} \left[ \ell(y, \boldsymbol{w} \cdot \boldsymbol{x}) + \frac{1}{N} R(\boldsymbol{w}) \right] \qquad \text{write as expectation} \tag{12.5}$$

where $D$ is the training data distribution (12.6)

Given this framework, you have the following general form of an

**Algorithm 33** StochasticGradientDescent($\mathcal{F}$, $\mathcal{D}$, $S$, $K$, $\eta_1$, ...)

$z^{(0)} \leftarrow \langle 0, 0, \ldots, 0 \rangle$ // initialize variable we are optimizing
**for** $k = 1 \ldots K$ **do**
  $D^{(k)} \leftarrow S$-many random data points from $\mathcal{D}$
  $g^{(k)} \leftarrow \nabla_z \mathcal{F}(D^{(k)})\big|_{z^{(k-1)}}$ // compute gradient on sample
  $z^{(k)} \leftarrow z^{(k-1)} - \eta^{(k)} g^{(k)}$ // take a step down the gradient
**end for**
**return** $z^{(K)}$

optimization problem:

$$\min_z \quad \mathbb{E}_\zeta[\mathcal{F}(z, \zeta)] \tag{12.7}$$

In the example, $\zeta$ denotes the random choice of examples over the dataset, $z$ denotes the weight vector and $\mathcal{F}(\boldsymbol{w}, \zeta)$ denotes the loss on that example *plus* a fraction of the regularizer.

Stochastic optimization problems are formally *harder* than regular (deterministic) optimization problems because you do not even get access to exact function values and gradients. The only access you have to the function $\mathcal{F}$ that you wish to optimize are noisy measurements, governed by the distribution over $\zeta$. Despite this lack of information, you can still run a gradient-based algorithm, where you simply compute *local* gradients on a current sample of data.

More precisely, you can draw a data point at random from your data set. This is analogous to drawing a single value $\zeta$ from its distribution. You can compute the gradient of $\mathcal{F}$ just at that point. In this case of a 2-norm regularized linear model, this is simply $\boldsymbol{g} = \nabla_{\boldsymbol{w}} \ell(y, \boldsymbol{w} \cdot \boldsymbol{x}) + \frac{1}{N}\boldsymbol{w}$, where $(y, \boldsymbol{x})$ is the random point you selected. Given this *estimate* of the gradient (it's an estimate because it's based on a single random draw), you can take a small gradient step $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \boldsymbol{g}$.

This is the **stochastic gradient descent** algorithm (**SGD**). In practice, taking gradients with respect to a *single* data point might be too myopic. In such cases, it is useful to use a small **batch** of data. Here, you can draw 10 random examples from the training data and compute a small gradient (estimate) based on those examples: $\boldsymbol{g} = \sum_{m=1}^{10} \nabla_{\boldsymbol{w}} \ell(y_m, \boldsymbol{w} \cdot \boldsymbol{x}_m) + \frac{10}{N}\boldsymbol{w}$, where you need to include 10 counts of the regularizer. Popular batch sizes are 1 (single points) and 10. The generic SGD algorithm is depicted in Algorithm 12.2, which takes $K$-many steps over batches of $S$-many examples.

In stochastic gradient descent, it is *imperative* to choose good step sizes. It is also very important that the steps get smaller over time at a reasonable slow rate. In particular, convergence can be guaranteed for learning rates of the form: $\eta^{(k)} = \frac{\eta_0}{\sqrt{k}}$, where $\eta_0$ is a fixed, initial step size, typically 0.01, 0.1 or 1 depending on how quickly you ex-

pect the algorithm to converge. Unfortunately, in comparisong to gradient descent, stochastic gradient is quite sensitive to the selection of a good learning rate.

There is one more practical issues related to the use of SGD as a learning algorithm: do you *really* select a random point (or subset of random points) at each step, or do you stream through the data in order. The answer is akin to the answer of the same question for the perceptron algorithm (Chapter 3). If you do not permute your data at all, very bad things can happen. If you *do* permute your data once and then do multiple passes over that same permutation, it will converge, but more slowly. In theory, you really should permute every iteration. If your data is small enough to fit in memory, this is not a big deal: you will only pay for cache misses. However, if your data is too large for memory and resides on a magnetic disk that has a slow seek time, randomly seeking to new data points for each example is prohibitivly slow, and you will likely need to forgo permuting the data. The speed hit in convergence speed will almost certainly be recovered by the speed gain in not having to seek on disk routinely. (Note that the story is very different for solid state disks, on which random accesses really are quite efficient.)

## 12.3 Sparse Regularization

For many learning algorithms, the test-time efficiency is governed by how many features are used for prediction. This is one reason decision trees tend to be among the fastest predictors: they only use a small number of features. Especially in cases where the actual *computation* of these features is expensive, cutting down on the number that are used at test time can yield huge gains in efficiency. Moreover, the amount of memory used to make predictions is also typically governed by the number of features. (Note: this is *not* true of kernel methods like support vector machines, in which the dominant cost is the number of support vectors.) Furthermore, you may simply *believe* that your learning problem can be solved with a very small number of features: this is a very reasonable form of inductive bias.

This is the idea behind sparse models, and in particular, sparse regularizers. One of the disadvantages of a 2-norm regularizer for linear models is that they tend to never produce weights that are *exactly* zero. They get close to zero, but never hit it. To understand why, as a weight $w_d$ approaches zero, its gradient *also* approaches zero. Thus, even if the weight *should* be zero, it will essentially never get there because of the constantly shrinking gradient.

This suggests that an alternative regularizer is required to yield a sparse inductive bias. An ideal case would be the zero-norm regular-

izer, which simply counts the number of non-zero values in a vector: $||\boldsymbol{w}||_0 = \sum_d [w_d \neq 0]$. If you could minimize this regularizer, you would be explicitly minimizing the number of non-zero features. Unfortunately, not only is the zero-norm non-convex, it's also discrete. Optimizing it is NP-hard.

A reasonable middle-ground is the one-norm: $||\boldsymbol{w}||_1 = \sum_d |w_d|$. It is indeed convex: in fact, it is the tighest $\ell_p$ norm that *is* convex. Moreover, its gradients do not go to zero as in the two-norm. Just as hinge-loss is the tightest convex upper bound on zero-one error, the one-norm is the tighest convex upper bound on the zero-norm.

At this point, you should be content. You can take your subgradient optimizer for arbitrary functions and plug in the one-norm as a regularizer. The one-norm is surely non-differentiable at $w_d = 0$, but you can simply choose any value in the range $[-1, +1]$ as a subgradient at that point. (You should choose zero.)

Unfortunately, this does not quite work the way you might expect. The issue is that the gradient might "overstep" zero and you will never end up with a solution that is particularly sparse. For example, at the end of one gradient step, you might have $w_3 = 0.6$. Your gradient might have $g_6 = 0.8$ and your gradient step (assuming $\eta = 1$) will update so that the new $w_3 = -0.2$. In the subsequent iteration, you might have $g_6 = -0.3$ and step to $w_3 = 0.1$.

This observation leads to the idea of **trucated gradients**. The idea is simple: if you have a gradient that would step you over $w_d = 0$, then just set $w_d = 0$. In the easy case when the learning rate is 1, this means that if the *sign* of $w_d - g_d$ is different than the sign of $w_d$ then you truncate the gradient step and simply set $w_d = 0$. In other words, $g_d$ should never be *larger* than $w_d$ Once you incorporate learning rates, you can express this as:

$$g_d \leftarrow \begin{cases} g_d & \text{if } w_d > 0 \text{ and } g_d \leq \frac{1}{\eta^{(k)}} w_d \\ g_d & \text{if } w_d < 0 \text{ and } g_d \geq \frac{1}{\eta^{(k)}} w_d \\ 0 & \text{otherwise} \end{cases} \tag{12.8}$$

This works quite well in the case of subgradient descent. It works somewhat less well in the case of *stochastic* subgradient descent. The problem that arises in the stochastic case is that wherever you choose to stop optimizing, you will have just touched a single example (or small batch of examples), which will increase the weights for a lot of features, before the regularizer "has time" to shrink them back down to zero. You will still end up with somewhat sparse solutions, but not as sparse as they could be. There are algorithms for dealing with this situation, but they all have a heuristic flavor to them and are beyond the scope of this book.

## 12.4 Feature Hashing

As much as speed is a bottleneck in prediction, so often is memory usage. If you have a very large number of features, the amount of memory that it takes to store weights for all of them can become prohibitive, especially if you wish to run your algorithm on small devices. Feature hashing is an incredibly simple technique for reducing the memory footprint of linear models, with very small sacrifices in accuracy.

The basic idea is to replace all of your features with hashed versions of those features, thus reducing your space from $D$-many feature weights to $P$-many feature weights, where $P$ is the range of the hash function. You can actually think of hashing as a (randomized) feature mapping $\phi : \mathbb{R}^D \to \mathbb{R}^P$, for some $P \ll D$. The idea is as follows. First, you choose a hash function $h$ whose domain is $[D] = \{1, 2, \ldots, D\}$ and whose range is $[P]$. Then, when you receive a feature vector $\boldsymbol{x} \in \mathbb{R}^D$, you map it to a shorter feature vector $\hat{\boldsymbol{x}} \in \mathbb{R}^P$. Algorithmically, you can think of this mapping as follows:

1. Initialize $\hat{\boldsymbol{x}} = \langle 0, 0, \ldots, 0 \rangle$
2. For each $d = 1 \ldots D$:
   (a) Hash $d$ to position $p = h(d)$
   (b) Update the $p$th position by adding $x_d$: $\hat{x}_p \leftarrow \hat{\boldsymbol{x}}_p + x_d$
3. Return $\hat{\boldsymbol{x}}$

Mathematically, the mapping looks like:

$$\phi(\boldsymbol{x})_p = \sum_d [h(d) = p] x_d \quad = \sum_{d \in h^{-1}(p)} x_d \tag{12.9}$$

where $h^{-1}(p) = \{d : h(d) = p\}$.

In the (unrealistic) case where $P = D$ and $h$ simply encodes a permutation, then this mapping does not change the learning problem at all. All it does is rename all of the features. In practice, $P \ll D$ and there will be collisions. In this context, a collision means that two features, which are really different, end up looking the same to the learning algorithm. For instance, "is it sunny today?" and "did my favorite sports team win last night?" might get mapped to the same location after hashing. The hope is that the learning algorithm is sufficiently robust to noise that it can handle this case well.

Consider the kernel defined by this hash mapping. Namely:

$$K^{(\text{hash})}(\boldsymbol{x}, \boldsymbol{z}) = \phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{z}) \tag{12.10}$$

$$= \sum_p \left( \sum_d [h(d) = p] x_d \right) \left( \sum_d [h(d) = p] z_d \right) \tag{12.11}$$

$$= \sum_p \sum_{d,e} [h(d) = p][h(e) = p] x_d z_e \tag{12.12}$$

$$= \sum_d \sum_{e \in h^{-1}(h(d))} x_d z_e \tag{12.13}$$

$$= \boldsymbol{x} \cdot \boldsymbol{z} + \sum_d \sum_{\substack{e \neq d, \\ e \in h^{-1}(h(d))}} x_d z_e \tag{12.14}$$

This **hash kernel** has the form of a linear kernel plus a small number of quadratic terms. The particular quadratic terms are exactly those given by collisions of the hash function.

There are two things to notice about this. The first is that collisions might not actually be bad things! In a sense, they're giving you a little extra representational power. In particular, if the hash function happens to select out feature pairs that benefit from being paired, then you now have a better representation. The second is that even if this doesn't happen, the quadratic term in the kernel has only a small effect on the overall prediction. In particular, if you assume that your hash function is pairwise independent (a common assumption of hash functions), then the *expected value* of this quadratic term is zero, and its variance decreases at a rate of $\mathcal{O}(P^{-2})$. In other words, if you choose $P \approx 100$, then the variance is on the order of 0.0001.

## 12.5 Exercises

**Exercise 12.1. TODO...**

# 13 | Unsupervised Learning

–

**Learning Objectives:**

- Explain the difference between linear and non-linear dimensionality reduction.
- Relate the view of PCA as maximizing variance with the view of it as minimizing reconstruction error.
- Implement latent semantic analysis for text data.
- Motivate manifold learning from the perspective of reconstruction error.
- Understand *K*-means clustering as distance minimization.
- Explain the importance of initialization in *k*-means and furthest-first heuristic.
- Implement agglomerative clustering.
- Argue whether spectral clustering is a clustering algorithm or a dimensionality reduction algorithm.

Dependencies:

If you have access to labeled training data, you know what to do. This is the "supervised" setting, in which you have a teacher telling you the right answers. Unfortunately, finding such a teacher is often difficult, expensive, or down right impossible. In those cases, you might still want to be able to analyze your data, even though you do not have labels.

Unsupervised learning is learning without a teacher. One basic thing that you might want to do with data is to **visualize** it. Sadly, it is difficult to visualize things in more than two (or three) dimensions, and most data is in hundreds of dimensions (or more). **Dimensionality reduction** is the problem of taking high dimensional data and **embedding** it in a lower dimension space. Another thing you might want to do is automatically derive a partitioning of the data into **clusters**. You've already learned a basic approach for doing this: the *k*-means algorithm (Chapter 2). Here you will analyze this algorithm to see why it works. You will also learn more advanced clustering approaches.

## 13.1 *K-Means Clustering, Revisited*

The *K*-means clustering algorithm is re-presented in Algorithm 13.1. There are two very basic questions about this algorithm: (1) does it converge (and if so, how quickly); (2) how sensitive it is to initialization? The answers to these questions, detailed below, are: (1) yes it converges, and it converges very quickly in practice (though slowly in theory); (2) yes it is sensitive to initialization, but there are good ways to initialize it.

Consider the question of convergence. The following theorem states that the *K*-Means algorithm converges, though it does not say how quickly it happens. The method of proving the convergence is to specify a **clustering quality** objective function, and then to show that the *K*-Means algorithm converges to a (local) optimum of that objective function. The particular objective function that *K*-Means

**Algorithm 34** K-MEANS($\mathbf{D}$, $K$)

```
1:  for k = 1 to K do
2:      μ_k ← some random location          // randomly initialize mean for kth cluster
3:  end for
4:  repeat
5:      for n = 1 to N do
6:          z_n ← argmin_k ||μ_k − x_n||    // assign example n to closest center
7:      end for
8:      for k = 1 to K do
9:          μ_k ← MEAN({ x_n : z_n = k })   // re-estimate mean of cluster k
10:     end for
11: until converged
12: return z                                // return cluster assignments
```

is optimizing is the *sum of squared distances from any data point to its assigned center.* This is a natural generalization of the definition of a mean: the mean of a set of points is the single point that minimizes the sum of squared distances from the mean to every point in the data. Formally, the $K$-Means objective is:

$$\mathcal{L}(\boldsymbol{z}, \boldsymbol{\mu}; \mathbf{D}) = \sum_n \left|\left| \boldsymbol{x}_n - \boldsymbol{\mu}_{z_n} \right|\right|^2 = \sum_k \sum_{n:z_n=k} ||\boldsymbol{x}_n - \boldsymbol{\mu}_k||^2 \tag{13.1}$$

**Theorem 15** ($K$-Means Convergence Theorem)**.** *For any dataset* $\mathbf{D}$ *and any number of clusters* $K$*, the K-means algorithm converges in a finite number of iterations, where convergence is measured by* $\mathcal{L}$ *ceasing the change.*

*Proof of Theorem 15.* The proof works as follows. There are only two points in which the $K$-means algorithm changes the values of $\boldsymbol{\mu}$ or $\boldsymbol{z}$: lines 6 and 9. We will show that both of these operations can never increase the value of $\mathcal{L}$. Assuming this is true, the rest of the argument is as follows. After the first pass through the data, there are are only finitely many possible assignments to $\boldsymbol{z}$ and $\boldsymbol{\mu}$, because $\boldsymbol{z}$ is discrete and because $\boldsymbol{\mu}$ can only take on a finite number of values: means of some subset of the data. Furthermore, $\mathcal{L}$ is lower-bounded by zero. Together, this means that $\mathcal{L}$ cannot decrease more than a finite number of times. Thus, it must stop decreasing at some point, and at that point the algorithm has converged.

It remains to show that lines 6 and 9 decrease $\mathcal{L}$. For line 6, when looking at example $n$, suppose that the previous value of $z_n$ is $a$ and the new value is $b$. It must be the case that $||\boldsymbol{x}_n - \boldsymbol{\mu}_b|| \leq ||\boldsymbol{x}_n - \boldsymbol{\mu}_b||$. Thus, changing from $a$ to $b$ can only decrease $\mathcal{L}$. For line 9, consider the second form of $\mathcal{L}$. Line 9 computes $\boldsymbol{\mu}_k$ as the mean of the data points for which $z_n = k$, which is precisely the point that minimizes squared sitances. Thus, this update to $\boldsymbol{\mu}_k$ can only decrease $\mathcal{L}$. □

There are several aspects of $K$-means that are unfortunate. First, the convergence is only to a local optimum of $\mathcal{L}$. In practice, this

means that you should usually run it 10 times with different initializations and pick the one with minimal resulting $\mathcal{L}$. Second, one can show that there are input datasets and initializations on which it might take an exponential amount of time to converge. Fortunately, these cases almost never happen in practice, and in fact it has recently been shown that (roughly) if you limit the floating point precision of your machine, $K$-means *will* converge in polynomial time (though still only to a local optimum), using techniques of **smoothed analysis**.

The biggest practical issue in $K$-means is initialization. If the cluster means are initialized poorly, you often get convergence to uninteresting solutions. A useful heuristic is the **furthest-first heuristic**. This gives a way to perform a semi-random initialization that attempts to pick initial means as far from each other as possible. The heuristic is sketched below:

1. Pick a random example $m$ and set $\boldsymbol{\mu}_1 = \boldsymbol{x}_m$.
2. For $k = 2 \dots K$:
   (a) Find the example $m$ that is as far as possible from *all* previously selected means; namely: $m = \arg\max_m \min_{k'<k} ||\boldsymbol{x}_m - \boldsymbol{\mu}_{k'}||^2$ and set $\boldsymbol{\mu}_k = \boldsymbol{x}_m$

In this heuristic, the only bit of randomness is the selection of the first data point. After that, it is completely deterministic (except in the rare case that there are multiple equidistant points in step 2a). It is extremely important that when selecting the 3rd mean, you select that point that *maximizes* the *minimum* distance to the closest other mean. You want the point that's as far away from *all* previous means as possible.

The furthest-first heuristic is just that: a heuristic. It works very well in practice, though can be somewhat sensitive to outliers (which will often get selected as some of the initial means). However, this outlier sensitivity is usually reduced after one iteration through the $K$-means algorithm. Despite being just a heuristic, it is quite useful in practice.

You can turn the heuristic into an algorithm by adding a bit more randomness. This is the idea of the $K$-means++ algorithm, which is a simple randomized tweak on the furthest-first heuristic. The idea is that when you select the $k$th mean, instead of choosing the *absolute furthest* data point, you choose a data point at random, with probability proportional to its distance squared. This is made formal in Algorithm 13.1.

If you use $K$-means++ as an initialization for $K$-means, then you are able to achieve an approximation guarantee on the final value

**Algorithm 35** K-MEANS++($\mathbf{D}$, $K$)

$\boldsymbol{\mu}_1 \leftarrow \boldsymbol{x}_m$ for $m$ chosen uniformly at random // randomly initialize first point
**for** $k = 2$ **to** $K$ **do**
  $d_n \leftarrow \min_{k'<k} ||\boldsymbol{x}_n - \boldsymbol{\mu}_{k'}||^2, \forall n$ // compute distances
  $\boldsymbol{p} \leftarrow \frac{1}{\sum_n n_d} \boldsymbol{d}$ // normalize to probability distribution
  $m \leftarrow$ random sample from $\boldsymbol{p}$ // pick an example at random
  $\boldsymbol{\mu}_k \leftarrow \boldsymbol{x}_m$
**end for**
run K-MEANS using $\boldsymbol{\mu}$ as initial centers

of the objective. This doesn't tell you that you will reach the global optimum, but it *does* tell you that you will get reasonably close. In particular, if $\hat{\mathcal{L}}$ is the value obtained by running $K$-means++, then this will not be "too far" from $\mathcal{L}^{(\text{opt})}$, the true global minimum.

**Theorem 16** ($K$-means++ Approximation Guarantee)**.** *The expected value of the objective returned by K-means++ is never more than $\mathcal{O}(\log K)$ from optimal and can be as close as $\mathcal{O}(1)$ from optimal. Even in the former case, with $2^K$ random restarts, one restart will be $\mathcal{O}(1)$ from optimal (with high probability). Formally: $\mathbb{E}\left[\hat{\mathcal{L}}\right] \leq 8(\log K + 2)\mathcal{L}^{(opt)}$. Moreover, if the data is "well suited" for clustering, then $\mathbb{E}\left[\hat{\mathcal{L}}\right] \leq \mathcal{O}(1)\mathcal{L}^{(opt)}$.*

The notion of "well suited" for clustering informally states that the advantage of going from $K-1$ clusters to $K$ clusters is "large." Formally, it means that $\mathcal{L}_K{}^{(\text{opt})} \leq \epsilon^2 \mathcal{L}_{K-1}{}^{(\text{opt})}$, where $\mathcal{L}_K{}^{(\text{opt})}$ is the optimal value for clustering with $K$ clusters, and $\epsilon$ is the desired degree of approximation. The idea is that if this condition does *not* hold, then you shouldn't bother clustering the data.

One of the biggest practical issues with $K$-means clustering is "choosing $K$." Namely, if someone just hands you a dataset and asks you to cluster it, how many clusters should you produce? This is difficult, because increasing $K$ will always decrease $\mathcal{L}_K{}^{(\text{opt})}$ (until $K > N$), and so simply using $\mathcal{L}$ as a notion of goodness is insufficient (analogous to overfitting in a supervised setting). A number of "information criteria" have been proposed to try to address this problem. They all effectively boil down to "regularizing" $K$ so that the model cannot grow to be too complicated. The two most popular are the Bayes Information Criteria (BIC) and the Akaike Information Criteria (AIC), defined below in the context of $K$-means:

$$\text{BIC:} \quad \arg\min_K \hat{\mathcal{L}}_K + K \log D \tag{13.2}$$

$$\text{AIC:} \quad \arg\min_K \hat{\mathcal{L}}_K + 2KD \tag{13.3}$$

The informal intuition behind these criteria is that increasing $K$ is going to make $\mathcal{L}_K$ go down. However, if it doesn't go down "by enough" then it's not worth doing. In the case of BIC, "by enough"

means by an amount proportional to $\log D$; in the case of AIC, it's proportional to $2D$. Thus, AIC provides a much stronger penalty for many clusters than does BIC, especially in high dimensions.

A more formal intuition for BIC is the following. You ask yourself the question "if I wanted to send this data across a network, how many bits would I need to send?" Clearly you could simply send all of the $N$ examples, each of which would take roughly $\log D$ bits to send. This gives $N \log D$ to send all the data. Alternatively, you could first cluster the data and send the cluster centers. This will take $K \log D$ bits. Then, for each data point, you send its center as well as its deviation from that center. It turns out this will cost exactly $\hat{\mathcal{L}}_K$ bits. Therefore, the BIC is precisely measuring how many bits it will take to send your data using $K$ clusters. The $K$ that minimizes this number of bits is the optimal value.

## 13.2 Linear Dimensionality Reduction

Dimensionality reduction is the task of taking a dataset in high dimensions (say 10000) and reducing it to low dimensions (say 2) while retaining the "important" characteristics of the data. Since this is an unsupervised setting, the notion of important characteristics is difficult to define.

Consider the dataset in Figure **??**, which lives in high dimensions (two) and you want to reduce to low dimensions (one). In the case of linear dimensionality reduction, the only thing you can do is to project the data onto a vector and use the projected distances as the embeddings. Figure **??** shows a projection of this data onto the vector that points in the direction of *maximal variance* of the original dataset. Intuitively, this is a reasonable notion of importance, since this is the direction in which most information is encoded in the data.

For the rest of this section, assume that the data is centered: namely, the mean of all the data is at the origin. (This will simply make the math easier.) Suppose the two dimensional data is $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$ and you're looking for a vector $\boldsymbol{u}$ that points in the direction of maximal variance. You can compute this by projecting each point onto $\boldsymbol{u}$ and looking at the variance of the result. In order for the projection to make sense, you need to constrain $||\boldsymbol{u}||^2 = 1$. In this case, the projections are $\boldsymbol{x}_1 \boldsymbol{u}\cdot, \ldots, \boldsymbol{x}_N \boldsymbol{u}\cdot$. Call these values $p_1, \ldots, p_N$.

The goal is to compute the variance of the $\{p_n\}$s and then choose $\boldsymbol{u}$ to maximize this variance. To compute the variance, you first need to compute the mean. Because the mean of the $\boldsymbol{x}_n$s was zero, the

**MATH REVIEW | EIGENVALUES AND EIGENVECTORS**

the usual...

Figure 13.1:

mean of the $p$s is also zero. This can be seen as follows:

$$\sum_n p_n = \sum_n \boldsymbol{x}_n \cdot \boldsymbol{u} = \left(\sum_n \boldsymbol{x}_n\right) \cdot \boldsymbol{u} = \mathbf{0} \cdot \boldsymbol{u} = \mathbf{0} \tag{13.4}$$

The variance of the $\{p_n\}$ is then just $\sum_n p_n^2$. Finding the optimal $\boldsymbol{u}$ (from the perspective of variance maximization) reduces to the following optimization problem:

$$\max_{\boldsymbol{u}} \quad \sum_n (\boldsymbol{x}_n \cdot \boldsymbol{u})^2 \quad \text{subj. to} \quad ||\boldsymbol{u}||^2 = 1 \tag{13.5}$$

In this problem it becomes apparent why keeping $\boldsymbol{u}$ unit length is important: if not, $\boldsymbol{u}$ would simply stretch to have infinite length to maximize the objective.

It is now helpful to write the collection of datapoints $\boldsymbol{x}_n$ as a $N \times D$ matrix $\mathbf{X}$. If you take this matrix $\mathbf{X}$ and multiply it by $\boldsymbol{u}$, which has dimensions $D \times 1$, you end up with a $N \times 1$ vector whose values are exactly the values $\boldsymbol{p}$. The objective in Eq (13.5) is then just the squared norm of $\boldsymbol{p}$. This simplifies Eq (**??**) to:

$$\max_{\boldsymbol{u}} \quad ||\mathbf{X}\boldsymbol{u}||^2 \quad \text{subj. to} \quad ||\boldsymbol{u}||^2 - 1 = 0 \tag{13.6}$$

where the constraint has been rewritten to make it amenable to constructing the Lagrangian. Doing so and taking gradients yields:

$$\mathcal{L}(\boldsymbol{u}, \lambda) = ||\mathbf{X}\boldsymbol{u}||^2 - \lambda \left(||\boldsymbol{u}||^2 - 1\right) \tag{13.7}$$

$$\nabla_{\boldsymbol{u}} \mathcal{L} = 2\mathbf{X}^\top \mathbf{X}\boldsymbol{u} - 2\lambda \boldsymbol{u} \tag{13.8}$$

$$\implies \quad \lambda \boldsymbol{u} = \left(\mathbf{X}^\top \mathbf{X}\right) \boldsymbol{u} \tag{13.9}$$

You can solve this expression ($\lambda \boldsymbol{u} = \mathbf{X}^\top \mathbf{X}\boldsymbol{u}$) by computing the first eigenvector and eigenvalue of the matrix $\mathbf{X}^\top \mathbf{X}$.

This gives you the solution to a projection into a one-dimensional space. To get a second dimension, you want to find a new vector $\boldsymbol{v}$ on which the data has maximal variance. However, to avoid redundancy, you want $\boldsymbol{v}$ to be orthogonal to $\boldsymbol{u}$; namely $\boldsymbol{u} \cdot \boldsymbol{v} = 0$. This gives:

$$\max_{\boldsymbol{v}} \quad ||\mathbf{X}\boldsymbol{v}||^2 \quad \text{subj. to} \quad ||\boldsymbol{v}||^2 = 1, \text{ and } \boldsymbol{u} \cdot \boldsymbol{v} = 0 \tag{13.10}$$

Following the same procedure as before, you can construct a La-

**Algorithm 36** PCA($\mathbf{D}$, $K$)

1: $\boldsymbol{\mu} \leftarrow \text{MEAN}(\mathbf{X})$ // compute data mean for centering
2: $\mathbf{D} \leftarrow \left(\mathbf{X} - \boldsymbol{\mu}\mathbf{1}^\top\right)^\top \left(\mathbf{X} - \boldsymbol{\mu}\mathbf{1}^\top\right)$ // compute covariance, $\mathbf{1}$ is a vector of ones
3: $\{\lambda_k, \boldsymbol{u}_k\} \leftarrow$ top $K$ eigenvalues/eigenvectors of $\mathbf{D}$
4: **return** $(\mathbf{X} - \boldsymbol{\mu}\mathbf{1})\,\mathbf{U}$ // project data using $\mathbf{U}$

grangian and differentiate:

$$\mathcal{L}(\boldsymbol{v}, \lambda_1, \lambda_2) = ||\mathbf{X}\boldsymbol{v}||^2 - \lambda_1 \left(||\boldsymbol{u}||^2 - 1\right) - \lambda_2 \boldsymbol{u} \cdot \boldsymbol{v} \tag{13.11}$$

$$\nabla_{\boldsymbol{u}} \mathcal{L} = 2\mathbf{X}^\top \mathbf{X}\boldsymbol{v} - 2\lambda_1 \boldsymbol{v} - 2\lambda_2 \boldsymbol{u} \tag{13.12}$$

$$\implies \lambda_1 \boldsymbol{v} = \left(\mathbf{X}^\top \mathbf{X}\right) \boldsymbol{v} - \lambda_2 \boldsymbol{u} \tag{13.13}$$

However, you know that $\boldsymbol{u}$ is the first eigenvector of $\mathbf{X}^\top\mathbf{X}$, so the solution to this problem for $\lambda_1$ and $\boldsymbol{v}$ is given by the *second* eigenvalue/eigenvector pair of $\mathbf{X}^\top\mathbf{X}$.

Repeating this analysis inductively tells you that if you want to project onto $K$ mutually orthogonal dimensions, you simply need to take the first $K$ eigenvectors of the matrix $\mathbf{X}^\top\mathbf{X}$. This matrix is often called the **data covariance matrix** because $[\mathbf{X}^\top\mathbf{X}]_{i,j} = \sum_n \sum_m x_{n,i} x_{m,j}$, which is the sample covariance between features $i$ and $j$.

This leads to the technique of **principle components analysis**, or **PCA**. For completeness, the is depicted in Algorithm **??**. The important thing to note is that the eigenanalysis only gives you the projection directions. It does not give you the embedded data. To embed a data point $\boldsymbol{x}$ you need to compute its embedding as $\langle \boldsymbol{x} \cdot \boldsymbol{u}_1, \boldsymbol{x} \cdot \boldsymbol{u}_2, \ldots, \boldsymbol{x} \cdot \boldsymbol{u}_K \rangle$. If you write $\mathbf{U}$ for the $D \times K$ matrix of $\boldsymbol{u}$s, then this is just $\mathbf{XU}$.

There is an alternative derivation of PCA that can be informative, based on *reconstruction error*. Consider the one-dimensional case again, where you are looking for a single projection direction $\boldsymbol{u}$. If you were to use this direction, your projected data would be $\mathbf{Z} = \mathbf{X}\boldsymbol{u}$. Each $Z_n$ gives the position of the $n$th datapoint along $\boldsymbol{u}$. You can project this one-dimensional data back into the original space by multiplying it by $\boldsymbol{u}^\top$. This gives you *reconstructed* values $\mathbf{Z}\boldsymbol{u}^\top$. Instead of maximizing variance, you might instead want to minimize the **reconstruction error**, defined by:

$$\left|\left|\mathbf{X} - \mathbf{Z}\boldsymbol{u}^\top\right|\right|^2 = \left|\left|\mathbf{X} - \mathbf{X}\boldsymbol{u}\boldsymbol{u}^\top\right|\right|^2 \qquad \text{definition of } \mathbf{Z} \tag{13.14}$$

$$= ||\mathbf{X}||^2 + \left|\left|\mathbf{X}\boldsymbol{u}\boldsymbol{u}^\top\right|\right|^2 - 2\mathbf{X}^\top \mathbf{X}\boldsymbol{u}\boldsymbol{u}^\top \qquad \text{quadratic rule} \tag{13.15}$$

$$= ||\mathbf{X}||^2 + \left|\left|\mathbf{X}\boldsymbol{u}\boldsymbol{u}^\top\right|\right|^2 - 2\boldsymbol{u}^\top\mathbf{X}^\top\mathbf{X}\boldsymbol{u} \qquad \text{quadratic rule} \tag{13.16}$$

$$= ||\mathbf{X}||^2 + ||\mathbf{X}||^2 - 2\boldsymbol{u}^\top\mathbf{X}^\top\mathbf{X}\boldsymbol{u} \qquad \boldsymbol{u} \text{ is a unit vector} \tag{13.17}$$

$$= C - 2\,||\mathbf{X}\boldsymbol{u}||^2 \qquad \text{join constants, rewrite last term} \tag{13.18}$$

Minimizing this final term is equivalent to maximizing $||\mathbf{X}\boldsymbol{u}||^2$, which is exactly the form of the maximum variance derivation of PCA. Thus, you can see that maximizing variance is identical to minimizing reconstruction error.

The same question of "what should $K$ be" arises in dimensionality reduction as in clustering. If the purpose of dimensionality reduction is to visualize, then $K$ should be 2 or 3. However, an alternative purpose of dimensionality reduction is to avoid the curse of dimensionality. For instance, even if you have labeled data, it might be worthwhile to reduce the dimensionality before applying supervised learning, essentially as a form of regularization. In this case, the question of an optimal $K$ comes up again. In this case, the same criteria (AIC and BIC) that can be used for clustering can be used for PCA. The only difference is the quality measure changes from a sum of squared distances to means (for clustering) to a sum of squared distances to original data points (for PCA). In particular, for BIC you get the reconstruction error plus $K \log D$; for AIC, you get the reconstruction error plus $2KD$.

## 13.3 Manifolds and Graphs

what is a manifold?
  graph construction

## 13.4 Non-linear Dimensionality Reduction

isomap
  lle
  mvu
  mds?

## 13.5 Non-linear Clustering: Spectral Methods

what is a spectrum
  spectral clustering

## 13.6 Exercises

**Exercise 13.1. TODO...**

# 14 | Expectation Maximization

**Learning Objectives:**
- Explain the relationship between parameters and hidden variables.
- Construct generative stories for clustering and dimensionality reduction.
- Draw a graph explaining how EM works by constructing convex lower bounds.
- Implement EM for clustering with mixtures of Gaussians, and contrasting it with $k$-means.
- Evaluate the differences betweem EM and gradient descent for hidden variable models.

Dependencies:

Suppose you were building a naive Bayes model for a text categorization problem. After you were done, your boss told you that it became prohibitively expensive to obtain labeled data. You now have a probabilistic model that assumes access to labels, but you don't have any labels! Can you still do something?

Amazingly, you can. You can treat the labels as **hidden variables**, and attempt to learn them at the same time as you learn the parameters of your model. A very broad family of algorithms for solving problems just like this is the **expectation maximization** family. In this chapter, you will derive expectation maximization (EM) algorithms for clustering and dimensionality reduction, and then see why EM works.

## 14.1 Clustering with a Mixture of Gaussians

In Chapter 7, you learned about probabilitic models for classification based on density estimation. Let's start with a fairly simple classification model that *assumes* we have labeled data. We will shortly remove this assumption. Our model will state that we have $K$ classes, and data from class $k$ is drawn from a Gaussian with mean $\boldsymbol{\mu}_k$ and variance $\sigma_k^2$. The choice of classes is parameterized by $\boldsymbol{\theta}$. The generative story for this model is:

1. For each example $n = 1 \dots N$:
   (a) Choose a label $y_n \sim \mathcal{D}isc(\boldsymbol{\theta})$
   (b) Choose example $\boldsymbol{x}_n \sim \mathcal{N}or(\boldsymbol{\mu}_{y_n}, \sigma_{y_n}^2)$

This generative story can be directly translated into a likelihood as before:

$$p(D) = \prod_n \mathcal{M}ult(y_n \mid \boldsymbol{\theta}) \mathcal{N}or(\boldsymbol{x}_n \mid \boldsymbol{\mu}_{y_n}, \sigma_{y_n}^2) \tag{14.1}$$

$$= \overbrace{\prod_n \underbrace{\theta_{y_n}}_{\text{choose label}} \underbrace{\left[2\pi\sigma_{y_n}^2\right]^{-\frac{D}{2}} \exp\left[-\frac{1}{2\sigma_{y_n}^2}\left|\left|\boldsymbol{x}_n - \boldsymbol{\mu}_{y_n}\right|\right|^2\right]}_{\text{choose feature values}}}^{\text{for each example}} \tag{14.2}$$

If you had access to labels, this would be all well and good, and you could obtain closed form solutions for the maximum likelihood estimates of all parameters by taking a log and then taking gradients of the log likelihood:

$$\theta_k = \text{fraction of training examples in class } k \tag{14.3}$$
$$= \frac{1}{N}\sum_n [y_n = k]$$
$$\boldsymbol{\mu}_k = \text{mean of training examples in class } k \tag{14.4}$$
$$= \frac{\sum_n [y_n = k]\boldsymbol{x}_n}{\sum_n [y_n = k]}$$
$$\sigma_k^2 = \text{variance of training examples in class } k \tag{14.5}$$
$$= \frac{\sum_n [y_n = k]\,||\boldsymbol{x}_n - \mu_k||}{\sum_n [y_n = k]}$$

Suppose that you *don't* have labels. Analogously to the $K$-means algorithm, one potential solution is to iterate. You can start off with guesses for the values of the unknown variables, and then iteratively improve them over time. In $K$-means, the approach was the *assign* examples to labels (or clusters). This time, instead of making hard assignments ("example 10 belongs to cluster 4"), we'll make **soft assignments** ("example 10 belongs half to cluster 4, a quarter to cluster 2 and a quarter to cluster 5"). So as not to confuse ourselves too much, we'll introduce a new variable, $\boldsymbol{z}_n = \langle z_{n,1}, \ldots, z_{n,K}$ (that sums to one), to denote a fractional assignment of examples to clusters.

? You should be able to derive the maximum likelihood solution results formally by now.

This notion of soft-assignments is visualized in Figure 14.1. Here, we've depicted each example as a pie chart, and it's coloring denotes the degree to which it's been assigned to each (of three) clusters. The size of the pie pieces correspond to the $\boldsymbol{z}_n$ values.

Formally, $z_{n,k}$ denotes the probability that example $n$ is assigned to cluster $k$:

$$z_{n,k} = p(y_n = k \mid \boldsymbol{x}_n) \tag{14.6}$$
$$= \frac{p(y_n = k, \boldsymbol{x}_n)}{p(\boldsymbol{x}_n)} \tag{14.7}$$
$$= \frac{1}{Z_n}\mathcal{M}ult(k \mid \boldsymbol{\theta})\mathcal{N}or(\boldsymbol{x}_n \mid \boldsymbol{\mu}_k, \sigma_k^2) \tag{14.8}$$

Here, the normalizer $Z_n$ is to ensure that $\boldsymbol{z}_n$ sums to one.

Given a set of parameters (the $\boldsymbol{\theta}$s, $\boldsymbol{\mu}$s and $\sigma^2$s), the **fractional assignments** $z_{n,k}$ are easy to compute. Now, akin to $K$-means, given

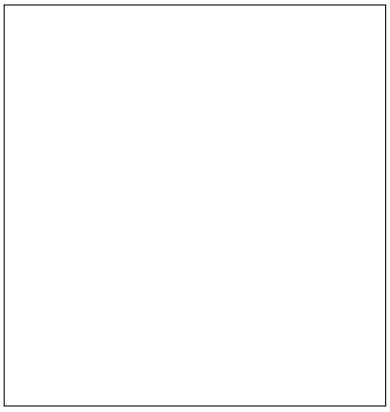


Figure 14.1: em:piecharts: A figure showing pie charts

**Algorithm 37** GMM($\mathbf{X}$, $K$)

**for** $k = 1$ **to** $K$ **do**
  $\boldsymbol{\mu}_k \leftarrow$ some random location // randomly initialize mean for $k$th cluster
  $\sigma_k^2 \leftarrow 1$ // initialize variances
  $\theta_k \leftarrow 1/K$ // each cluster equally likely a priori
**end for**
**repeat**
  **for** $n = 1$ **to** $N$ **do**
    **for** $k = 1$ **to** $K$ **do**
      $z_{n,k} \leftarrow \theta_k \left[2\pi\sigma_k^2\right]^{-\frac{D}{2}} \exp\left[-\frac{1}{2\sigma_k^2}||\boldsymbol{x}_n - \boldsymbol{\mu}_k||^2\right]$ // compute (unnormalized) fractional assignments
    **end for**
    $\boldsymbol{z}_n \leftarrow \frac{1}{\sum_k z_{n,k}} \boldsymbol{z}_n$ // normalize fractional assignments
  **end for**
  **for** $k = 1$ **to** $K$ **do**
    $\theta_k \leftarrow \frac{1}{N}\sum_n z_{n,k}$ // re-estimate prior probability of cluster $k$
    $\boldsymbol{\mu}_k \leftarrow \frac{\sum_n z_{n,k}\boldsymbol{x}_n}{\sum_n z_{n,k}}$ // re-estimate mean of cluster $k$
    $\sigma_k^2 \leftarrow \frac{\sum_n z_{n,k}||\boldsymbol{x}_n - \mu_k||}{\sum_n z_{n,k}}$ // re-estimate variance of cluster $k$
  **end for**
**until** converged
**return** $\boldsymbol{z}$ // return cluster assignments

fractional assignments, you need to recompute estimates of the model parameters. In analogy to the maximum likelihood solution (Eqs (**??**)-(**??**)), you can do this by counting fractional points rather than full points. This gives the following re-estimation updates:

$$\theta_k = \text{fraction of training examples in class } k = \frac{1}{N}\sum_n z_{n,k} \tag{14.9}$$

$$\boldsymbol{\mu}_k = \text{mean of fractional examples in class } k = \frac{\sum_n z_{n,k}\boldsymbol{x}_n}{\sum_n z_{n,k}} \tag{14.10}$$

$$\sigma_k^2 = \text{variance of fractional examples in class } k = \frac{\sum_n z_{n,k}||\boldsymbol{x}_n - \mu_k||}{\sum_n z_{n,k}} \tag{14.11}$$

All that has happened here is that the hard assignments "$[y_n = k]$" have been replaced with soft assignments "$z_{n,k}$". As a bit of foreshadowing of what is to come, what we've done is essentially replace known labels with *expected labels*, hence the name "expectation maximization."

Putting this together yields Algorithm 14.1. This is the **GMM** ("**Gaussian Mixture Models**") algorithm, because the probabilitic model being learned describes a dataset as being drawn from a mixture distribution, where each component of this distribution is a

Gaussian.

Just as in the $K$-means algorithm, this approach is succeptible to local optima and quality of initialization. The heuristics for computing better initializers for $K$-means are also useful here.

? Aside from the fact that GMMs use soft assignments and $K$-means uses hard assignments, there are other differences between the two approaches. What are they?

## 14.2 The Expectation Maximization Framework

At this point, you've seen a method for learning in a particular probabilistic model with hidden variables. Two questions remain: (1) can you apply this idea more generally and (2) why is it even a reasonable thing to do? Expectation maximization is a *family* of algorithms for performing maximum likelihood estimation in probabilistic models with hidden variables.

The general flavor of how we will proceed is as follows. We want to maximize the log likelihood $\mathcal{L}$, but this will turn out to be difficult to do directly. Instead, we'll pick a surrogate function $\tilde{\mathcal{L}}$ that's a lower bound on $\mathcal{L}$ (i.e., $\tilde{\mathcal{L}} \leq \mathcal{L}$ everywhere) that's (hopefully) easier to maximize. We'll construct the surrogate in such a way that increasing it will force the true likelihood to also go up. After maximizing $\tilde{\mathcal{L}}$, we'll construct a *new* lower bound and optimize that. This process is shown pictorially in Figure 14.2.

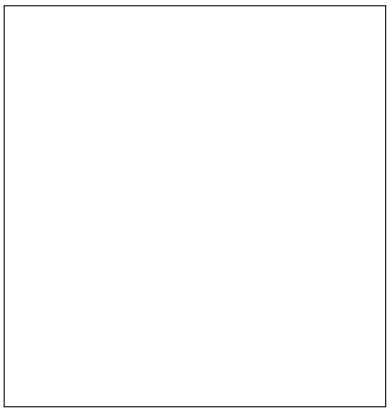


Figure 14.2: em:lowerbound: A figure showing successive lower bounds

To proceed, consider an arbitrary probabilistic model $p(\boldsymbol{x}, \boldsymbol{y} \mid \boldsymbol{\theta})$, where $\boldsymbol{x}$ denotes the observed data, $\boldsymbol{y}$ denotes the hidden data and $\boldsymbol{\theta}$ denotes the parameters. In the case of Gaussian Mixture Models, $\boldsymbol{x}$ was the data points, $\boldsymbol{y}$ was the (unknown) labels and $\boldsymbol{\theta}$ included the cluster prior probabilities, the cluster means and the cluster variances. Now, given access *only* to a number of examples $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$, you would like to estimate the parameters ($\boldsymbol{\theta}$) of the model.

Probabilistically, this means that some of the variables are unknown and therefore you need to marginalize (or sum) over their possible values. Now, your data consists only of $\mathbf{X} = \langle \boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N \rangle$, not the $(\boldsymbol{x}, y)$ pairs in $D$. You can then write the likelihood as:

$$p(\mathbf{X} \mid \boldsymbol{\theta}) = \sum_{y_1} \sum_{y_2} \cdots \sum_{y_N} p(\mathbf{X}, y_1, y_2, \ldots y_N \mid \boldsymbol{\theta}) \qquad \text{marginalization} \tag{14.12}$$

$$= \sum_{y_1} \sum_{y_2} \cdots \sum_{y_N} \prod_n p(\boldsymbol{x}_n, y_n \mid \boldsymbol{\theta}) \qquad \text{examples are independent} \tag{14.13}$$

$$= \prod_n \sum_{y_n} p(\boldsymbol{x}_n, y_n \mid \boldsymbol{\theta}) \qquad \text{algebra} \tag{14.14}$$

At this point, the natural thing to do is to take logs and then start taking gradients. However, once you start taking logs, you run into a

problem: the log cannot eat the sum!

$$\mathcal{L}(\mathbf{X} \mid \boldsymbol{\theta}) = \sum_n \log \sum_{y_n} p(\boldsymbol{x}_n, y_n \mid \boldsymbol{\theta}) \tag{14.15}$$

Namely, the log gets "stuck" outside the sum and cannot move in to decompose the rest of the likelihood term!

The next step is to apply the somewhat strange, but strangely useful, trick of multiplying by 1. In particular, let $q(\cdot)$ be an arbitrary probability distribution. We will multiply the $p(\dots)$ term above by $q(y_n)/q(y_n)$, a valid step so long as $q$ is never zero. This leads to:

$$\mathcal{L}(\mathbf{X} \mid \boldsymbol{\theta}) = \sum_n \log \sum_{y_n} q(y_n) \frac{p(\boldsymbol{x}_n, y_n \mid \boldsymbol{\theta})}{q(y_n)} \tag{14.16}$$

We will now construct a lower bound using **Jensen's inequality**. This is a very useful (and easy to prove!) result that states that $f(\sum_i \lambda_i x_i) \geq \sum_i \lambda_i f(x_i)$, so long as (a) $\lambda_i \geq 0$ for all $i$, (b) $\sum_i \lambda_i = 1$, and (c) $f$ is concave. If this looks familiar, that's just because it's a direct result of the definition of **concavity**. Recall that $f$ is concave if $f(ax + by) \geq af(x) + bf(x)$ whenever $a + b = 1$.

? Prove Jensen's inequality using the definition of concavity and induction.

You can now apply Jensen's inequality to the log likelihood by identifying the list of $q(y_n)$s as the $\lambda$s, log as $f$ (which is, indeed, concave) and each "$x$" as the $p/q$ term. This yields:

$$\begin{aligned}
\mathcal{L}(\mathbf{X} \mid \boldsymbol{\theta}) &\geq \sum_n \sum_{y_n} q(y_n) \log \frac{p(\boldsymbol{x}_n, y_n \mid \boldsymbol{\theta})}{q(y_n)} && (14.17)\\
&= \sum_n \sum_{y_n} \Big[ q(y_n) \log p(\boldsymbol{x}_n, y_n \mid \boldsymbol{\theta}) - q(y_n) \log q(y_n) \Big] && (14.18)\\
&\triangleq \tilde{\mathcal{L}}(\mathbf{X} \mid \boldsymbol{\theta}) && (14.19)
\end{aligned}$$

Note that this inequality holds for *any* choice of function $q$, so long as its non-negative and sums to one. In particular, it needn't even by the same function $q$ for each $n$. We will need to take advantage of both of these properties.

We have succeeded in our first goal: constructing a lower bound on $\mathcal{L}$. When you go to optimize this lower bound for $\boldsymbol{\theta}$, the only part that matters is the first term. The second term, $q \log q$, drops out as a function of $\boldsymbol{\theta}$. This means that the the maximization you need to be able to compute, for fixed $q_n$s, is:

$$\boldsymbol{\theta}^{(\text{new})} \leftarrow \arg\max_{\boldsymbol{\theta}} \sum_n \sum_{y_n} q_n(y_n) \log p(\boldsymbol{x}_n, y_n \mid \boldsymbol{\theta}) \tag{14.20}$$

This is *exactly* the sort of maximization done for Gaussian mixture models when we recomputed new means, variances and cluster prior probabilities.

The second question is: what should $q_n(\cdot)$ actually be? Any reasonable $q$ will lead to a lower bound, so in order to choose one $q$ over another, we need another criterion. Recall that we are hoping to maximize $\mathcal{L}$ by instead maximizing a lower bound. In order to ensure that an increase in the lower bound implies an increase in $\mathcal{L}$, we need to ensure that $\mathcal{L}(\mathbf{X} \mid \boldsymbol{\theta}) = \tilde{\mathcal{L}}(\mathbf{X} \mid \boldsymbol{\theta})$. In words: $\tilde{\mathcal{L}}$ should be a lower bound on $\mathcal{L}$ that makes contact at the current point, $\boldsymbol{\theta}$. This is shown in Figure **??**, including a case where the lower bound does *not* make contact, and thereby does not guarantee an increase in $\mathcal{L}$ with an increase in $\tilde{\mathcal{L}}$.

## 14.3 EM versus Gradient Descent

computing gradients through marginals
step size

## 14.4 Dimensionality Reduction with Probabilistic PCA

derivation
advantages over pca

## 14.5 Exercises

**Exercise 14.1. TODO...**

# 15 | Semi-Supervised Learning

**Learning Objectives:**

- Explain the cluster assumption for semi-supervised discriminative learning, and why it is necessary.
- Dervive an EM algorithm for generative semi-supervised text categorization.
- Compare and contrast the query by uncertainty and query by committee heuristics for active learning.

You may find yourself in a setting where you have access to some labeled data and some unlabeled data. You would like to use the labeled data to learn a classifier, but it seems wasteful to throw out all that unlabeled data. The key question is: what can you do with that unlabeled data to aid learning? And what assumptions do we have to make in order for this to be helpful?

One idea is to try to use the unlabeled data to learn a better decision boundary. In a discriminative setting, you can accomplish this by trying to find decision boundaries that don't pass too closely to unlabeled data. In a generative setting, you can simply treat some of the labels as observed and some as hidden. This is **semi-supervised learning**. An alternative idea is to spend a small amount of money to get labels for some subset of the unlabeled data. However, you would like to get the most out of your money, so you would only like to pay for labels that are useful. This is **active learning**.

Dependencies:

## 15.1 *EM for Semi-Supervised Learning*

naive bayes model

## 15.2 *Graph-based Semi-Supervised Learning*

key assumption
  graphs and manifolds
  label prop

## 15.3 *Loss-based Semi-Supervised Learning*

density assumption
  loss function
  non-convex

## 15.4 Active Learning

motivation
  qbc
  qbu

## 15.5 Dangers of Semi-Supervised Learing

unlab overwhelms lab
  biased data from active

## 15.6 Exercises

**Exercise 15.1. TODO...**

# 16 | GRAPHICAL MODELS

**Learning Objectives:**
- foo

## 16.1 *Exercises*

**Exercise 16.1. TODO...**

Dependencies: None.

# 17 | Online Learning

**Learning Objectives:**

- Explain the experts model, and why it is hard even to compete with the single best expert.
- Define what it means for an online learning algorithm to have no regret.
- Implement the follow-the-leader algorithm.
- Categorize online learning algorithms in terms of how they measure changes in parameters, and how they measure error.

All of the learning algorithms that you know about at this point are based on the idea of training a model on some data, and evaluating it on other data. This is the **batch learning** model. However, you may find yourself in a situation where students are constantly rating courses, and also constantly asking for recommendations. **Online learning** focuses on learning over a stream of data, on which you have to make predictions continually.

You have actually already seen an example of an online learning algorithm: the perceptron. However, our use of the perceptron and our analysis of its performance have both been in a batch setting. In this chapter, you will see a formalization of online learning (which differs from the batch learning formalization) and several algorithms for online learning with different properties.

Dependencies:

## 17.1 Online Learning Framework

regret
  follow the leader
  agnostic learning
  algorithm versus problem

## 17.2 Learning with Features

change but not too much
  littlestone analysis for gd and egd

## 17.3 Passive Agressive Learning

pa algorithm
  online analysis

## 17.4 *Learning with Lots of Irrelevant Features*

winnow
  relationship to egd

## 17.5 *Exercises*

**Exercise 17.1. TODO...**

# 18 | Structured Learning Tasks

–

**Learning Objectives:**
- TODO...

```
- Hidden Markov models: viterbi
- Hidden Markov models: forward-backward
- Maximum entropy Markov models
- Structured perceptronn
- Conditional random fields
- M3Ns
```

## 18.1 Exercises

**Exercise 18.1. TODO...**

Dependencies:

# 19 | Bayesian Learning

**Learning Objectives:**
- TODO...

## 19.1 *Exercises*

**Exercise 19.1. TODO...**

Dependencies:

## Code and Datasets

| Rating | Easy? | AI? | Sys? | Thy? | Morning? |
|---|---|---|---|---|---|
| +2 | y | y | n | y | n |
| +2 | y | y | n | y | n |
| +2 | n | y | n | n | n |
| +2 | n | n | n | y | n |
| +2 | n | y | y | n | y |
| +1 | y | y | n | n | n |
| +1 | y | y | n | y | n |
| +1 | n | y | n | y | n |
| 0 | n | n | n | n | y |
| 0 | y | n | n | y | y |
| 0 | n | y | n | y | n |
| 0 | y | y | y | y | y |
| -1 | y | y | y | n | y |
| -1 | n | n | y | y | n |
| -1 | n | n | y | n | y |
| -1 | y | n | y | n | y |
| -2 | n | n | y | y | n |
| -2 | n | y | y | n | y |
| -2 | y | n | y | n | n |
| -2 | y | n | y | n | y |

# NOTATION

# BIBLIOGRAPHY

Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408, 1958. Reprinted in *Neurocomputing* (MIT Press, 1998).

# INDEX