A Tour of

Machine Learning

Version 2018-02-11

by: Michael S. Gashler

Table of Contents

1 Preface

1.1 Contributors

Well, no one besides the main author has directly contributed any content to this book yet, but you could be the first. Just think, your name could be here! All you need to do is make a worthwhile improvement and send it to me.

1.1.1 How to edit this book

Here are some questions and answers about how to edit this book.

- ***What software do you use to write this book?*** My copy is in LibreOffice format. Be sure to get that version if you plan to contribute your changes back into my copy.
- ***How did you make the equations?*** LibreOffice has a LaTeX plug-in.
- ***How did you make the figures and diagrams?*** Mostly with Inkscape.
- ***Where are your references for all this material?*** I initially wrote this book in a rapid brain-dump manner, using other sources only indirectly. Of course, those other sources should still be cited. I am working to find references for all my material, but this effort may take some time. I would appreciate any help in this effort.

1.2 License of this book

This book is distributed under the CC0 license. For more information, see http://creativecommons.org/publicdomain/zero/1.0

1.2.1 Q and A about this license

The following are some questions and answers about this choice of license:

- ***Why are you using the CC0 license?*** Science is a collection of free and verifiable information. As a scientist, I seek to add to this collection.
- ***May we publish and promote a printed version of your book,***

without giving you any royalties? I hope you make a fortune.

- ***Can I make derivative works from your book?*** Yes, I deliberately picked this license to encourage derivative works. Please use as much or as little as you deem suitable for your project. There are no strings attached. I only hope you will pay it forward.
- ***Does this license mean I can simply put my name on your book and tell people I wrote it?*** Legally, yes. Ethically, no. CC0 is only a legal license. It grants you legal permissions, but it will not save you from embarrassing yourself if you attempt to take credit that you have not earned.
- ***If you want attribution, then shouldn't you be using the CC-BY license, which requires attribution?*** No. Among other things, this work is an attempt to test the hypothesis that science can advance just as effectively (and possibly more effectively) without the involvement of any legal mechanisms.
- ***Can I make changes to this book without releasing them under a CC0 public domain dedication?*** You can make your own fork with the license of your choice, however, my copy only accepts contributions that are explicitly released under a CC0 public domain dedication.

1.3 Layout of this book

Machine learning has grown into a very big field, perhaps too big to reasonably permit teaching it as a single subject. At a high level, I see machine learning beginning to self-organize into three major camps of philosophy. There is significant interdependence among these camps. Often, students will become emphatically converted to one of these camps, will feel somewhat neutral about one, and develop strong feelings that one of them is the byproduct of misguided exuberance. As their understanding matures, they begin to see how principles in one camp provide significant insight into being effective in another. Finally, as genuine expertise begins to coalesce, they will begin to see significant value in all three philosophies.

These three major camps are:

1. The data miners rely predominantly on the plethora of data-driven

machine learning techniques. They generally focus on finding ways to obtain higher predictive accuracy.

2. The statisticians study sampling and Bayesian methods. They are largely interested in finding correct theory, and doing things the right way.
3. The connectionists study brain-inspired approaches, such as neural networks. They are largely interested in giving machines more human-like capabilities for thinking and learning.

I have tried to structure this book at a high-level to mirrors these three camps. Within each major camp, I lightly survey relevant concepts in the order that makes the most sense to me. Sometimes, this structure forces me to explain the same concept multiple times for different applications. I do not consider this to be a problem. Often, students will not fully grasp the elegance of a concept until they learn it for the n^{th} time from the n^{th} perspective. Also, it is often easier for a reader to skip a little redundancy than to find missing explanations.

There are, of course, many other fields that have significant overlap with machine learning. For example, computer vision, natural language processing, information retrieval, bioinformatics, artificial intelligence, robotics and control theory all rely heavily on machine learning techniques, and they each bring their own focuses, tricks, and biases to the field. I will not even attempt to cover any of these or other related areas in this book.

In my opinion, it is not really possible to do justice to the field of machine learning in any fewer than six full-semester courses: one to survey each of the three major camps, and one to add some depth to each of them. More courses would be better. Unfortunately, practical limitations in my department necessitate that we make do with fewer courses than would be ideal. As an outlet for my frustration at having to operate under practical limitations, I went and did something quite impractical for an assistant professor to do: I wrote this book. It may end up costing my career in academics—we shall see.

2 Data-driven learning methods

2.1 Basic concepts

2.1.1 The value of knowledge

Not all knowledge has equal value. Knowing historical trivia will not help you make as effective decisions as knowing about significant historical trends. Knowing about yesterday's stock prices is certainly not as valuable as knowing tomorrow's stock prices. Knowing how to make someone happy is generally more valuable when that person is around than when that person is far away. Knowledge that is likely to show up on a future test is certainly more valuable than knowledge that you know will never be tested. So, what makes some knowledge more important that other knowledge? What exactly gives knowledge, or the information in which we attempt to encode it, its value?

Surely, the answer to this question has something to do with how we use it. All of these situations could be summarized by saying, *knowledge we will use has more valuable than knowledge we will not use*. So next, let's ask the natural follow-up questions: What do humans use knowledge for? Why are we so hungry for more? What do we plan to do with it?

Humans use knowledge to make decisions. To a large extent, we are planning machines. We constantly attempt to foresee the consequences of choices that we can make. Why do we do that? Because, if we can predict the consequences of choices, then we can choose the consequences that we want. The value of knowledge is its utility for influencing decisions. In other words, its value is determined by its usefulness in making accurate predictions.

So, can we quantify this? Can we assign a number to pieces of knowledge? Can we sort pieces of knowledge by their value? In general, the answer to this question is “no”, because it is difficult to know all the potential uses for any piece of knowledge.

When I was younger, I complained about how much time educators “wasted” teaching me math skills with no obvious practical value! Now, many years later, I am a scientist. How grateful I am that the people in charge of the curriculum had the foresight to know that math was the gateway to the world of all quantified things! At the time, I did not evaluate very many things quantitatively, so math seemed to have little value to me. Now, it has great value to me.

As technology progresses, we are able to evaluate more and more things quantitatively. This trend is increasing the precision of our predictions, and thus, is making the human race as a whole more intelligent.

Although it may be impractical to attempt to quantify the value of a piece of knowledge by itself, the problem becomes much easier if we pair it up with a prediction task. The following example illustrates this difference:

Suppose you attend a university where 58% of the students are male, and 42% are female. What is the value of knowing this bit of trivia? That is indeed difficult to determine. Now, suppose you are in a public place on campus, and your friend offers to buy you lunch if you can predict the sex of the next person who walks through a certain door. Suppose lunch costs $8. Now, what is the value of that knowledge?

Well, if you didn't have this knowledge, you might predict male or female with equal probability. Thus, you might say the value of having a friend who randomly puts lunch on the line is at least
0.5 * $8 = $4. (Of course, this is completely ignoring other potential sources of value, but let's keep it simple for now.)

If you have the additional knowledge that 58% of the students at this university are male, then it makes sense to predict that a *male* will walk through the door next. Assuming this actually gives you a 58% chance of being right, then the value of having this additional knowledge in this situation is at least (0.58 * $8) - $4 = $0.64.

Suppose you did not already have this additional knowledge, but someone offered to sell it to you for $0.50. Should you buy it? Of course, that really depends on the circumstances. Could you get the same information elsewhere for less? Would your friend be annoyed to see you using other resources in attempt to obtain his or her money? And, how could you possibly evaluate the value of the information if you did not yet have it? But if we look past all such complications, $0.50 is a good deal for information that is worth $0.64 in this situation alone.

Often, real-world circumstances are much too complicated for us to evaluate the exact monetary value of information, but quantitative methods are still useful for comparing them. Suppose you wish to compare two pieces of information. Let's call these pieces of information A and B. Suppose A improves your ability to predict a certain thing of interest by 4%, and B improves your ability to predict that thing of interest by 6%. You may not be able to say what is the exact value of either A or B with this prediction task, but you can trivially determine that B has more value than A with this prediction task. Now, suppose you use both A and B together, and they improve your ability to predict the thing of interest by exactly 6%. What does that tell you? It tells you that B encapsulates all of the value found in A. By contrast, if they improve your predictions by 8% when used together, then this tells you that A offers some additional perspective on the situation that B does not include.

2.1.2 Automated learning

Learning is the ability to improve with experience. Learning involves gathering knowledge, and using it to improve predictions. Arguably, gathering knowledge and using it to predict the consequences of possible choices is an essential part of what makes humans intelligent. Let us define *intelligence* as the ability to solve problems. If we can predict the effects of our choices, then we can make the choices that lead to solving the problem.

Interestingly, humans are not the only entities that gather knowledge and use it to make predictions. Governments do it. Many companies do it. And they are doing it to a much greater extent than ever before. Even fully-automated computer programs can improve with experience.

This book studies the mechanisms by which machines can learn. To some extent, humans, governments, and companies may be thought of as types of machines. Hence, many of the concepts involved in machine learning are applicable to learning in general.

2.1.3 The baseline algorithm

Suppose there was a machine that could reliably predict whether the value of a certain stock would increase or decrease the following day. What would you pay for such a machine if it always made correct predictions? How much would you pay if it always made incorrect predictions?

Be careful in answering this question. At first impulse, many people are often inclined to suppose that a machine that is always wrong is worthless. In fact, such a machine would really have no less value than the machine that always made correct predictions. To use it effectively, you would simply need to observe what it predicts, and know that the opposite must be right. In fact, such a machine would be extremely useful. Although it may seem counter-intuitive at first, when the answer is binary, it is just as difficult to be consistently wrong as it is to be consistently right. The machine that is consistently wrong must utilize just as much intelligence as the machine that is consistently right to achieve such impressive reliability.

Now, suppose we had a machine that always predicted that the value of the stock would increase the following day. This could be a very simple machine. In fact, it need not even have the ability to make a pessimistic prediction. It might simply be the word "increase" written in permanent ink on a piece of paper. What would be the value of this static machine? If such a machine did have significant value, then everyone would have one, because they are so very easy to build. But, alas, such a machine truly contributes no intelligence whatsoever.

But, let us be careful here in our assertions. What exactly do we mean by saying, it *contributes no intelligence?* The stock market has a slight tendency to climb slowly and crash quickly. It also exhibits a general overall trend of climbing. Therefore, the machine that always predicts "increase" will be right more often than it is wrong. This might actually be slightly useful information to a person who did not already know that. So, why do we say that it *contributes no intelligence?* Because it is static. It did not utilize any information in making its prediction. Intelligence requires doing some amount of computation in order to solve a problem, not simply waiting for the right problems to be thrown at you.

Perhaps there are some terminology spaces that might call a static machine "intelligent" if it makes predictions that are better than random. In the terminology of this book, however, we will define intelligence such that static machines exhibit exactly zero intelligence. For example, this book exhibits no intelligence, even though it contains such wonderfully valuable information, and is extremely useful ...and well-written ...and the author is such a smart person!

In order to evaluate the intelligence of other learning machines, it is useful to have one that exhibits exactly zero intelligence. We will call this machine, or algorithm, *baseline*. (Note that sometimes this term is used with other meanings, so it may be necessary to define what exactly is meant by the term *baseline* when communicating with others.) In this book, *baseline* is an algorithm that makes the best predictions that are possible for a static machine to make.

Hence, if baseline is used to predict whether the stock market will rise or fall, it will always predict that it will rise. If it is used to predict a continuous value, it will always predict the mean. So, if you use baseline to predict how much the stock market will change in one day, it will always predict the mean change, which is probably very close to zero.

Even though baseline completely lacks intelligence, it is not the worst algorithm one can use to make predictions. For example, consider a machine that randomly predicts “increase” with probability 0.5, and “decrease” with probability 0.5. Over time, this machine will achieve an average accuracy 50%. This is slightly worse than the average accuracy that baseline will achieve.

Interestingly, it is possible for an algorithm to achieve lower accuracy than baseline, and yet still contribute some intelligence. It is even possible to combine predictions from several such algorithms to achieve predictions that are very accurate. (We will discuss these concepts in more detail in later chapters.) So, if you have an algorithm that does worse than baseline, that does not necessarily mean your algorithm lacks intelligence or even that it lacks value. It just means that you have not yet demonstrated the value of your algorithm, since one does better that completely lacks intelligence and is trivial to implement.

2.1.4 Types of learning

Learning techniques are often broadly divided into two categories: *supervised* and *unsupervised*. Supervised learners are those that learn from examples. As you might guess, unsupervised learners are those that learn without examples.

An example of supervised learning might be an application designed to diagnose medical conditions. Patients enter their symptoms into a computer. For a while, a human medical doctor provides the diagnosis. Eventually, the computer program learns to respond the same way the doctor probably would. We call this supervised learning because the computer program is designed to learn from the examples provided by the human doctor. In other words, the doctor *supervises* the training of the computer.

An example of unsupervised learning might be an application designed to discover previously unrecognized diseases. Perhaps, it might be discovered that people who receive frequent impacts to their heads also have memory problems. Perhaps, it might be observed that people who drink water from a certain source also frequently report the symptom of diarrhea, but surprisingly tend to live longer than people who drink water from other sources. Perhaps a correlation is found between eating a lot of sugar and lacking desire to exercise. By finding natural groupings in the data, it might be observed that a only a small number of conditions are actually necessary to explain a large set of observed symptoms. Humans may be required to give meaningful names, or *labels*, to the conditions that are discovered, but it is quite possible to discover their existence without supervision.

The data structures that a learning algorithm uses to represent its knowledge is called a *model*. The process of learning values for the model is called *training*. The data that is used for training is called the *training data*, or *training set*. In supervised learning, training data consists of two parts: example inputs and example outputs. The inputs are also called *input features*, or collectively, a *pattern* or *feature vector*. The outputs are also called *labels*, or collectively, a *label vector.* (To be slightly more precise, the term *feature* refers to the portion of data that will be used for learning. Sometimes features are obtained by preprocessing the raw data before presenting it to a learning algorithm. In these cases, the features are only derived from the data. As learning algorithms become more capable, however, it is increasingly considered better to present all of the available information to the learning algorithms, so *features* are becoming somewhat synonymous with *data*.)

In unsupervised learning, the training data consists only of features, and typically contains no labels. We sometimes refer to these as *samples,* or *data points*. If we were doing supervised learning, then the term *sample* would refer to both the pattern or feature vector with its corresponding label. If the data is stored in a database table, then this is sometimes called a *row*. Unfortunately, when mathematicians use machine learning, they often store patterns in column vectors, so this can be a little bit confusing. For consistency, this book will use the database terminology, so a *row* contains a sample or data point, which consists of both a pattern and its label (or label vector).

The training data for the Virtual Doctor example might look something like this. (Note: This is bogus data. Really useful training data for this purpose would probably require a much bigger table than this.)

Hives?	***Itchy?***	***Dry mouth?***	***Sleepy?***	***Condition***
Y	N	Y	N	Appendicitis
Y	N	N	Y	Hypochondria
N	Y	Y	N	Hypochondria
N	Y	Y	Y	Dehydration
Y	N	Y	Y	Appendicitis
Y	Y	Y	N	Dehydration

In this case, each row represents one patient. The left-most 4 columns contain the input features. The right-most column contains the output label. So, each patient is represented by a pattern of size 4. In this case, the label vector has a size of 1. The term *feature space* refers to the set of all possible feature vectors. And, *label space* refers to the set of all possible label vectors.

Sometimes, the columns are called *dimensions*, so in this example, the feature space consists of 4 dimensions, and the label space consists of only one dimension. Sometimes the columns are called *attributes*. This is yet another term that is synonymous with dimension or column.

Why don't we all just agree on some consistent terminology? Well, machine learning was formed by the collision of many different disciplines, and it is still growing to engulf several others. Every time it eats another discipline, it brings in a new group of people that use different terms. So, being flexible with terminology is an important attribute (*snicker) of someone who can operate effectively in the field of machine learning.

When we are trying to understand a thing, it is often helpful to pretend that very clear boundaries exist. With learning techniques, however, there is not always a clear boundary between supervised and unsupervised methods. Some techniques are flexible enough to be used for both supervised and unsupervised learning. Other techniques perform operations that are not easily labeled as either *supervised* or *unsupervised*. In my opinion, it is counter-productive to separate them into different areas for study. I think that students who study both paradigms together are more likely to develop a thorough understanding of available learning methods. So, this book deliberately presents them together.

2.1.5 Types of data

In the previous example, the *values* or *elements* in the training data are *categorical*, as opposed to *continuous*. Categorical values, also called *nominal values*, are drawn from a finite set of possible values. Here are some examples of categorical attributes:

- sex={female, male}
- race={African, Asian, European, Latin, Native American}
- response={yes, no, not applicable}

Categorical values typically have no implicit ordering. By contrast, *continuous* values (also called *real* values or *floating-point* values) do have an implicit ordering. Here are some examples of continuous values:

- seconds=0.000237
- height=63.2

There are many other possible types of data, but these two types are the most common. When other types of data are encountered, the typical approach is to try to convert them to categorical and/or continuous values, then process these with one of the many algorithms designed to handle these types of data.

For example, suppose you have an attribute with values that have an implicit ordering, but do not permit fractional values. Such values are called *discrete*, *integer*, or *numeric*. Examples:

- number of wings = 4

- number of children = 2
- number of spots = 5

Since integers are a subset of real values, you can naturally use integer values as inputs into an algorithm that is designed to operate on real inputs. If your algorithm outputs real values and you need it to output integer values, all you need to do is round the output to the nearest integer value.

2.1.6 Architecture of supervised learners

Supervised learning typically involves training a *model*. A model can be thought of as a function or program that can be trained to map from a feature vector to a corresponding label vector. After the model is trained, it is called a *trained model*, or a *hypothesis*. We might write it in math form, $\mathbf{y} = h(\mathbf{x})$. In that case, $\mathbf{x}$ is the feature vector, $\mathbf{y}$ is the label vector, and h is the hypothesis. We might also visualize h as a machine or part of a system, $\mathbf{x} \rightarrow \boxed{h} \rightarrow \hat{\mathbf{y}}$ where $\mathbf{x}$ goes in, and $\mathbf{y}$ comes out. The set of all possible hypotheses is called the *hypothesis space*, or sometimes the *model space*. (Hypotheses is the plural form of hypothesis. Hypotheses is pronounced high-POTH-uh-seez, and hypothesis is pronounced high-POTH-uh-siss.)

In the simplest cases, the model is expected to be fully trained before it is ever used. This is called *batch training*. If supervised learning is implemented in an object-oriented language, it might implement an interface something like this:

```
Interface SupervisedLearner
{
	abstract void train(Data& features, Data& labels);

	abstract void predict(vector& in, vector& out);
}
```

Each instance of this interface represents a model that can be trained. The *train* method is called to train the model. This method takes two parameters of type *Data*, which is a matrix or a two-dimensional table of values. It is expected that *features* and *labels* both have the same number of rows. Each row in *features* is a sample input vector, and each row in *labels* is the corresponding label vector.

The *predict* method makes a prediction. It would be an error to call this method before *train* is ever called. The user passes an feature vector as the parameter *in*. The parameter *out* is a buffer that this method will use to store the predicted label vector. (You could return the label vector instead of using an out-buffer, but in many languages this ends up being less efficient. Some poorly-designed machine learning toolkits only support one-dimensional labels, but it is better to plan to handle labels of arbitrary size, because there are many problems where more than one label dimension is needed.) The *in* vector must have the same number of elements as columns in the *feature* table or matrix that was passed to *train*, and the *out* vector will have the same number of elements as columns in the *label* table or matrix that was passed to *train*.

Example training data to train a model to predict the value of a car based on some of its features:

Features			Labels
Doors	***Color***	***Odometer***	***Value***
2	Red	12200	96000
4	Brown	258221	4500
4	Blue	68420	24000
6	Brown	48750	35000
2	Brown	134229	8699
4	Red	12200	17650

Each learning algorithm implements the *train* algorithm in a different way. Let's examine how the baseline algorithm would train on this data:

Remember, the baseline algorithm always predicts the mean value for continuous label attributes, and the most-common value for categorical label attributes. In this case, there is only one label attribute, and it is continuous. Ideally, it would be best to use the mean value of all cars that our model will ever be used to predict. Unfortunately, that information is not available, so we will estimate it by computing the mean value of all the cars in the training data.

So, the train method of the Baseline learning algorithm would simply compute the mean value of the one label attribute. In this case, it is 30974.833333333. This value is the trained model of this algorithm. Most learning algorithms have more complex models, but this is all Baseline needs to make its predictions.

Now that our model is trained, we can use it to make predictions. So, suppose you see a Blue car for sale. It has 6 doors and the odometer reads 89200. What is the value of this car? Well, let's ask our trained model. So, we pass the feature vector,

```
6,	Blue,	89200
```

in to the predict method of our trained model. Baseline will just ignores the inputs and return the out vector,

```
30974.833333333.
```

Is this a reliable estimate of the value of this car? Well, Baseline is (intentionally) not a very good algorithm. But, what if we used a better learning algorithm? Would the prediction be good then?

Well, these three features are probably not sufficient to estimate the value of a car with very much precision. No matter how well-designed your learning algorithm may be, it cannot possibly make good predictions if you don't give it good information to work with. If we really wanted to train a machine to predict the value of a car, we would probably need to give it a few more features than this. Furthermore, we only gave it 6 training samples. That is not very much. How much is enough? Well, that really depends on the problem. There are some really simple problems for which 30 samples might be just enough. There are some hard problems for which millions of samples do not even come close to being enough. But 6 samples is definitely not enough for this problem.

There is a saying commonly repeated in machine learning circles: “Garbage in, garbage out”. It means, *if you do not have good training data, you cannot expect good predictions, no matter how well-designed your algorithm may be.*

2.1.7 Measuring accuracy

The best way to evaluate a hypothesis is to measure its accuracy empirically. For continuous label values, we can define the term *error* to refer to the difference between the prediction and the correct label,

$$e = y - h(\mathbf{x}).$$

This error value may be positive or negative. Values close to zero indicate that the hypothesis has made a good prediction. For categorical values, it is common to use Hamming distance, which uses 0 for correct predictions and 1 for incorrect predictions.

$$e = \begin{cases} 0, & \text{if } y = h(\mathbf{x}) \\ 1, & \text{if } y \neq h(\mathbf{x}) \end{cases}$$

These error values tell us how badly a hypothesis performs with a single pattern. To evaluate a hypothesis in general, we aggregate the error scores over an entire dataset containing n patterns. There are many metrics we can use to aggregate error values:

Mean absolute error (MAE) computes the average of the absolute values of the error scores,

$$\text{MAE} = \frac{1}{n} \sum_i |y_i - h(\mathbf{x}_i)|.$$

“Σ” is the uppercase Greek letter *sigma*. Although it looks somewhat like the letter “E”, it sounds like it begins with the letter “S”. Thus, it is often used in math equations to mean “sum”. In this equation, we sum over the rows in the training set. We represent each row index with the letter i, such that $\mathbf{x}_i$ is an input vector in row i, and y_i is the corresponding label for that row. The vertical bars represent absolute value. Examples:

$$|-1.234| = 1.234,$$
$$|2| = 2,$$
$$|-4| = 4.$$

Sum squared error (SSE) computes the sum of the squared error,

$$\text{SSE} = \sum_i (y_i - h(\mathbf{x}_i))^2.$$

If there are multiple dimensions in the label vector, then it is computed as the sum of the squared magnitude of the differences,

$$\mathrm{SSE} = \sum_i ||\mathbf{y}_i - h(\mathbf{x}_i)||^2.$$

The double vertical bars represent *magnitude*, where

$$||\mathbf{a}|| = \sqrt{a_1^2 + a_2^2 + \cdots + a_m^2}.$$

(In the case of a scalar, magnitude is the same as absolute value.)

Mean squared error (MSE) computes the average of the squared error,

$$\mathrm{MSE} = \frac{1}{n}\mathrm{SSE} = \frac{1}{n}\sum_i ||\mathbf{y}_i - h(\mathbf{x}_i)||^2.$$

(Note that n does not consider the number of dimensions in the label vector, so MSE will typically be bigger in cases where the label vector has multiple dimensions. If all the label dimensions had approximately the same meaning, then it might make sense to also divide by the number of label dimensions, but then that would be a different metric, so we would have to call it something other than MSE.)

Root mean squared error (RMSE) is the square root of MSE,

$$\begin{aligned} \mathrm{RMSE} &= \sqrt{\mathrm{MSE}} \\ &= \sqrt{\frac{1}{n}\sum_i ||\mathbf{y}_i - h(\mathbf{x}_i)||^2}. \end{aligned}$$

Notice how similar the formula for RMSE is to the formula for Euclidean distance. Because of this similarity, RMSE has several desirable properties. Consequently, it is probably the most-common metric of choice. We will discuss some of these properties in later chapters.

Since $h(\mathbf{x})$ is a predicted estimate of y, these equations are sometimes written with the symbol $\hat{y}$ in place of $h(\mathbf{x})$. The ^ symbol above the y is called a “hat”, and it means *estimated*. So, the formula for RMSE can also be written as,

$$= \sqrt{\frac{1}{n}\sum_i ||\mathbf{y}_i - \hat{\mathbf{y}}_i||^2}.$$

Predictive error is the ratio of incorrect predictions over total predictions.

(This metric is generally only suitable for problems that involve a one-dimensional categorical label.)

Predictive accuracy is the ratio of correct predictions over total predictions.

There are other metrics for evaluating a hypothesis, but these will suffice for our purposes in this chapter.

2.1.7.1 Importance of testing with out-of-band data

Some learning algorithms (such as certain instance-based learners) store the entire training data as a component of their hypotheses. If we use the training data to evaluate these hypotheses, the total error may be zero. There is no possible error score better than zero! Does this mean that the hypothesis ideal?

No. If we test such hypotheses on other data, they no longer make perfectly accurate predictions. This situation might be analogous to giving students the answer key to a test before administering that test. They will probably score well, but unfortunately, this score is not a very good indicator of how effectively the students will be able to solve other related problems.

Even learning algorithms that do not explicitly store any part of the training data will typically perform better when they are tested on the training data than when they are tested on some other data. When we evaluate a hypothesis, therefore, it is essential to test it on data that was not used to train it. The evaluation will only be an accurate prediction of future performance to the same extent that the evaluation was performed with data that was as independent of the training data as the patterns that will be evaluated in the future.

When a learning algorithm is used to predict labels for feature vectors that it has never seen before, we call this *generalization*. Measuring accuracy with the training data does not measure ability to generalize. Since the ability to generalize is usually the reason we train learning algorithms in the first place, that ability is usually what we want to measure.

When a learning algorithm cannot effectively predict labels for the training data, we call this condition *underfit*. When the learning algorithm can predict the labels on the training data well, but cannot generalize well, this condition is called *overfit*. Much of the theory in machine learning focuses on seeking to fit models to data without overfitting, so they will generalize well.

Data that is not part of the training data is often called *out-of-band* data, a *hold-out set*, or *test data*. It is a common practice in machine learning to take all of the available labeled data, and divide it up into portions. One portion is used for the training set, and another portion is used for the test set.

2.1.7.2 *n*-fold Cross-validation

Labeled data is not always easy to obtain. Often, the labels are given by human experts, and human experts tend to be busy, expensive, and slow. For example, if you wanted to make a program that diagnoses medical conditions, it might be easy to find people who are willing to tell you their symptoms for a free diagnosis, but it might be much more difficult to find a medical doctor who is anxious to volunteer to help you automate his job of performing the diagnoses.

If you don't have much labeled data to begin with, then dividing the data into a training set and a test set will only exacerbate the problem of not having enough to train with. *n*-fold cross-validation is a good solution to this problem. It works like this:

1- Divide your data (including both inputs and outputs) into *n* portions of approximately equal size, called *folds*. In this example, we will suppose that *n*=3. (This is a different *n* than we used earlier to represent the number of data points. Now, we are using *n* to represent the number of folds, which must be ≤ the number of data points. I would have chosen a different letter, but it is so common to use the letter *n* for both of these purposes that I think it is better to just warn you about it.)

2- Withhold just one fold, and train on all the other (*n*-1) folds. Test your newly trained model using just the one fold that was withheld.

3- Repeat step 2 until you have tested with each fold.

and then...

If *n* is set to the number of data points, then it is called *leave-one-out cross-validation*, since the test set will always have a size of 1.

n-fold cross-validation can be performed using any of the metrics we discussed for evaluating a hypothesis. For example, if you are using SSE, you would sum all of the squared-errors in each fold to obtain a SSE score over the entire dataset. If you are using RMSE, you would calculate SSE over all of the folds, then divide by the total number of data points (including all folds), and then compute the square root of this value.

Note, however, that *n*-fold cross-validation tests a different hypothesis with each fold. So, the final score does not reflect the effectiveness of any single hypothesis. Rather, it should be thought of as a sampling technique for evaluating the effectiveness of a learning algorithm.

A common use case for *n*-fold cross-validation is to decide which of two learning algorithms is best-suited for a particular task. For example, suppose you make a modification to a learning algorithm, and you want to know whether your modification made it better for some task. You could perform cross-validation with and without your modification. If your modification improved the algorithm (for the task represented by your data), then the results of cross-validation should be better.

2.1.7.2.1 Numerical example

Let's perform 3-fold cross-validation with the baseline algorithm on the example data (about cars). We will use RMSE for our evaluation metric.

In the first fold, we train our algorithm on all but the first two rows. Since the baseline algorithm just computes the average label value for its hypothesis, we just need to compute this average now:

$$\frac{24000+35000+8699+17650}{4}=21337.25.$$

Next, we test this hypothesis using the two samples that were withheld from training:

Sample 1:
Target = 96000
Prediction = 21337.25
Error = 96000 – 21337.25 = 74662.75

Sample 2:
Target = 4500
Prediction = 21337.25
Error = 4500 – 21337.25 = −16837.25

(Note that most learning algorithms will predict a different label vector for each feature vector, but Baseline is unique in that it always makes the same prediction.)

Fold 1 is now complete, so we move on to fold 2. In this fold, the training data consists of the first two and last two samples. We use these to compute the new hypothesis:

$$\frac{96000 + 4500 + 8699 + 17650}{4} = 31712.25.$$

And, we evaluate this hypothesis using the two samples that were withheld:

Sample 3:
Target = 24000
Prediction = 31712.25
Error = 24000 – 31712.25 = −7712.25

Sample 4:
Target = 35000
Prediction = 31712.25
Error = 35000 – 31712.25 = 3287.75

Then, we do it again for the third fold:

$$\frac{96000 + 4500 + 24000 + 35000}{4} = 39875.$$

Sample 5:
Target = 8699
Prediction = 39875
Error = 8699 – 39875 = –31176

Sample 6:

Target = 17650
Prediction = 39875
Error = 17650 – 39875 = −22225

Finally, we compute the score:

$$
\begin{aligned}
\mathrm{SSE} = {} & 74662.75^2 + (-16837.25)^2 + \\
& (-7712.25)^2 + 3287.75^2 + \\
& (-31176)^2 + (-22225)^2 = 7394200926.25.
\end{aligned}
$$

$$\mathrm{RMSE} = \sqrt{\frac{7394200926.25}{6}} \approx 35105.0826.$$

2.1.7.2.2 Shuffling data

Sometimes, datasets are not provided in a completely random order. This can be problematic for training and evaluating a model. For example, suppose your labeled data consists of a set of symptoms that a medical doctor has labeled with a diagnosis. Suppose all of the *hay-fever* cases are presented first, and all of the *cancer* cases are presented next. You divide this data into a training set and a test set. If you train on the portion where most patients are diagnosed with hay-fever, the learning algorithm will probably learn to predict *hay-fever* in most cases. Now, when you test this hypothesis using the other part of the data, it will incorrectly diagnose most of the patterns.

A simple solution to this problem is to shuffle the rows in the data before dividing it into portions or folds. Here is a simple algorithm to shuffle all the rows in a dataset.

```
for $i$ from $n$ down to 2
    $r$ = random($i$)
    swap row $i$-1 with row $r$
```

In this algorithm, the function “random(*i*)” returns a random integer value with probability uniformly distributed from 0 to *i*-1. (Note that whenever you swap two feature vectors, the corresponding label vectors must also be swapped with them, such that corresponding features and labels stay together.)

2.1.7.2.3 Folds and repetitions

In order to increase the accuracy estimated by cross-validation, it is common to perform several repetitions of cross-validation and average the results together. The rows are shuffled before each repetition.

The best number of repetitions and folds to use depends on the data. If plenty of data is available, (usually tens-of-thousands of samples), then a common choice is 5x2 cross-validation. This means that 5 repetitions of 2-fold cross-validation are performed.

When not a lot of data is available, (perhaps a few hundred samples), then 1x10 cross-validation is typically used instead. This means that only one repetition of 10-fold cross-validation is used.

When lots of data is available, few folds and more repetitions is a better way to go. If too many folds are used, there is a slight tendency to underestimate the accuracy of the learner. Unfortunately, the variance is higher with 2-fold cross-validation, so it is necessary to use more repetitions to get a good estimate. When not so much data is available, more folds are necessary in order to ensure that enough training data is used to train the model.

In general, using more repetitions will take longer, but will lead to better accuracy. If you have time to spare, performing more repetitions is almost always a good idea.

2.1.8 Architecture of unsupervised learners

Unsupervised learning algorithms tend to be somewhat more diverse than supervised learning algorithms. Consequently, it is not easy to provide a general-purpose interface for unsupervised learning.

The only thing that they all have in common is that they transform a dataset in some way. So, a very general architecture for unsupervised learning might look like this:

```
Interface UnupervisedLearner
{
    abstract void transform(Data& in, Data& out);
}
```

Some, but definitely not all, unsupervised operations utilize an internal model. These operations can be divided into a training phase, which constructs the model, and a transforming phase, which uses the model to transform the data. So, a more useful interface might be:

```
Interface UnupervisedLearner
{
	abstract void train(Data& trainingdata);

	abstract void transform(Data& in, Data& out);
}
```

For unsupervised operations that do not utilize any kind of model, the *train* method would do nothing at all, and all the functionality would be implemented in the *transform* method.

It is sometimes argued that methods which do not utilize an internal model do not actually *learn*. If you do not want to consider such algorithms to be learning algorithms, that is fine with me, but they may still be useful for many purposes, so let's not be in too big of a hurry to throw them out.

Some, but definitely not all, unsupervised operations are reversible, so it might be useful to add an operation for reversing the transformation:

```
Interface UnupervisedLearner
{
	abstract void train(Data& trainingdata);

	abstract void transform(Data& in, Data& out);

	abstract void untransform(Data& in, Data& out);
}
```

For unsupervised operations that cannot be reversed, perhaps the *untransform* method might be implemented to throw an exception.

Some, but definitely not all, unsupervised operations can operate on individual samples, instead of operating on entire datasets as a whole. For these techniques, a better interface might be something like this:

```
Interface IncrementalUnupervisedLearner
{
	abstract void train(Data& trainingdata);

	abstract void transform(Vector& in, Vector& out);

	abstract void untransform(Vector& in, Vector& out);
}
```

Incidentally, this architecture is well-suited for operations that perform type conversion, which we discuss in the next section.

2.1.9 Data type conversion

Some learning algorithms are only able to operate on continuous values. Others are only able to operate on categorical values. There are even some algorithms that are designed to operate on continuous inputs, but can only predict categorical labels. Thus, it is important to be able to convert arbitrary data to whatever format is needed, so we are not limited in our choices of learning algorithms.

Some algorithms implicitly handle data with missing values. Other algorithms only work if all values are known. Some algorithms expect values that fall within a certain range. All of these issues can be handled in general as well, so there is no necessity to make the learning algorithms more complex to handle all of these cases.

2.1.9.1 Converting categorical values to a continuous representation

A naïve approach for converting categorical values to a continuous representation is to simply enumerate the categories. Unfortunately, this inappropriately suggests that there is an implicit ordering to the categorical values.

A better method is to represent each categorical value with a categorical distribution, which assigns a probability to each category. In other words, a single categorical dimension with *k* possible values would be converted to a representation that occupies *k* continuous dimensions. For example, the data,

2.5	dog	0.01
1.0	mouse	0.02
1.2	cat	-0.01
3.7	cat	-0.03
4.1	dog	0.07

Would be converted to,

2.5	1	0	0	0.01
1.0	0	1	0	0.02
1.2	0	0	1	-0.01
3.7	0	0	1	-0.03
4.1	1	0	0	0.07

Sometimes this is called *binarization*, because it represents categorical values that have a known value with a binary vector. Since each category is represented in a separate dimension, which treats it as an orthogonal value, no implicit ordering is falsely implied.

We can also convert from a categorical distribution back to a categorical value. This is done by finding the mode of the distribution. (The mode is the value with the highest probability.)

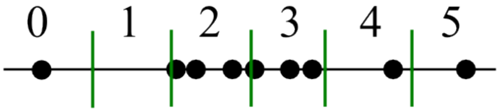
To illustrate why this approach is better than simply enumerating the categories, consider a vision application that attempts to classify the animal in a picture as one of {dog, cat, mouse}. Suppose our learning algorithm predicts with 0.5 confidence that it is a dog, 0.1 confidence that it is a mouse, and 0.4 confidence that it is a cat. If we find the mode of this distribution, then “dog” is the best prediction. By contrast, if we simply use an enumeration, where dog=0, mouse=1, and cat=2, then the mean prediction will come out to

$$0.5 * 0 + 0.1 * 1 + 0.4 * 2 = 0.9$$

This value is closest to 1, so “mouse” would be predicted. This is a poor value to predict, since it has the lowest confidence of them all. Thus, we see that falsely implying an unnatural ordering can create undesired results.

2.1.9.2 Discretization

Discretization is a method to convert continuous values to discrete values. It is done by grouping the values into buckets.



To convert from a bucket index back to a continuous value, the central value of the bucket is typically used. (This, of course, is a lossy operation.)

Sophisticated techniques exist for estimating the best number of buckets to use for a dataset, and the width of each bucket. A simple technique is to just use $\text{floor}(\sqrt{n})$ buckets, and to distribute them evenly between the min and max value in the attribute. This will create approximately the same number of buckets as the average number of points in each bucket. It may result in some buckets with zero points. For applications where this is problematic, more sophisticated techniques may be desirable.

2.1.9.3 Normalization

Suppose your continuous values lie in the range [a,b], but you want to use an algorithm that expects all values to fall in the range [c,d]. You can use a linear mapping to convert values to the desired range,

$$x' = (x - a)/(b - a) * (d - c) + c.$$

This equation is more intuitive if we break it down into four steps:

1. Subtract out the old minimum: $(x - a)$.
2. Divide out the old range: $/(b - a)$.
3. Scale in the new range: $*(d - c)$.
4. Add in the new minimum: $+c$.

This is called *normalizing* the data. You can also convert in the opposite direction with the same approach,

$$x = (x' - c)/(d - c) * (b - a) + a.$$

2.1.9.4 Baseline imputation

Imputation is the process of predicting missing values in data.

Data might be missing for any number of possible reasons: Maybe someone didn't fill in all the values in a survey. Maybe a new field was added to a survey after many samples were already collected. Maybe there was data corruption. Maybe a sensor malfunctioned. Maybe some values in the data were determined to be unreliable due to extreme values. The simple fact is that data is often missing in real-world situations, so we need to be able to deal with that. Imputation is one way of dealing with missing values, by predicting likely values for the missing elements.

To enable you to get started immediately with with data that may contain missing values, we describe a simple method called *baseline imputation*. More sophisticated imputation techniques exist, but we will study them later. For categorical attributes, we compute the most-common value (not counting missing elements), and replace all missing elements with that value. For continuous attributes, we compute the mean value (not counting missing elements), and replace all missing elements with that value. That's it.

It is rarely necessary to convert back in the other direction, but you could certainly throw out any values that were imputed if you really wanted to.

2.1.9.5 Filters

Each of these data transformations (binarization, discretization, normalization and imputation), is an unsupervised operation. It is common to perform one or more of them on data before passing it as the training data to a supervised learning algorithm. When this is done, the predictions made by that supervised learning algorithm often need to be converted back to the original form, so the predictions will be meaningful to the original caller.

A *filter* is a wrapper class that combines an unsupervised transformation with a supervised learning algorithm, as shown in this diagram. A filter presents data to the supervised learning algorithm only in the format that it supports, but enables the user to work with whatever data is available.

This helps to keep the supervised learning algorithms simple, by avoiding cluttering them with code for performing data type transformations.

Diagram of how a filter is used for training:

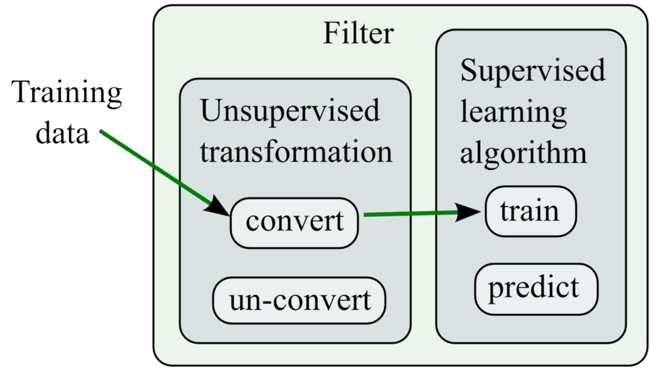
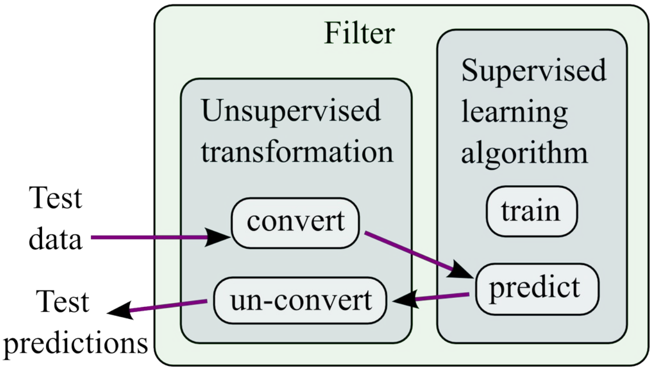


Diagram of how a filter is used for testing:



2.2 *Instance-based learning*

Instance-based learning is a very intuitive way to make predictions. The general idea of instance-based learning is that to make a prediction, find similar situations that occurred before, and predict that the same labels will be appropriate again.

2.2.1 Nearest neighbor algorithm

The most common implementation of instance-based learning is called the nearest neighbor algorithm. It is a simple algorithm (if you are not too concerned about efficiency), and it is very effective with some problems. It is trained by simply storing a copy of the training data, so it can be used later when predictions are needed. In other words, the training data itself defines the model. To prediction a label vector for some input pattern, the nearest neighbor algorithm searches through the training data to find the pattern that is most-similar to the input pattern. It predicts the same label vector for the input pattern as the nearest pattern in the training data.

The nearest neighbor algorithm relies on a distance (or similarity) metric to determine which pattern from the training set is nearest. If no other metric is specified, Euclidean distance is assumed. That is,

$$\text{Euclidean distance} = \sqrt{\sum_i (a_i - b_i)^2}\,.$$

In the case of categorical features, the term $a_i - b_i$ is usually replaced with Hamming distance,

$$\begin{cases} 0, & \text{if } a_i = b_i \\ 1, & \text{if } a_i \neq b_i \end{cases}.$$

(Note that since $0^2 = 0$ and $1^2 = 1$, it does not matter whether Hamming distance is substituted for $a_i - b_i$ or whether it is substituted for $(a_i - b_i)^2$. The result is the same either way.)

2.2.1.1 Numerical example

As a simple numerical example, suppose the following training data is given:

Gender	***Number of siblings***	***Age***	***Favorite color***
M	4	35	blue
F	0	12	pink
M	1	17	black

(Note that this is certainly not enough training data to accurately represent any reasonable population. Furthermore, some of the features are probably completely irrelevant to the label, which is not a very good property in training data. Nevertheless, even though we cannot expect very good results, we can still apply the nearest algorithm to this data.)

Suppose we wish to predict a label for the pattern:

F, 1, 75.

We begin by computing the distance between this pattern and each of the three patterns in the training data.

$$\begin{aligned}
d_1 &= \sqrt{1^2 + (4-1)^2 + (35-75)^2} \\
&= \sqrt{1610} \approx 40.125 \\
d_2 &= \sqrt{0^2 + (0-1)^2 + (12-75)^2} \\
&= \sqrt{3970} \approx 63.008 \\
d_3 &= \sqrt{1^2 + (1-1)^2 + (17-75)^2} \\
&= \sqrt{3365} \approx 58.009
\end{aligned}$$

It turns out that pattern number 1 is the “nearest neighbor”, so we predict that the favorite color of this new person is “blue”.

(Note that because square root is *monotonic* with positive values, it is not really necessary to calculate the square root in order to identify the nearest neighbor. Often, this operation is omitted in efficient implementations.)

2.2.1.2 Mahalanobis distance

If we examine the result from the previous numerical example, we see that the three features (Gender, Siblings, and Age) were not really given equal influence on the outcome. If the gender were different, that would only affect the distance measurement by 1. If the number of siblings were different, that would only affect the distance measurement by a small amount. But there was so much more deviation in the values of the age attribute that it dominated the distance metric.

To give each attribute equal influence on the prediction, we can divide the component of error due to each attribute by its deviation. This causes the distance metric to behave as if all the attributes had equal deviation. This is called Mahalanobis distance:

$$\begin{array}{c}\text{Mahalanobis}\\ \text{distance}\end{array} = \sqrt{\sum_i \left(\frac{a_i - b_i}{\sigma_i}\right)^2}$$

$$= \sqrt{\sum_i \frac{(a_i - b_i)^2}{\sigma_i^2}}\,.$$

“σ” is the lowercase Greek letter *sigma*. It is commonly used to represent deviation. (Likewise, “σ^2” is commonly used to refer to variance, which is the square of deviation.) In this equation, we use σ_i to represent the expected deviation in attribute i. For continuous attributes, σ_i can be estimated by first computing the mean value in attribute i in the training set,

$$\mu_i = \frac{1}{n}\sum_j a_{i,j}\,,$$

and then computing the average deviation from the mean,

$$\sigma_i = \sqrt{\frac{\sum_j (a_{i,j} - \mu_i)^2}{n-1}}\,.$$

(“μ” is the lowercase Greek letter *mu*, and it is commonly used to represent a mean.)

One might ask, *why do we divide by* $n-1$ *instead of dividing by* n *when we calculate the average deviation*? That is a good question—you must be paying attention. The answer is somewhat subtle, but is important to understand in machine learning. If we divide by n, then the result is the deviation within the training data. If we divide by $n-1$, then the result is the maximum-likelihood estimator of the deviation within the system that the training data represents. Remember, the training data are just samples of something. The more samples you take, the better those samples represent the system you take them from. The larger the value of n, the less difference it makes whether you divide by n or $n-1$. At the other extreme, suppose you have only one sample point in your training data. The deviation within this training data will be exactly zero. But what does that tell you about the deviation within the system? Absolutely nothing. It takes at least two points to get any kind of measurement of deviation. So, that's why we divide by $n-1$. If you want more details, you can read about *unbiased estimators of deviation*.

Intuitively, deviation represents the amount of error that can be expected to occur on average. For categorical attributes that are evaluated with Hamming distance, the expected error can be calculated as

$$\sigma_i = \sum_v p_v(1 - p_v),$$

where v iterates over each possible categorical value that attribute i supports, and p_v represents the portion of them that have the value v. In other words, if there are n rows in the training data, and m of them have the value, v, in attribute i, then $p_v = m/n$.

I think this equation is best explained by example. Suppose your training data contained a categorical attribute with possible values from the set {cat, dog, mouse}. Suppose there are 70 sample points in this data. 20 of them have the value “cat” in this attribute, 40 have the value “dog”, and 10 have the value “mouse”. Now, you are asked to predict a label for an out-of-band sample. We must assume that this out-of-band sample comes from approximately the same distribution as the training data. (This fundamental assumption is made by nearly all supervised learning algorithms. Supervision only has value to the extent that this assumption is valid.) So, about 2/7 of the time, the input pattern will have the value “cat” in this attribute. When this happens, 20 out of 70 comparisons will match, resulting in the Hamming error of 0. In 50 out of 70 comparisons, the Hamming error will be 1. We must also consider the possibility of “dog”, which occurs about 4/7 of the time, and “mouse”, which occurs about 1/7 of the time. So, overall, the expected Hamming error from this attribute will be

$$\frac{20}{70} \cdot \frac{50}{70} + \frac{40}{70} \cdot \frac{30}{70} + \frac{10}{70} \cdot \frac{60}{70} = 0.571,$$

which matches the formula.

So let's repeat the numerical example from the previous section using our normalized error metric. We will use the same training data. So, we begin by computing the mean value of each continuous attribute:

$$\begin{aligned}\mu_2 &= \frac{4+0+1}{3} = \frac{5}{3},\\ \mu_3 &= \frac{35+12+17}{3} = \frac{64}{3}.\end{aligned}$$

Next, we compute the expected error of each attribute:

$$\begin{aligned}\sigma_1 &= \frac{2}{3}\cdot\frac{1}{3}+\frac{1}{3}\cdot\frac{2}{3} = \frac{4}{9} \approx 0.444,\\ \sigma_2 &= \sqrt{\frac{(4-\frac{5}{3})^2+(0-\frac{5}{3})^2+(1-\frac{5}{3})^2}{2}}\\ &= \sqrt{\frac{13}{3}} \approx 2.082,\\ \sigma_3 &= \sqrt{\frac{(35-\frac{64}{3})^2+(12-\frac{64}{3})^2+(17-\frac{64}{3})^2}{2}}\\ &= \sqrt{\frac{439}{3}} \approx 12.097.\end{aligned}$$

Now that we know the expected error for each attribute, we are ready to predict a label for the input pattern,

F,   1,   75.

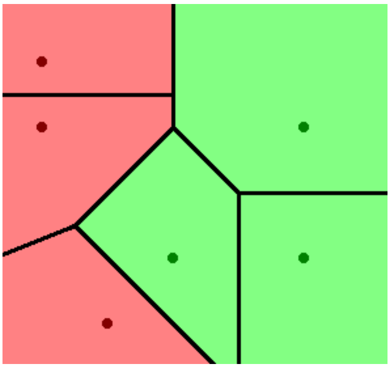
We do this by computing its normalized distance with each input pattern in the training data.

$$\begin{aligned}d_1 &= \sqrt{\left(\frac{1}{0.444}\right)^2+\left(\frac{4-1}{2.082}\right)^2+\left(\frac{35-75}{12.097}\right)^2}\\ &= 4.252,\\ d_2 &= \sqrt{\left(\frac{0}{0.444}\right)^2+\left(\frac{0-1}{2.082}\right)^2+\left(\frac{12-75}{12.097}\right)^2}\\ &= 5.230,\\ d_3 &= \sqrt{\left(\frac{1}{0.444}\right)^2+\left(\frac{1-1}{2.082}\right)^2+\left(\frac{17-75}{12.097}\right)^2}\\ &= 5.297.\end{aligned}$$

It turns out that the first training pattern is the nearest neighbor, so we predict the label *blue*. In this case, normalizing did not change the predicted label for this particular pattern, but it was somewhat closer. [todo: darn, make an example where it does have an impact.] Of course, there are many other possible input patterns, and many of them will have crossed over to receive a different prediction.

2.2.1.3 Voronoi diagrams

The model space of the nearest neighbor algorithm can be visualized with a Voronoi diagram. For example, here is a diagram of a model involving 6 training points. Each point consists of two continuous input attributes (visualized by its position on the horizontal and vertical axes) and a binary label (visualized with a red or green dot).



The region enclosing each training point indicates places where that point will be the nearest neighbor. The boundary between any two adjacent regions always occurs where the two corresponding points are equidistant.

2.2.2 *k*-nearest neighbors

One problem with the nearest neighbor algorithm is that it is very sensitive to noise in the training data. For example, just one mis-labeled point in the training data will create a region in the model space where patterns will be mis-classified.

A common approach to mitigate this problem is to find the *k*-nearest neighbors, instead of just the one nearest neighbor. The *k* points are combined in the same manner as the baseline algorithm to produce a single predicted label vector. That is, for continuous label attributes, the mean is predicted, and for categorical label attributes, the most frequently-occurring value is predicted.

The *k*-nearest neighbors algorithm, often abbreviated *k*-nn, is a generalization that encompasses both the nearest neighbor algorithm and the baseline algorithm. If *k* = 1, then it is the nearest neighbor algorithm. If *k* is equal to the number of points in the training set, *k*-nn is the baseline algorithm. When *k* is some value in between, *k*-nn is more intelligent than the baseline algorithm, and more robust to noise than the nearest neighbor algorithm.

2.2.2.1 Weighting the neighbors

k-nn can be made even more effective if the nearest neighbors are weighted in an intelligent manner, instead of giving each neighbor an equal vote. It is very common to give each neighbor a weight of $1/d$ or $1/d^2$, where d is the distance from the input pattern to the neighbor in the training set.

For example, suppose *k*=3. Suppose the three neighbors are respectively labeled “true”, “false”, and “true”, and suppose the distances to these neighbors are respectively 3.1, 1.3, and 2.8. If we give each neighbor an equal vote, then the predicted label will be “true”, because two neighbors are labeled true. If we weight each neighbor by $1/d$, then we tally their votes according to their weight:

$$\frac{1}{3.1} + \frac{1}{2.8} \approx 0.680 \text{ votes for true, and}$$

$$\frac{1}{1.3} \approx 0.769 \text{ votes for false.}$$

So, the predicted label would be “false”, because it received the largest amount of weighted vote.

If the labels are continuous we would calculate a weighted mean for the prediction. For example, if the labels were 2.1, 3.0, and 1.9, then the weighted mean would be:

$$\frac{\frac{1}{3.1} \cdot 2.1 + \frac{1}{1.3} \cdot 3.0 + \frac{1}{2.8} \cdot 1.9}{\frac{1}{3.1} + \frac{1}{1.3} + \frac{1}{2.8}} \approx 2.53.$$

(With equal weighting, the prediction would be

$$\frac{2.1 + 3.0 + 1.9}{3} \approx 2.33.$$

Notice that weighting each neighbor by $1/d$ tends to favor the neighbors that are closer. If the weighting is $1/d^2$, then the closer neighbors are favored even more.)

The following graph shows how predictions are interpolated between two neighbors for different weightings. The horizontal axis represents a continuous input value. The vertical axis is a continuous output label. The two black dots represent two neighbor points from the training set. The green line shows the prediction if each neighbor is given equal weight. The red line shows the prediction if each neighbor is given a weight of $1/d$. (This is called *linear interpolation*.) The yellow-brown curve shows the prediction if each neighbor is given a weight of $1/d^2$.

To be more general, you can give each neighbor a weight of $1/d^q$, where d is the distance to that neighbor, and q is

0 for equal weighting (green),
1 for linear interpolation (red), and
2 for that curvey one (yellow-brown). (todo: find a more technical name)

If you wanted something in-between, you could even use a non-integer value for q.

Another common weighting is to use a Gaussian kernel. This weighting is used in cases where a smooth or differentiable model-space is required. (For example, a model used with the extended Kalman filter needs to be differentiable.) With the Gaussian kernel, the weight of each neighbor is,

$$e^{-d^2/2\sigma^2}.$$

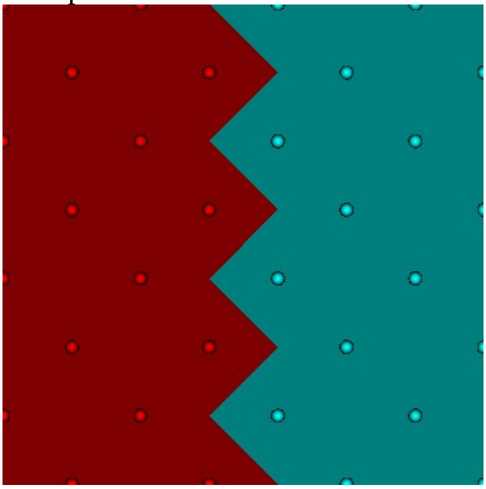
This will give a lot of weight to points that lie within one or two deviations, and very little weight to points that are several deviations away. Consequently, it becomes reasonable to combine the labels from every point in the training data, instead of just the *k*-nearest points. So, the Gaussian kernel saves us from having to pick a value for *k*, but it introduces a new parameter, σ.

Perhaps, one approach for picking σ might be to visit each point in the training data and calculate the average distance between each point and its k^{th} neighbor. Then, σ depends on k. So, the only thing really gained is that the model-space is now smooth.

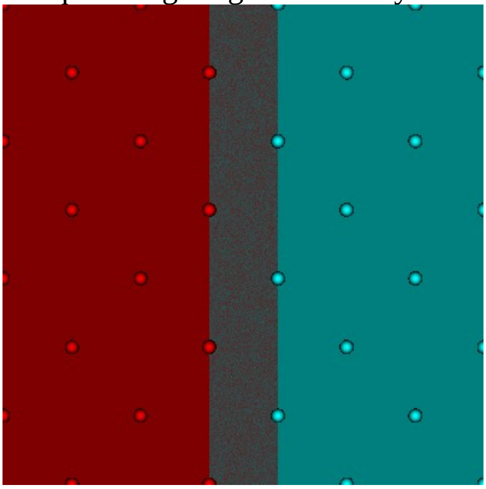
2.2.2.2 Some visual model-space comparisons

Let's take a look at some model-space visualizations. These involve two continuous input attributes (visualized on the horizontal and vertical axes), and a binary class label (visualized in red and cyan.) The training data is distributed in a regular pattern. (This is not likely to occur in real-world data, but it is useful for visualizing the behavior of *k*-nn.)

The model-space of 1-nn:

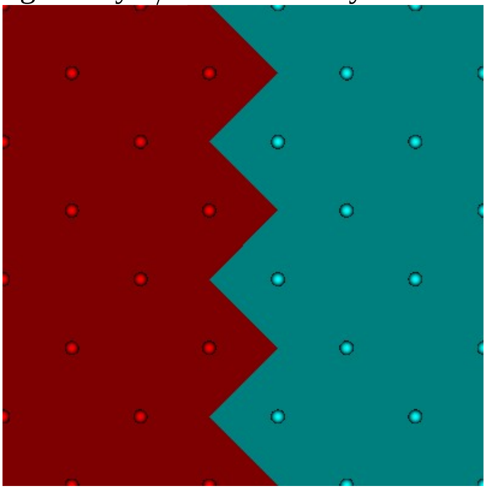


2-nn with equal weighting and a binary label:



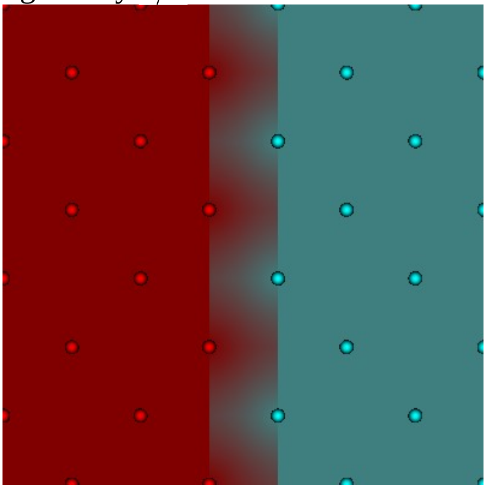
The vertical stripe in the middle shows a region of ambiguity, where both classes receive equal vote. For this reason, it is common to use odd values for *k* whenever equal weighting is used.

2-nn weighted by $1/d$ with a binary label:



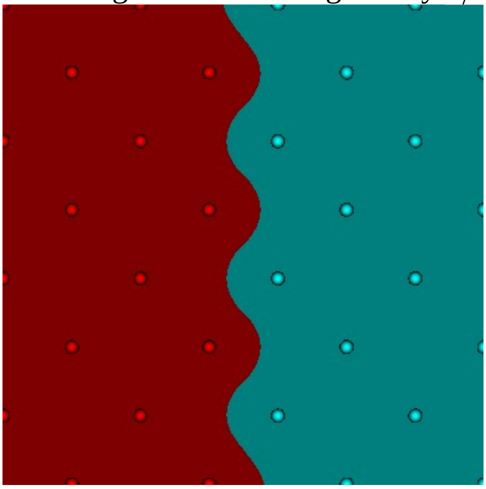
(In this case, results are identical with 1-nn, but this is not always the case with other datasets.)

2-nn weighted by $1/d$ with a continuous label:



Note that there are continuous gradients in this model-space, but there are also other places where hard boundaries occur. These hard boundaries occur where there are changes in the set of nearest-neighbors. (If this were a problem, we could solve it by setting k equal to the number of points in the training set and using a Gaussian kernel weighting. For many applications, however, hard boundaries are not really a problem.)

7-nn with a categorical label weighted by $1/d$:



2.2.3 Alternate distance metrics

Other distance metrics may be substituted for Euclidean distance to alter the behavior of *k*-nn. The distance metric used can have a significant effect on the behavior of *k*-nn. In fact, if *k* is equal to the number of training points, then the right distance metric could make *k*-nn behave exactly like any other learning algorithm.

2.2.3.1 L^p-norm

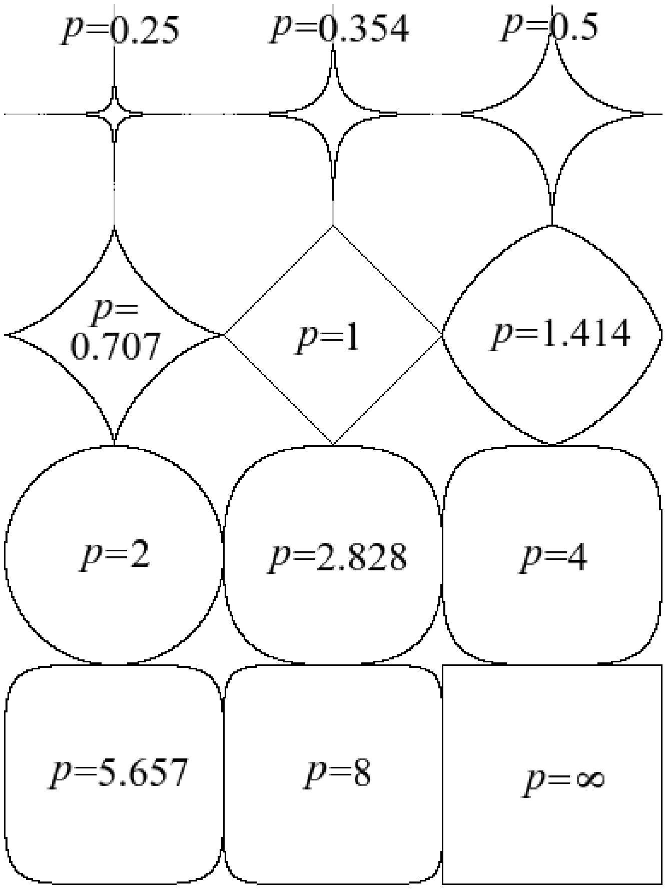
One interesting metric is L^p-norm distance. It is defined as,

$$L^p\text{-norm distance} = \left(\sum_i (a_i - b_i)^p\right)^{\frac{1}{p}}.$$

In this equation, the value of *p* determines how the distance is calculated. L^p-norm distance is a generalization of several other well-known distance metrics:

p	***Distance metric***
0	Hamming distance (if each category is represented with a unique binary attribute)
1	Manhattan distance (a.k.a. rectilinear distance, a.k.a. taxicab distance, a.k.a. city block distance)
2	Euclidean distance
∞	Chebyshev distance (a.k.a. chessboard distance)

The following chart shows the perimeter at the unit distance (meaning a distance of 1) from the origin with several values of *p*. (The origin is located in the center of each depicted shape.)



Note that Euclidean distance (*p*=2) is the only L^p-norm that is invariant to rotation. For this reason, Euclidean distance is generally the best-suited L^p-norm for data where the values may be expressed in combinations of attributes. With many datasets, however, each attribute represents an independent concept with an unknown influence on the target label. In such cases, when the dimensionality of the inputs is large, it has been found that smaller values for *p* tend to yield higher predictive accuracy. [todo: cite]

The concept of Mahalanobis distance is easily combined with L^p-norm distance:

$$\begin{array}{c}\text{Mahalanobis}\\ L^p\text{-norm distance}\end{array} = \left(\sum_i \left(\frac{a_i - b_i}{\sigma_i}\right)^p\right)^{\frac{1}{p}}.$$

2.2.3.2 Sparse metrics

Suppose you wish to use *k*-nn to build a recommendation system for movies. Users of your system will rate movies that they have seen. After a user has rated a few movies, the system can recommend movies that the user is predicted to enjoy. This is done by finding the *k* users whose preferences are closest to the preferences expressed by the current user. We can then combine all the ratings from these *k* users to predict how much the current user will enjoy other movies that he or she has not yet rated. (In this case, the training data is all of the ratings by all of the users except for the current user.)

One problem that we will face when working with this data is that most of the values in the training data are missing. It is extremely unlikely that every user has seen every movie. So, how do we work effectively with data that has missing values?

One significant weakness of L^p-norm distance is that it does not handle missing values very well at all. If you simply omit the terms that involve missing values when calculating the distance, then the training patterns with the most missing values will be very close to everything, and will thus have the most influence. In fact, these are probably the training patterns that you would prefer to have the least influence, since so little is known about them. Other possible solutions are to assume the missing value has the same error as the average error of the other values in the vector, or to use a constant error term whenever a missing value occurs. These approaches may be a little better, but they are still very sensitive to the conditions in the data. A better solution when many values are missing is to use a distance metric that is designed to handle missing values well.

One such metric is *Pearson's correlation coefficient* (also called the *sample correlation coefficient)*. It is defined as,

$$\rho = \frac{\sum_i (a_i - \mu_a)(b_i - \mu_b)}{\sqrt{\sum_i (a_i - \mu_a)^2}\sqrt{\sum_i (b_i - \mu_b)^2}},$$

where $\mathbf{a}$ and $\mathbf{b}$ are the two vectors being compared, a_i refers to element i in $\mathbf{a}$, b_i refers to element i in $\mathbf{b}$, ρ is the lowercase Greek letter “rho”, but it looks like a “p”, so it is sometimes used to represent Pearson's correlation coefficient, μ_a is the mean of the element values in $\mathbf{a}$, and μ_b

is the mean of the element values in $\mathbf{b}$. In this equation, if either a_i or b_i is missing, then its value is assumed to be μ_a or μ_b, which effectively drops the term from the equation. (This also makes sparse implementations very efficient, because only the non-missing values need to be visited.) Since the impact of each value on the equation depends on its difference from the mean, missing values have little significant net impact on the overall metric.

Pearson's correlation coefficient returns a value in the range [-1,1]. It is better termed a *similarity metric*, rather than a distance metric, because it returns a larger value for similar vectors and a smaller value for dissimilar vectors. (Of course, it would trivial to multiply by -1 if you prefer to use it as a distance metric.)

A closely-related similarity metric is *cosine similarity*. It is defined as,

$$c = \frac{\sum_i (a_i)(b_i)}{\sqrt{\sum_i (a_i)^2}\sqrt{\sum_i (b_i)^2}}$$

This computes the cosine of the angle formed by the two vectors, $\mathbf{a}$ and $\mathbf{b}$, with the origin. Like Pearson's correlation coefficient, this metric handles sparse data very well. It also returns a value in the range [-1, 1], with bigger values indicating a stronger similarity.

With cosine similarity, vector-element values that are close to zero have very little impact on the overall metric, and values far from zero have a larger impact.

As an example, let us consider an information retrieval application in which we wish to find the *k* documents that most-closely match a particular query. We might represent each document by counting the number of times that each possible word occurs in that document. There are about 60,000 commonly-used words in the English language, so each document will be represented as a 60,000-dimensional vector. Of course, the majority of words will never occur in the majority of documents, so our dataset of vector-encoded documents will contain mostly zeros.

Note that this is not the same kind of sparsity as in the previous (movie recommender) example. It is important to distinguish between data where sparse means that most values are missing, and data where sparse means that most values are the same. Techniques that work well with one type of sparse data will not necessarily work well with the other. In this case, sparse means that most elements have the same value. This data is likely a good fit for cosine similarity because cosine similarity treats zero as its point-of-origin.

For datasets where zero has no implicit meaning as a value, it is desirable to choose an origin that is not arbitrary. One way to do this is to subtract the mean from all of the values, thus moving the centroid to the origin. Thus, the equation for *adjusted cosine similarity* becomes,

$$c = \frac{\sum_i (a_i - \mu_i)(b_i - \mu_i)}{\sqrt{\sum_i (a_i - \mu_i)^2}\sqrt{\sum_i (b_i - \mu_i)^2}}.$$

This equation looks extremely similar to the equation for Pearson's correlation coefficient. There is, however, a subtle difference: The subscript on μ is not the same. In Pearson's correlation coefficient, we subtracted the mean of the elements in the vector. In adjusted cosine similarity, we subtract the mean of the values in attribute i in the training data. The only difference is in the mean that they use as a point-of-reference.

So, which point-of-reference makes more sense for your data? Well, of course, that depends on the data. We might settle the matter intuitively, by asking which mean is more meaningful in your data, but it is probably better to settle the matter empirically. So, the best answer is, use cross-validation to try them all, and pick the one that gives the best score. And while you're at it, you might see if you can come up with a few other clever variations. If one of those does better in cross-validation, then it is probably better-suited for your data.

Often in the real world, empirical results disagree with our intuition. When that happens, we must admit that there are yet factors that we do not understand. Some “data scientists” have a tendency to become so converted to a particular technique because of its intuitive elegance that they reject other approaches even when they perform better empirically. Such dogmatic behavior is inconsistent with the scientific method.

2.2.3.3 Using a learning algorithm as the distance metric

Yet another possibility is to use an arbitrary learning algorithm to combine the labels from the *k*-nearest neighbors into a single predicted label. This would have to be a very efficient learning algorithm, however, because it would need to be trained and evaluated each time the instance-based learner was asked to make a prediction. For efficiency reasons, therefore, this approach is not often used.

2.2.4 Performance enhancements

k-nn is one of the fastest learning algorithms to train (because all it needs to do is store the training data). Unfortunately, its hypotheses are very bulky (because they consist of the entire training set), and it is one of the slower algorithms for making predictions (because it must find the *k*-nearest neighbors to make a prediction). In order to turn *k*-nn into a serious learning algorithm, it is necessary to do something about its size and speed.

2.2.4.1 Pruning

One way to make *k*-nn faster is to reduce the size of the training data that it stores as its hypothesis. If you throw out the right data points, you can improve both its size and speed without significantly reducing its accuracy. Thus, pruning effectively shifts the computational cost from prediction-time to training-time, which is generally a good thing.

Many sophisticated techniques for pruning the hypothesis of *k*-nn have been studied. In this section, we review only some of the simplest techniques.

Ideally, we would like to remove points in the order that reduces accuracy the least. Of course, you cannot get a good estimate of accuracy if you test on the training set—especially with *k*-nn, because it stores the training set. Therefore, it is necessary to hold out some of the training data. With a hold out set, we could systematically try removing each point, and then actually remove the one that leaves the hypothesis with the highest accuracy. (It is even possible that removing some points will improve accuracy—those should definitely be the first to go.)

The problem with this technique is that it is has ludicrous computational cost. If the training set has *n* points, and the hold out set has *m* points, then we would need to measure accuracy with the hold out set *n* times, just to remove a single point from the training set. To fully prune a dataset in this systematic manner could have a computational cost as bad as $O(mn^3)$. That may be a polynomial, but it is much to slow for practical purposes.

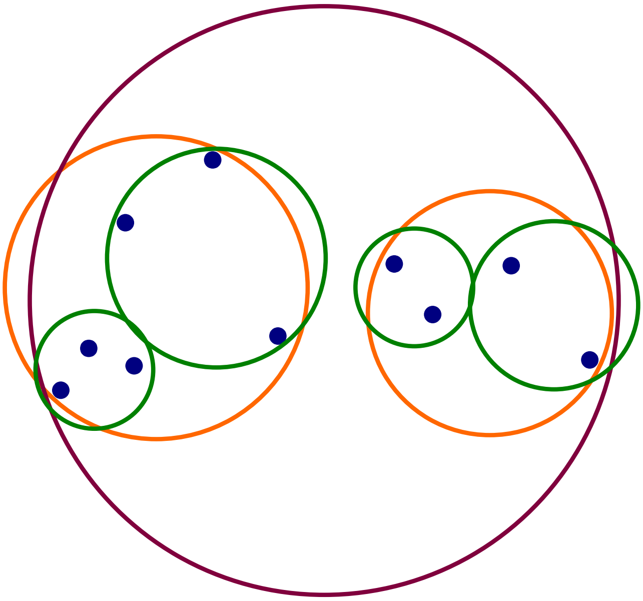
Faster approaches typically involve guessing which points will have the least negative impact on accuracy. For example, points whose *k*-nearest neighbors all have the same class label are quite likely to have little impact when they are removed. There are cases where such points could be important, but such points are more likely to be unimportant. If the label is continuous, rather than categorical, we could remove points whose *k*-nearest neighbors have an average label deviation that is only a small percentage of the overall label deviation in the training set. After these points are removed, then the systematic approach will be faster, because it will then have fewer points to operate on.

We can also make the systematic approach less expensive by making it somewhat less systematic. For example, as we evaluating the impact of removing each point, we could immediately remove any point if its removal resulted in higher accuracy than that of any of the previous *u* points that were tried, where *u* is some constant number. This is called a *satisficing* approach. Instead of seeking an optimal solution, *satisficing* seeks solutions that are “good enough”.

2.2.4.2 Ball trees

(Before reading this section, it might be a good idea to read the appendix that reviews essential linear algebra concepts.)

A ball tree is a data structure that can be used to accelerate the process of finding neighbors. It is a hierarchy of balls that encompass portions of the data. The root ball encompasses all of the data. It references two child balls, which each encompass half of the data. They each have two child balls, and so forth, until the leaf balls encompass some constant number of points points.



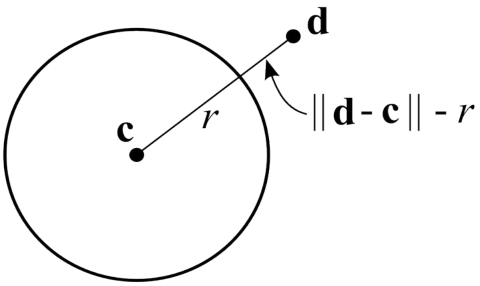
Here is a recursive algorithm to construct a ball tree for a dataset, $\mathbf{D}$:

```
function buildBallTree($\mathbf{D}$)
   Let $\mathbf{D}$ be a set of point indexes (not the actual
      data points themselves, just row indexes).
   Compute the center, $\mathbf{c}$, and radius, $r$, of a ball
   that bounds all the points referenced in $\mathbf{D}$.
   if $\mathbf{D}$ contains at least 12 point indexes:
         Let $\mathbf{a}$ be an arbitrary point drawn from $\mathbf{D}$.
         Let $\mathbf{e}$ be the furthest point in $\mathbf{D}$ from $\mathbf{a}$.
         Let $\mathbf{f}$ be the furthest point in $\mathbf{D}$ from $\mathbf{e}$.
         Let $\mathbf{G}$ reference the set of points in $\mathbf{D}$ that
         are closer to $\mathbf{e}$ than they are to $\mathbf{f}$.
         Let $\mathbf{H}$ reference the set of points in $\mathbf{D}$ that
         are closer to $\mathbf{f}$ than they are to $\mathbf{e}$.
         leftChild = buildBallTree($\mathbf{G}$)
         rightChild = buildBallTree($\mathbf{H}$)
         return a new interior ball containing
         $\mathbf{c}$, $r$, leftChild, and rightChild.
   else
         return a new leaf ball containing
         $\mathbf{c}$, $r$, and a copy of the point indexes in $\mathbf{D}$.
```

In order to implement this algorithm, you will also need to know how to compute a center and radius for a bounding ball. An easy solution would be to just use the centroid for the center of the ball, and find the point farthest away from this centroid to determine the radius. Unfortunately, this simple approach often creates very loose-fitting bounding balls. So, here is a better algorithm that will generally find a snug-fitting bounding ball around the points in $\mathbf{D}$:

```
function compute_bounding_ball($\mathbf{D}$)
   Let $\mathbf{a}$ be an arbitrary point referenced in $\mathbf{D}$.
   Let $\mathbf{e}$ be the furthest point in $\mathbf{D}$ from $\mathbf{a}$.
   Let $\mathbf{f}$ be the furthest point in $\mathbf{D}$ from $\mathbf{e}$.
   $r = ||\mathbf{f} - \mathbf{e}||/2$
   $\mathbf{c} = (\mathbf{e} + \mathbf{f})/2$
   while any point $\mathbf{d} \in \mathbf{D}$ is outside the ball
         $\mathbf{c} = \mathbf{c} + \left(1 - \frac{r}{||\mathbf{d} - \mathbf{c}||}\right)(\mathbf{d} - \mathbf{c})$
         $r = 1.02r$
```

This algorithm works by moving the center of the bounding ball just enough to enclose each point that is found outside the ball, and growing the ball by a small amount each time.



The reason we use balls (or spheres) is because it is very easy to compute the distance between a point and the sphere. We start by computing the distance between the point and the center of the sphere. If that distance is less than the radius of the sphere, then the point is within the sphere, so the distance between the point and sphere is 0. If it is greater, then we just subtract the radius to obtain the distance between the point and the nearest part of the sphere. So, the distance between point $\mathbf{d}$ and sphere $\{\mathbf{c}, r\}$ is,

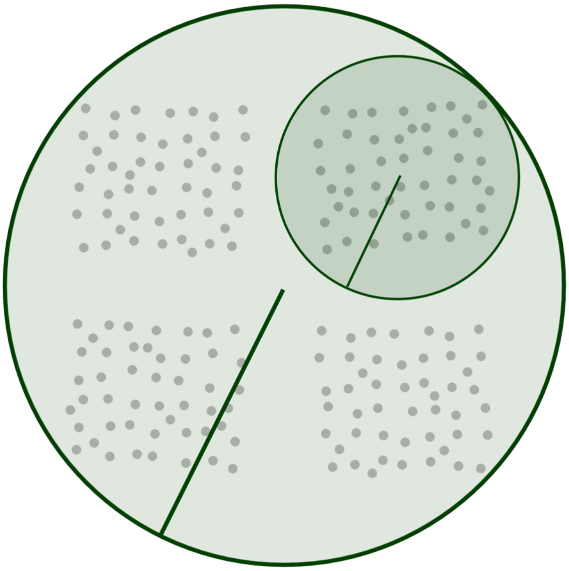
$$\max(||\mathbf{d} - \mathbf{c}|| - r, 0).$$

After we construct a ball tree, we can use it to efficiently find the nearest neighbors of any point. This is done using a priority queue that orders the balls according to their proximity to the point for which we are seeking neighbors. When the nearest ball is farther away than the *k*-nearest points we have considered so far, we know we can stop, because all other points will be farther away.

Here is a pseudo-code description of how to use a ball tree to find the *k*-nearest neighbors of a point, $\mathbf{d}$:

```
function findNeighbors($\mathbf{d}$)
    Let N be a priority queue of points, sorted by
    distance from $\mathbf{d}$, furthest first.
    Let Q be a priority queue of balls, sorted by
    distance from $\mathbf{d}$, closest first.
    Add the root ball of the ball tree to Q.
    while Q contains at least one ball:
        Remove a ball, S, from Q.
        if there are at least $k$ points in N:
            if S is further from $\mathbf{d}$ than the first
            point in N is from $\mathbf{d}$:
                break out of the while loop
        if S is a leaf ball:
            Add each point in S to N.
            While N contains more than $k$ points.
                Discard the first point in N.
        else
            Add each child ball in S to Q.
    Return N.
```

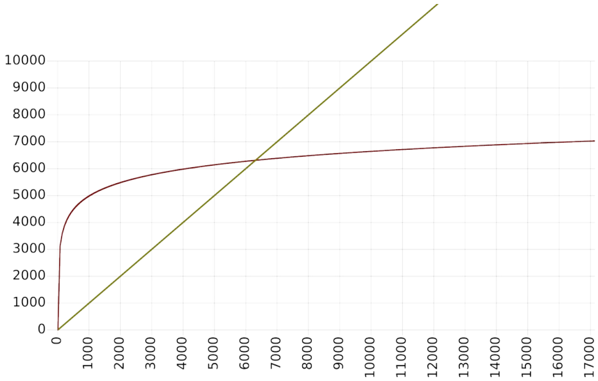
Todo: talk about the triangle inequality, and whether ball trees are suitable for non-Euclidean distance metrics.



It can be observed that in 2-dimensional space, it can takes 2 divisions to reduce the radius of a bounding ball by a factor of about 2. More generally, in *m*-dimensional space, it may take up to *m* divisions of the data to reduce the radius of a bounding ball by a factor of 2. Consequently, the computation required to find the *k*-nearest neighbors of a single point using a ball tree that was constructed over *n* points in *m*-dimensions will be approximately proportional to

$$\log(n)/\log(2^{\frac{1}{m}}).$$

For example, if the data points occupy 500 dimensions, then it may take more than 6000 data points before a ball tree offers any improvement over a brute-force approach to finding neighbors (and this is ignoring the time required to build the ball tree, the consequences of possibly sub-optimal divisions, and any extra memory and computational overhead associated with the tree).



The red line in this chart shows the theoretical scalability of a ball tree in 500-dimensional space. The yellow line shows the scalability of the brute-force approach to finding neighbors.

When there are many data points, the improvement offered by a ball tree can be quite dramatic. If the points happen to fall on a low-dimensional manifold within the high-dimensional space, then it may take even fewer points before improvements are obtained.

2.2.5 Instance-based imputation

Instance-based learning can also be used in an unsupervised manner. One useful example is to predict the missing values in a dataset.

This is done by finding the *k*-patterns that are most similar to the one with missing values that we would like to fill-in. We then combine the values from those patterns to predict a value for the missing element.

Of course, the *k*-nearest patterns might also be missing some values, so we only combine values that are actually known. To make sure that predicted values do not influence subsequent predictions, it may be necessary to maintain two copies of the data, or else keep track of which values were imputed. It is also necessary to use a distance metric that can handle missing values. Thus, Pearson's correlation coefficient or cosine similarity are common choices.

2.2.6 Instance-based collaborative filtering

Suppose you wanted to build a recommendation system, where users are asked to rate movies, books, recipes, music, or other items with which they are familiar. Then, the system will recommend new items that the user is likely to rate highly. How can we predict the ratings that a user is likely to give for an item that user has never even seen before?

Well, we find the *k*-users whose rating vector is most similar with this user, and we combine the ratings of those users to make predictions for this user.

If you look closely, you will notice that collaborative filtering is really the same thing as imputation. It simply involves making predictions to fill in missing values in a big table of ratings.

2.3 Decision tree learning

A decision tree is a model that uses a tree structure to decide what label to predict for a pattern. This tree is formed by recursively dividing the training data until all of the points remaining in each leaf node have homogeneous label vectors.

For example, suppose we are given the following training data:

Pattern				Label
Socks	***Shirt***	***Pants***	***Tie***	***Fashion***
match	T-shirt	shorts	no	stylish
match	none	none	yes	dork
mismatch	button-up	shorts	no	dork
match	button-up	slacks	yes	stylish
mismatch	T-shirt	slacks	no	dork
match	none	shorts	yes	dork
mismatch	T-shirt	shorts	no	dork

We start by checking whether the labels are homogeneous (all the same). They are not, so next we pick an input attribute to divide on. Let's say that we pick "Socks". There are two possible values in this attribute, so we divide the data into two parts,

Pattern				Label
Socks	***Shirt***	***Pants***	***Tie***	***Fashion***
match	T-shirt	shorts	no	stylish
match	none	none	yes	dork
match	button-up	slacks	yes	stylish
match	none	shorts	yes	dork

Pattern				Label
Socks	***Shirt***	***Pants***	***Tie***	***Fashion***
mismatch	button-up	shorts	no	dork
mismatch	T-shirt	slacks	no	dork
mismatch	T-shirt	shorts	no	dork

Then, we recursively call the same function with each part of the data.

The mismatching portion has homogeneous labels, so it will create a leaf node with the label “dork”. The matching portion does not have homogeneous labels, so it will divide again. Perhaps, this time it will pick the “Tie” attribute, so the data will be divided into:

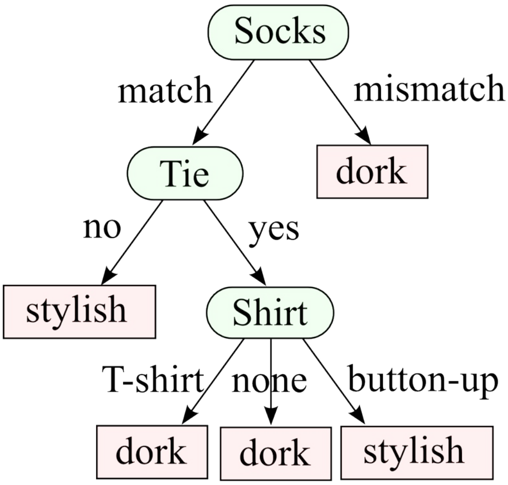
Pattern				Label
Socks	***Shirt***	***Pants***	***Tie***	***Fashion***
match	T-shirt	shorts	no	stylish

and

Pattern				Label
Socks	***Shirt***	***Pants***	***Tie***	***Fashion***
match	none	none	yes	dork
match	button-up	slacks	yes	stylish
match	none	shorts	yes	dork

Now, one part has only one data point, so it is definitely homogeneous. The other part is still not homogeneous, so it would be divided yet again. Perhaps, this time it will divide on the “Shirt” attribute.

The decision tree that results from this recursive dividing process could be visualized like this:



We can then use this decision tree to make predictions. For example, suppose we have the test pattern <match, button-up, shorts, yes>. To predict a label for this pattern, we start at the root of the tree and follow the appropriate arrows until we arrive at the predicted label, “stylish”.

Is this really a good prediction? A tie with shorts is stylish? Well, we didn't really have enough training data to develop a hypothesis with any real fashion sense, but we did manage to capture some of the trends in the data. (Does that even count as a pun? I don't think so. Fortunately, no one hates bad puns.) Perhaps if we had more data, we might be able to produce a tree that might actually be useful to fashion-impaired individuals.

2.3.1 Some implementation issues

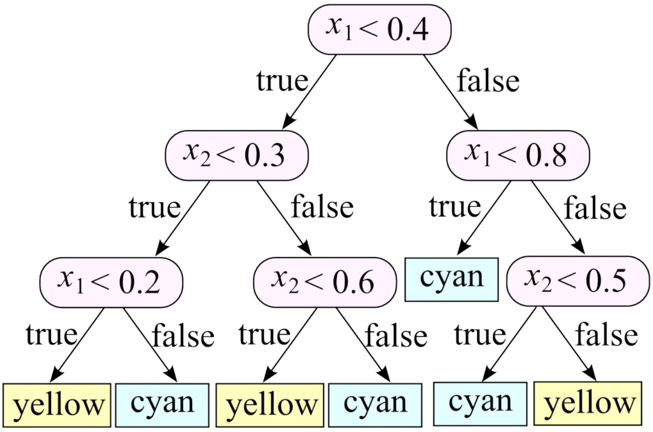
Some common issues arise when people write code to implement a decision tree. This section attempts to preempt these issues:

- The example diagram of a decision tree in the previous section shows a word or string in each node of the tree, and along each edge. In practice, however, it would be inefficient to implement a decision tree to do string comparisons. A better implementation typically involves enumerating all of the possible values before the tree is even built. This way, interior nodes need only store the index of the attribute on which they divide, and leaf nodes need only store the index of a label value.
- If a continuous input attribute is chosen for the division, it is necessary to also choose a pivot value for comparison. Data points with a value less than the pivot in this attribute will go one way, and data points with a value greater than or equal to the pivot value in this attribute will go the other way.
- When implementing the code that builds a decision tree, it is important that at least one data point in the training set is separated from the others by each division. Otherwise, you might keep “dividing” on the same attribute and never terminate. Furthermore, ineffective divisions add useless computational cost to the prediction step, which is a bad thing. If a computational cost must be paid, it is better to do it at training time. A good way to ensure that each division is effective is to maintain a white-list

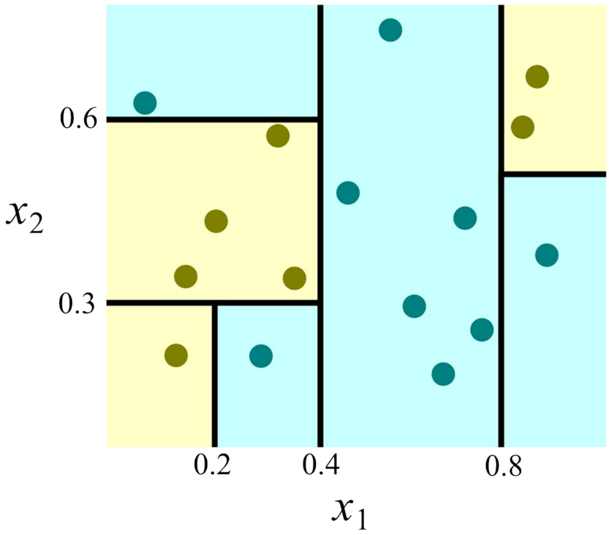
of candidate attributes. Before you build the tree, you initialize this list to contain all possible attributes. A copy of this list is passed to each recursive call that builds the tree. Whenever an attribute is found to have no dividing power with the remaining portion of the data, it is removed from the list, and another one is selected.

When continuous input attributes are picked for the division, they are not usually removed from the list of candidate attributes, because a different pivot value might be chosen next time.

For example, here is a decision tree that operates in a space of two continuous inputs, and one binary label:



And here is a visualization of the model space represented by this decision tree. (This diagram also shows 16 training points that could have been used to train this model.



Note that this tree can divide on both x_1 and x_2 multiple times, because it uses a different pivot value each time.

- When an attribute has at least 3 possible categorical values, it may sometimes occur that there are no data points to represent one of the values. (If you look closely, you will see that this condition occurs in the example problem in the previous section. When we divide on the “Shirt” attribute, there are only 3 data points in this branch, and none of them have the value “T-shirt”. So, how did we know to assign the label “dork” to this branch?) When this occurs, the label should be the baseline label of the parent node. (For categorical labels, this is the most common value. For continuous labels, this is the mean value.) In order to implement this, each interior node needs to calculate the baseline label (or baseline label vector, if there are multiple label dimensions).

  There is another possible solution to this problem, which may be somewhat simpler. It is to only perform binary divisions. So, each interior node would store both an attribute and a value. One branch would be for data points that have that attribute-value pair, and the other branch would take all the other values. This solution also simplifies the process of picking the best division, which we discuss in the next section. If this solution is used, then the white-list of attributes should actually be a white-list of attribute-value-pairs.

2.3.2 Choosing a division

The attributes (or attribute-value pairs) that you choose to divide on can have a big impact on the effectiveness of your decision tree.

2.3.2.1 Random trees

Perhaps the simplest way to choose a division is to pick on at random from the list of candidate attributes (or attribute-value pairs). This results in what is called a *random tree*. This does not mean that the tree itself is random. It is still guided by the data. It only means that the divisions within the tree are chosen randomly.

When a continuous attribute is randomly chosen, it is also necessary to choose a pivot value. One simple way to do this is to pick a random sample from the portion of the training data in the current branch, and use the value in that sample for the pivot.

Compared with other models, decision trees tend to be very *volatile*. This means that a small change in the training data can have a big impact on the predictions that it makes. Volatility can be either good or bad, depending on the application. In situations where volatility is bad, decision trees are typically not used at all. In situations where volatility is good, using random divisions makes decision trees even more volatile. We will visit this concept in further detail when we discuss ensembles.

2.3.2.2 Entropy-reducing trees

A more systematic approach to choosing divisions is to choose the one that does the best job of separating the labels. In other words, we want the division that makes the labels as homogeneous as possible.

Entropy is a measure of heterogeneity. It is defined as,

$$\text{entropy} = \sum_v -p_v \log_2(p_v).$$

In this equation, v iterates over the possible label values. p_v is the portion of the remaining data points with label v. (When p_v is zero, the term $-p_v \log_2(p_v)$ is evaluated as zero.) In our fashion dataset example, 2 out of 7 samples were labeled “stylish”, and 5 out of 7 were labeled “dork”.

So, the entropy of the labels in this dataset is

$$-\frac{2}{7}\log_2\left(\frac{2}{7}\right)-\frac{5}{7}\log_2\left(\frac{5}{7}\right)\approx 0.863\,.$$

In many programming languages, the “log” function computes $\log_e$. You can use this equation to help you calculate $\log_2$:

$$\log_2(x)=\frac{\log_e(x)}{\log_e(2)}\,.$$

After we divide the data with the attribute “Tie”, there are two portions. In the first portion, 2 out of 4 samples are labeled “stylish”, and 2 out of 4 are labeled “dork”. So, the entropy of this portion is

$$-\frac{2}{4}\log_2\left(\frac{2}{4}\right)-\frac{2}{4}\log_2\left(\frac{2}{4}\right)=1\,.$$

This is as heterogeneous as is possible with only two possible label values. In the second portion, none of the samples are labeled “stylish”, and all three of the samples are labeled “dork”. Therefore, the entropy of this portion is

$$-\frac{0}{3}\log_2\left(\frac{0}{3}\right)-\frac{3}{3}\log_2\left(\frac{3}{3}\right)=0\,.$$

To evaluate this division, we need to combine these two results into a single weighted entropy score. We weight each result by the portion of the data it represents to produce a combined score,

$$\frac{4}{7}\cdot 1+\frac{3}{7}\cdot 0\approx 0.571\,.$$

So, an entropy-reducing tree would evaluate each attribute (or each attribute-value pair) in this manner, and always choose the one that results in the lowest combined entropy score.

The difference between the entropy before the division and the entropy after the division is called *information gain*. In this case, the information gain is

0.863 – 0.571 = 0.292.

So, an entropy-reducing tree always chooses the division that maximizes information gain.

For continuous labels, variance could be used in place of entropy.

For continuous input features, it is necessary to measure the combined entropy score for each candidate attribute-pivot pair. If we try every possible pivot, this could be very computationally expensive. A more efficient implementation is to sample a reasonable number of pivot values (by randomly picking samples from the training data), and choosing the one that results in the lowest combined entropy score.

2.3.2.2.1 Laplacian smoothing

It is important to remember that the training data is just a sample from some system that we are trying to model. It is the system that we are really interested in modeling, not the data. When we have plenty of samples, the ratios we measure are likely to represent the system quite well. But each time we divide the data, the number of samples diminishes, so our confidence that they correctly represent the system should probably diminish as well.

A simple way to work this concept into the computation is to pretend that there is always one more sample than we really have, with its value evenly divided among all possible classes. For example, suppose our data has a binary class label. Suppose we are down to just 3 samples in the current branch of the tree, and 2 of them have the label “true”, and 1 of them has the label “false”. Instead of calculating the entropy as

$$-\frac{2}{3}\log_2\left(\frac{2}{3}\right)-\frac{1}{3}\log_2\left(\frac{1}{3}\right),$$

we hypothetically add one additional sample, so we add 1 to the denominator, and add $1/c$ to the numerator, where c is the number of possible classes,

$$-\frac{2.5}{4}\log_2\left(\frac{2.5}{4}\right)-\frac{1.5}{4}\log_2\left(\frac{1.5}{4}\right).$$

When the number of samples is large, this will have very little effect on the computed entropy. When the number of samples is small, it will have a greater effect.

We don't have to smooth with exactly one smoothing sample. We could add l such samples, where l is any number, even a fractional number. Of course, the best value for l depends on the problem, so this becomes just another “knob” that you can tune when you need to optimize your decision tree learner for a particular problem.

2.3.2.3 Oblique trees

An *oblique tree*, also called a *linear combination tree* or a *perceptron tree*, is a decision tree that divides on a linear combination of attributes instead of dividing on axis-aligned boundaries.

Oblique trees are designed to operate on continuous input features. Data with categorical input features is typically converted to use a continuous representation before it is used to build an oblique tree.

Like random trees, oblique trees are not usually as effective as entropy-reducing trees at generalizing, however, they have been shown to be effective at promoting diversity, which can be a significant advantage when they are used in ensembles.

For example, suppose your data has a binary class label, and you want to quickly find a line that roughly separates them. (Note that it is not necessary to achieve perfect separation with this one division, because we will recursively divide again if necessary.) Let μ_{true} be the centroid of the points labeled true, and μ_{false} be the centroid of the points labeled false. Let
$\mathbf{c} = (\mu_{\text{true}} + \mu_{\text{false}})/2$, and let $\mathbf{d} = \mu_{\text{true}} - \mu_{\text{false}}$. So, $\mathbf{c}$ is a point between the centroids of the two clusters, and $\mathbf{d}$ is vector that can be said to point in the direction from $\mathbf{c}$ toward μ_{true} (and away from μ_{false}). $\mathbf{c}$ and $\mathbf{d}$ are all we need to define an oblique line that roughly separates the two classes. If a new point, $\mathbf{p}$ is given you can compute $(\mathbf{p} - \mathbf{c}) \cdot \mathbf{d}$ to determine on which side of the line $\mathbf{p}$ falls. If this value is less than 0, then $\mathbf{p}$ is on the false side of the line.

Note that this is not the the best possible dividing line. It is just a quick-and-easy one that we can compute. If we want to further refine it, we could do this with one of the iterative refinement approaches that we will discuss later in this book.

In the case where the class labels are not binary, we can divide them into two clusters anyway. For example, we could represent the labels in continuous vector form, and then divide on the hyperplane orthogonal to the first principal component of these vectors. This would group the labels into two clusters as equally as possible.

2.3.2.4 Nonlinear divisions

The divisions used by a tree need not necessarily even be linear. In this section, we give one example of a common nonlinear division.

It might be reasonable to assume that all of the points with a common class label are distributed approximately as a Gaussian point cloud. In other words, we assume that they follow a multivariate normal distribution. So, to represent each class, we would compute its centroid, μ , and its covariance, $\mathbf{C}$, from the training data.

(Note that “Σ” is often used in the literature to represent covariance. Unfortunately, that can be a little confusing because the same letter (uppercase Greek sigma) is also frequently used to mean *sum*. For clarity, we will use “$\mathbf{C}$” for covariance and
“Σ” for sum, but the reader should be aware that much of the literature uses “Σ” for *covariance* as well as *sum*.)

If there are *d* input dimensions in the input portion of the training data, $\mathbf{X}$, then the covariance, $\mathbf{C}$, is the $d \times d$ matrix where,

$$\mathbf{C}_{i,j} = \frac{\sum_{k=1}^{k \le n}(x_{k,i} - \mu_i)(x_{k,j} - \mu_j)}{n - 1}.$$

The division is defined by μ and $\mathbf{C}$. The probability density function (PDF) of the multivariate normal distribution uses these values to estimate the likelihood that each pattern, $\mathbf{p}$, belongs with each side.

The full PDF of the multivariate normal distribution is

$$\frac{1}{\sqrt{(2\pi)^d \det(\mathbf{C})}} e^{-\frac{1}{2}(\mathbf{p}-\mu)^{\mathrm{T}}\mathbf{C}^{-1}(\mathbf{p}-\mu)},$$

but since we only need to know which cluster is the most likely, we really only need to compute which cluster maximizes the expression,

$$-(\mathbf{p} - \mu)^{\mathrm{T}}\mathbf{C}^{-1}(\mathbf{p} - \mu).$$

We then divide the training data, such that each point goes with the side of greatest predicted likelihood, and recurse until the labels are homogeneous, or some other stopping criteria are met.

At prediction time, we use the same approach to determine on which side of each division **p** belongs. (The matrix inversion in the equation can be done at training time, so matrix-vector multiply is the only significant operation necessary to make the predictions.)

If there are exactly 2 class labels, then divisions made in this manner will be multivariate conic cross-sections. That is, the divisions will be parabolas, ellipses, and hyperbolas.

2.3.3 Reducing overfit

Since decision trees recursively divide the data until the labels are homogeneous, they have a strong tendency to over-fit to noise in the training data.

One very simple approach to limit overfit is to stop dividing when there are fewer than *k* data points in the current branch.

A more complex technique is *reduced error pruning*. You systematically try removing each node in the tree, starting with the leaf nodes. If removing the node improves accuracy with a hold-out set, then you keep going. If removing that node makes the accuracy worse, you put it back and stop.

2.4 Rule learning

Rule learners are learning that train models consisting of *rules*. A rule is a mapping from a partially-specified list of attributes and values to a label vector. Examples:

> "if x_1 == 1.11 and x_2 >= 0.0 and x_3 == true,
> then y_1 = 3.1415, y_2 = potato",
> "if age < 16 and gender == male,
> then label = boy",
> "if sky == cloudy and barometer > 31.1,
> then chance_of_rain = high",

One useful way to think of rule learners is as a generalization of both instance-based learning and decision-tree learning. In instance-based learning, each instance may be considered to be a fully-specified rule. That is, a value is given for every attribute in the instance. In decision trees, each node may be considered to be a minimally-specified rule. That is, a value is specified for only one attribute in order to specify a decision boundary. Rule-based learners essentially reside somewhere between these two extremes, by forming rules that specify values for some, but usually not all, attributes.

Imagine a single instance in 3-space. Suppose you pick some distance, and find all the points that are exactly this distance from the instance. All of those points would form a ball. Now, suppose that one of the values in that instance is no longer specified. How do you measure the distance to an unspecified value? Well, that's a tricky problem, so let's just ignore that attribute for now. Instead, we will just measure distance in the attributes that are specified, and assume that the unspecified ones are irrelevant. Now all of the points that are the same distance from the partially-specified instance form a cylinder.

Is a cylinder more like a ball or a plane? Well, in the specified dimensions it is curved, like a ball. In the unspecified dimensions, it is straight, like a plane. Instance-based learners essentially rely on balls around each instance. Decision trees essentially rely on planes to divide the space. Rule learners do a little of both at the same time. Now, imagine what happens in *n*-space. The more attribute-values are specified in a rule, the more the shape at equidistant points resembles a ball. The fewer attribute-values are specified in the rule, the more it resembles a plane.

When we work with continuous attributes, precise values are unlikely to ever occur more than once. (For example, if Alice is 160.0304951 centimeters tall, and Betty is 160.0304952 centimeters tall, are they the same height? Technically no, but it would probably be a mistake in most applications to completely ignore the surprisingly close similarity of these measurements.) Instance-based learners typically address this problem by measuring distances that give some weight to the instance. By contrast, decision trees typically address this problem by specifying inequalities in the decision nodes instead of specifying exact values. Which approach is better to use with rules?

Well, the answer to this question is not exactly clear, yet. Should the rules be organized into an ordered hierarchy, like a tree? Or should the most relevant rules be identified based on some weighting? What is the best way to combine rules?

A rule could specify both a center and a radius. Then, it could use a distance and an inequality. For example, “if $||x_7 – 2.34|| < 1.5$ and $||x_9 – 5.1|| >= 6.0$ then ...”

Alternatively, rules could specify exact values, and they could be combined based on some weighting scheme. Many things could be done.

As you might recall, a *k*-nn model can be a baseline learner (if *k*=*n*), and it can be a nearest neighbor model (if *k*=1). Well, a rule learner can be a *k*-nn model (if its rules are fully specified), and thus it can be a generalization of both models. A rule learner can also be a decision tree (if the rules are minimally specified, use inequalities for continuous attributes, and are organized hierarchically). To make our rule learner as general as possible, we might even use some form of L^P-norm distance, which generalizes several interesting distance metrics, and combine results using a weighting of $1/d^q$, where *d* is the distance, and *q* is a parameter that controls the weighting. Now, we have created a super-powerful learner that can represent many interesting models across many spectra! There's just one problem—how do we train it?

Several algorithms exist for generating models comprised of rules. Unfortunately, none of them have yet emerged as clearly being the right way to do it. Most of them are only designed to manage rules of a particular form that are combined in a particular manner, and none of them have really demonstrated to be consistently superior to both instance-based learners and decision trees at the same time. So, rule learning really remains an open area for research.

One discouraging way to analyze the problem of training a rule learner is to show that it maps to a satisfiability problem. In fact, satisfiability problems are usually presented as a list of rules, so this mapping is really quite straight-forward. Since satisfiability problems are proven to be NP-complete, it follows that there may very well be no general solution to training a general rule learner. Of course, many things that cannot be solved exactly can still be approximated very well. So, at this point, they are not very practical, but they still seem to have a lot of potential.

2.5 Ensemble techniques

Imagine that after a routine checkup with your medical doctor, she informs you that you are exhibiting subtle symptoms you didn't even notice, and they suggest that you have a rare condition that will result in your certain death within the next six months. What would you do?

Well, before you make any drastic lifestyle changes, it might be a good idea to first seek a second opinion.

Now, suppose that a panel of 12 expert doctors review your case, and 10 of them agree with the original diagnosis. Now, how confident are you in the diagnosis?

What if you discovered that the two dissenters came from independent medical schools, while the 10 doctors who agreed were all closely related, and educated at the same school in the same class as your original doctor? How would that affect your confidence in the majority opinion of the panel?

An ensemble is a model that combines several other models to, hopefully, achieve greater predictive power. When the right conditions are created, an ensemble can make more reliable predictions than any of its individual models.

2.5.1 Basic theory

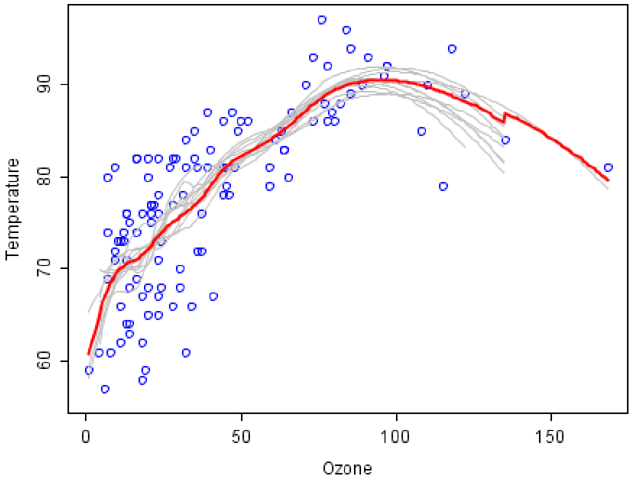
In theory, the hypothesis of the ensemble could be any combination of all the possible hypotheses of its models. In other words, the model space of an ensemble is the outer product of the model spaces of its constituent models. In practice, however, it is difficult to find training data that will drive the various models to opposite extremes in their model spaces. Because the hypotheses that an ensemble combines were all obtained using the same training data, they are often quite correlated. The differences between these hypotheses tend to be due to the limitations and biases of the learning algorithms that produced them. So, combining them tends to have the very nice effect of reducing the overall limitations and biases of the learning algorithms they combine together.

Of course, this nice property comes with a computational cost. Since multiple models must be trained, training takes longer. Since multiple models must make predictions, predicting also costs more too.

2.5.2 Bootstrap aggregation

Bootstrap aggregation, often abbreviated *bagging*, is a simple-but-powerful ensemble technique. It combines models by giving them each equal vote in the prediction. For categorical labels, it predicts the value that is most common among the predictions of the models it aggregates.

For continuous labels, it predicts the mean of the predictions of the models it aggregates.



In this visualization, 100 models were trained to fit to a set of points. Predictions from 10 of them are shown in gray. Predictions from a bagging ensemble of all 100 models is shown in red.

In order to promote diversity among the models, bagging trains each model using a slightly different subset of the training data. Let **T** be the available table of training data. Let **S** be an initially-empty table of data. Bagging picks a random training sample from **T** and adds a copy of it to **S**, until **S** is the same size as **T**. The samples are not removed from **T**. This is called *sampling with replacement*. (Technically you would have to first remove the sample from **T** before you could then replace it, but that would just be unnecessary processing. It's easier to just leave it there.) Note that sampling with replacement will result in some samples appearing multiple times, and some samples not appearing at all. This is deliberate. Each model in the ensemble is trained with a unique resampling, **S**.

Of course, **S** doesn't absolutely have to be the same size as **T**. You could very well add a “knob” to your bagging ensemble that would let you adjust the ratio of the size of **S** compared with the size of **T**. Then, you could tune this knob to optimize your ensemble for your data.

2.5.2.1 Forests

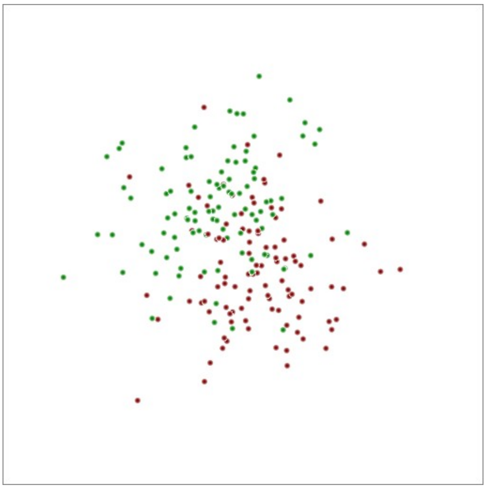
When decision trees are combined in a bagging ensemble, it is sometimes called a “forest”. (Ha ha, lots of trees make a forest ...*golf clap.) Since decision trees tend to be quite volatile, the small changes caused by resampling the training data can result in significantly different trees, which effectively adds essential diversity to the ensemble. In theory, it would probably be preferable to add a diversity of algorithms to the ensemble, but it is not always practical to gather many learning algorithms together. Furthermore, it can be difficult to determine whether differing algorithms really bring diversity, or whether they merely reinforce each others' implicit biases. So, using many instances of a volatile learning algorithm, like decision tree, often results in a pretty-good bagging ensemble.

In particular, random trees are well-known to do well in bagging ensembles. The random divisions that occur early in each tree ensure that the later divisions are all trained in a unique context. Thus, the similarities between random trees tend to be things that help make good predictions, rather than things that are merely common due to properties of the model itself.

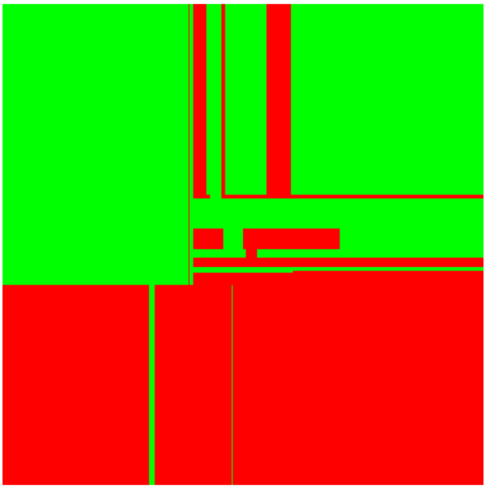
Unfortunately, every learning algorithm has a weakness. The weakness of forests is data that has few dimensions and contains lots of noise. Forests of random trees are even more vulnerable to overfit to the noise than forests of entropy-reducing trees.

2.5.2.1.1 Some visualizations

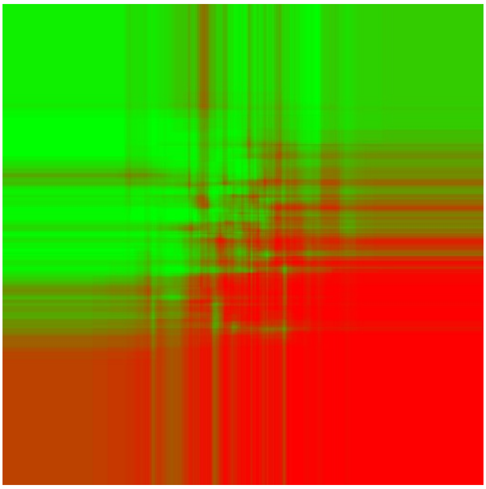
In this section, we show several visualizations produced by various forest algorithms. To make them interesting, we deliberately made a dataset that these algorithms would struggle to model effectively.



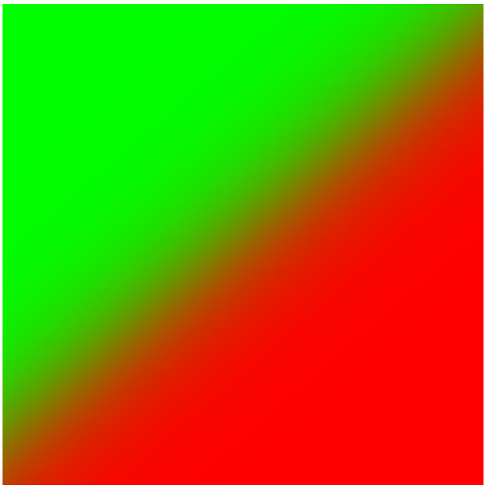
Our training data consists of two continuous input dimensions (represented as position on the horizontal and vertical axes), and one binary label dimension (represented with red and green color). The data points are distributed as Gaussian point clouds with the same variance, but slightly different centroids.



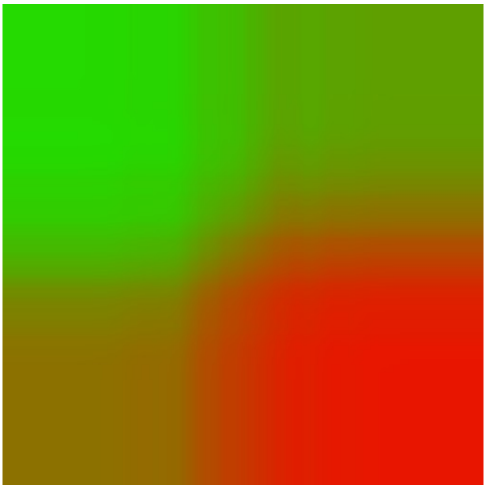
We trained a single decision tree using this data. This plot shows the label (with red or green color) that this tree predicts for every region in the input space.



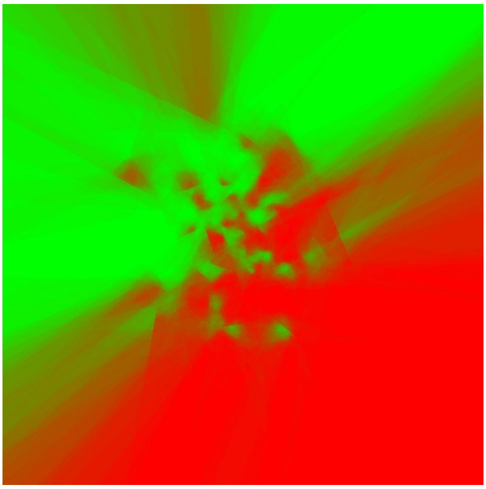
We trained a bagging ensemble of 50 random trees on the same data. In this case, we mixed red and green color according to the mix of predictions of the trees in the ensemble. (We also tried an ensemble of 2000 random trees, but its model space was visually indistinguishable from this one. Increasing the number of trees usually improves the model, but it does not necessarily prevent the model from overfitting to noise.)



For the sake of comparison, this model was produced by logistic regression, an unrelated technique that has nothing to do with bagging or trees. These results are very good, because they make predictions consistent with the system we used to generate the training data. Hence, this model will generalize very well. Although logistic regression gives much better results with this particular problem, forests tend to do much better than logistic regression with a wide range of difficult problems (not shown here).



To reduce the overfit of our forest of random trees, we limited the depth of each tree to 4, and increased the number of trees to 2000. This solved the overfit problem, but did not quite make the model as ideal as the one produced by logistic regression.



We also trained a forest of 50 oblique trees. Notice that the decision boundaries are not axis-aligned in this visualization.

2.5.3 Boosting

Todo: write this section

2.5.4 Bucket of models

A *bucket of models* is an ensemble technique that uses some model selection technique to pick the best model for each problem. *Model selection* is the process of identifying the best model to use for a particular problem. The most common approach for model selection is *cross-validation selection*, which can be summarized as “use cross-validation to evaluate each model using the training data, and select the one that does best”.

When accuracy is all that matters, cross-validation selection is the best model selection technique available. Unfortunately, it can be an expensive approach if some the models are expensive to train. Specifically, if you have *m* models, and you use *n*-fold cross-validation to evaluate them, then each of your *m* models will be trained *n* times (once per fold) before you know which model is probably the best. Then, when you know which model is probably the best, you need to train it one more time using all of the available data. (Why not just use one of the model you trained during cross-validation? Well, you could do that if you want, but those models were not trained using all of the available data. Specifically, each was trained using just $(n-1)/n$ folds of the data.)

2.5.5 Gating

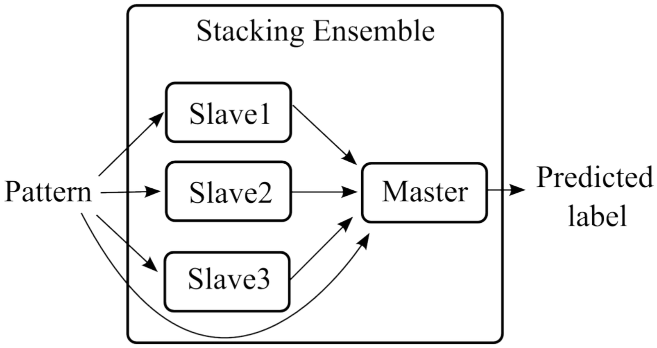
Gating is an ensemble technique in which a *master* model decides which of a collection of *slave* models is best-suited to evaluate each pattern. (It differs from cross-validation selection in that cross-validation selection picks a single model to handle the entire problem, whereas gating picks a single model for each instance or pattern that needs a prediction.)

A gating ensemble is trained by dividing the labeled data into two parts. All of the slave models are trained using one part of the data. With each sample in the other part of the data, the slave that makes the best prediction is identified. The master model is then trained using this information, so that it can predict which slave model is best-suited to handle each pattern. When a prediction is needed, the master is first asked to identify the best slave for the job, and then that slave is asked to make a prediction.

2.5.6 Stacking

Stacking is similar to gating, but the master is trained to combine the predictions of all the slaves, rather than to merely identify which slave is best for each job.

Like gating, the labeled training data is divided into two portions. The slaves are all trained on one portion. The slaves then predict labels for all the samples in the other portion. Unlike gating, however, the master is then trained on an augmented dataset consisting of the input patterns, the predictions made by each slave, and the correct label vectors. When a prediction is needed, all of the slaves first make their predictions, then the master will combine the input pattern with all of the slave predictions to produce the input it needs to make a final prediction.



Stacking may be considered to be a generalization of all other ensemble techniques, because the master model could potentially be designed in a manner that causes the stacking ensemble as a whole to behave exactly like any other type of ensemble. In practice, however, it can be difficult to find suitable models for particular tasks. Thus, stacking has a reputation for being somewhat awkward and cumbersome to work with. Yet, when used effectively, stacking has produced better predictive accuracy results than many other techniques with some challenging problems.

2.5.7 Bayes' optimal classifier

What is the theoretically best way to make a prediction? It is to only listen to the correct hypothesis, of course. But, what if the correct hypothesis is not in the ensemble? Well, since we're in theory-land, let's put every possible hypothesis in our ensemble. That way, we can be sure that the correct one is in there somewhere. Some models, including *k*-NN, decision trees, and multi-layer perceptrons, are *universal function approximators* (a.k.a *arbitrary function approximators*). These are models that can memorize the training set, and thus can represent any hypothesis function within a negligible amount of precision. So, if we use such a model, then at least one of its possible hypotheses is close enough. Unfortunately, universal function approximators tend to have hypothesis-spaces of infinite or near-infinite size. Since we're in theory-land, we'll just pretend we have a Turing machine, and press forward.

Okay, but now the problem is that we still don't know for sure which hypothesis is the correct one. So, what do we do? According to probability theory, the best way to make decisions when there is uncertainty is to weight each possibility according to the probability that it is the correct one. So, the best-possible prediction that can ever be made when we are uncertain about which hypothesis is correct is described by the distribution,

$$P(\mathbf{y}|\mathbf{x}) = \sum_h \left\{ \begin{array}{rl} P(h|\mathbf{D}) & \text{if } h(\mathbf{x}) = \mathbf{y} \\ 0 & \text{if } h(\mathbf{x}) \neq \mathbf{y} \end{array} \right.\text{'}$$

where $\mathbf{x}$ is the an pattern, $\mathbf{y}$ is the label for which we would like to know the ideal predicted probability, $h(\mathbf{x})$ is the prediction made by each hypothesis, h, and $P(h|\mathbf{D})$ is the probability that $h$is the right hypothesis given the training data, $\mathbf{D}$. In other words, we ask all of the hypotheses to make a prediction, and we weight all of their predictions by the probability that this is the correct hypothesis given the training data.

If the hypothesis returns a distribution over predicted label vectors, instead of just a single predicted label vector, then we can rewrite our equation as,

$$P(\mathbf{y}|\mathbf{x}) = \sum_h P(\mathbf{y}|h)P(h|\mathbf{D}),$$

where $P(\mathbf{y}|h)$ is the probability that $h(\mathbf{x}) = \mathbf{y}$.

So, how do we evaluate $P(h|\mathbf{D})$? In English, we are asking, *what is the probability that a hypothesis, h, accurately describes the source of data,* *given some observed data,* $\mathbf{D}$*?* How does one measure such a thing? Unfortunately, this is a difficult problem. No one really knows how to measure that. So instead, we use Bayes' law,

$$P(a|b) = \frac{P(b|a)P(a)}{P(b)},$$

to break it down into smaller pieces. In this case, a is the hypothesis, and b is the data. So, plugging in Bayes' law gives us,

$$P(\mathbf{y}|\mathbf{x}) = \sum_h \frac{P(\mathbf{y}|h)P(\mathbf{D}|h)P(h)}{P(\mathbf{D})}.$$

Now, we have even more terms that we need to quantitatively evaluate. Is this really better? Well, lets see if we can do a better job of assigning meaningful numbers to these terms.

First, let's convert them to English. The term, $P(y|h)$, is the probability that $h(\mathbf{x}) = y$. The term $P(\mathbf{D}|h)$ is the probability that the training data might occur in a universe where h is the correct hypothesis. $P(h)$ is the prior probability that h is the correct hypothesis. (That is, it is the probability of h before we ever saw $\mathbf{D}$.) And finally, $P(\mathbf{D})$ is the prior probability that $\mathbf{D}$ might occur.

That last one is a doozy—there is no rational way we could possibly assign a meaningful number to it. Fortunately, if we are just trying to do classification, meaning pick the best categorical label for the pattern, instead of predict a whole distribution, then we only need the mode of this distribution. So, the best class label is,

$$\underset{\mathbf{y}}{\operatorname{argmax}} \sum_h P(\mathbf{y}|h)P(\mathbf{D}|h)P(h)\,.$$

Notice that we were able to drop the term, $P(\mathbf{D})$, because it does not depend on the value of $\mathbf{y}$. No matter what is the correct value for $\mathbf{D}$, the most-likely $\mathbf{y}$ is still the most-likely $\mathbf{y}$. All of the other terms do depend on the probability of $\mathbf{y}$. In other words, they are not constant with respect to the probability of $\mathbf{y}$, so finding the most-likely $\mathbf{y}$ does not enable us to drop these other terms. So, we're going to have to actually evaluate the other terms.

If we assume that each training sample is independent, which is a typical assumption in machine learning, then

$$P(\mathbf{D}|h) = \prod_{\mathbf{d}} P(\mathbf{d}|h),$$

where $P(\mathbf{d}|h)$ is the probability that each data instance, $\mathbf{d}$, might occur in a universe where h is the correct hypothesis. So, now we just need to use a model that generates hypotheses that will allow us to evaluate $P(\mathbf{d}|h)$. Some models make this term easy to evaluate. Some do not. We will just ignore those models for now.

So, the only term remaining is $P(h)$. This is the prior probability of h. That is, it is the probability that h might occur in a universe where we have not seen any training data yet. How can we determine that? Perhaps, the most common knee-jerk solution is to assume that all hypotheses are equally likely. Unfortunately, this is a very poor approach. As we will show next, such an assumption actually makes all learning completely futile! So, what is the right prior distribution over hypotheses? Well, to do it right, you would have to sum up all knowledge that has yet been obtained about the universe until the moment just before the training data was obtained. That task would be somewhat difficult :). Perhaps, a reasonable approximation is given by Occam's razor. In this context, it suggests that hypotheses should not be more complex than is necessary to explain the training data. Since $P(h)$ refers to a probability at the moment *before* the training data was obtained, this reduces to, *simple hypotheses are better*. So, we just need some way to evaluate the complexity of a hypothesis. For example, shallow decision trees should be give a higher prior probability than deep decision trees, because shallow trees are simpler. Since no one really knows for sure how to evaluate prior probabilities, we can just do our best and call it good enough.

So, Bayes' Optimal Classifier is the theoretically perfect ensemble technique. It is not really practical to implement, so why did we even study it? Because a common objective of ensembles is to combine models as well as possible, and that means they strive to be as much like Bayes' Optimal Classifier as possible.

The major assumptions of Bayes' Optimal Classifier are:

1. It assumes the ensemble contains the true hypothesis. (This is often solved in theory by using an arbitrary function

approximator, and throwing the entire hypothesis space into the ensemble.)

2. It assumes that the ensemble contains only models that can be used to predict distributions.
3. It assumes the user can supply a prior distribution over all hypotheses.

When ensembles try to approximate Bayes' Optimal Classifier, but ignore these assumptions, rather than address them, they generally accomplish little more than using a lot of computation to make very poor predictions.

2.5.7.1 No free lunch

Consider a simple sequence-predicting problem. You are given the following sequence,

> 0, 2, 4, 6, 8, 10, 12, 14, 16, 18,

and are asked to predict the next value. To a human, this problem is very simple. ...or is it? Well, that depends on what the right answer is. If the answer is 20, then most intelligent humans will get it right. But, what if the correct answer is 7?

A hypothesis is a mapping from every position in the feature space to corresponding labels. Let us temporarily make the assumption that all possible hypotheses are equally likely. This means that the hypothesis

> 0, 2, 4, 6, 8, 10, 12, 14, 16, 18 -> 7

is just as likely as

> 0, 2, 4, 6, 8, 10, 12, 14, 16, 18 -> 20.

In fact, there must be a hypothesis with every possible label following this same sequence. If we assume that they are all equally likely to be correct, then how can we ever make accurate predictions? Well, simply put, you can't.

This same principle works in every domain of machine learning, not just sequence prediction. For example, in regression, suppose that wildly jumping curves are just as likely as smooth continuous curves. Could you ever predict a value for inputs you had never seen before? No, because it is just as likely as not that the true curve jumps wildly around that point, making the correct prediction impossible to even guess.

As an intuitive analogy, suppose you observe the temperature rise every summer and fall every winter. One hypothesis is that the temperature loosely follows a periodic cycle. Another hypothesis is that all patterns in temperature over the past several million years are entirely coincidence, and there is absolutely no way to predict what the temperature will be tomorrow. This implies that any success that any meteorologist has had so far has just been dumb luck, and will probably not occur again. Are these two hypotheses equally likely? Certainly not! We do not live in a universe where all hypotheses are equally likely, so assuming that we do is utterly ridiculous.

So, what can we conclude from all of this reasoning? Well, it shows that the assumption that all hypotheses are equally likely is absurd. It follows that using a uniform prior distribution over hypotheses in Bayes' optimal classifier will result in garbage predictions. If we want reasonable results, we must choose a prior distribution over hypotheses that is somewhat more consistent with the universe.

We might ask, then, if Bayes' optimal classifier, which is the most-powerful classifier in all of machine learning theory, cannot make good predictions with uniform priors, then why do other learning algorithms work at all? The answer is because all effective learning algorithms implicitly use a non-uniform prior distribution over possible hypotheses. To put it another way, they are all biased in some way. This bias causes them to favor some hypotheses over others, even when the training data does not tell it to do so. An unbiased learner could never learn. Since we, as humans, can learn, we know that learning is not futile—it just requires knowing something (also called “having a bias”) about how the universe tends to operate.

It follows that no matter how effective some learning algorithm may be, there must exist (in theory) some data that would cause it to give terrible results. Fortunately, the universe seems to have a lot of bias too (generally toward simplicity), so biased algorithms tend to give good predictions. Thus, the ultimate learning algorithm, would not be one that is unbiased, but rather, would be one whose implicit bias matches that of the universe in which it operates.

2.5.8 Bayesian model averaging

Todo: write this section

2.5.9 Bayesian model combination

Todo: write this section

2.6 Some meta-analysis

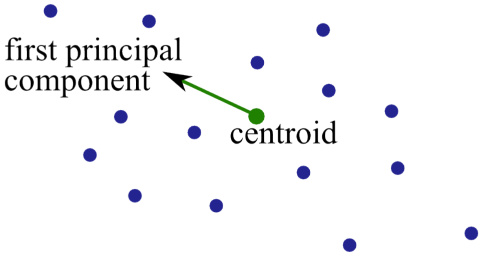
Todo: write this section

	Baseline	*k*-NN	Decision tree	Forests
Trains efficiently	■	■	■	
Predicts efficiently	■		■	
Good accuracy		■	■	■
Great accuracy				■
Compact model	■			
Can handle sparse data well	■	■		

2.7 Dimensionality reduction

Todo: introduce the topic

The first principal component is a vector that points in the direction of greatest variance. (Since variance is bi-directional, it does not matter if you negate a principle component vector. Either direction means the same thing.)



Here is a simple algorithm to quickly estimate the first principal component of a set of points, $\mathbf{S}$:

```
function estimate_first_principal_component($\mathbf{S}$)
    Center $\mathbf{S}$ about the origin by subtracting the
        the centroid from each point in $\mathbf{S}$.
    Let $\mathbf{p}$ be a random vector about the origin (of
        the same size as the points in $\mathbf{S}$).
    do 10 times:
        Let $\mathbf{t}$ be a vector of magnitude 0.
        for each point $\mathbf{s} \in \mathbf{S}$:
            $\mathbf{t} = \mathbf{t} + (\mathbf{s} \cdot \mathbf{p})\mathbf{s}$
        $\mathbf{p} = \mathbf{t}/||\mathbf{t}||$
    return $\mathbf{p}$
```

2.7.1 Feature selection

Often, data contains irrelevant attributes. These attributes make the data bigger, but they do not actually improve predictions. Sometimes, they even reduce accuracy. *Feature selection* is the process of determining which attributes are important, so you can throw out the rest.

The brute force approach to feature selection is to try every possible combination, with some attributes included, and other attributes excluded. For each combination, cross-validation is performed. The combination that produces the best score contains the features that should be selected.

Unfortunately, if there are d attributes, then the brute force approach requires that you perform cross-validation 2^d times. This is only tractable if there are already very few attributes. The most common reason for performing feature selection, however, is because there are too many of them. Therefore, the brute force method is pretty-much worthless.

The following pseudocode describes a more efficient method for attribute selection:

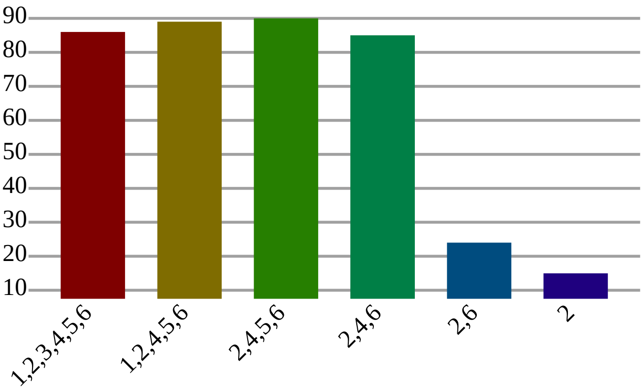
```
function greedyAttributeSelection()
  Let D be a dataset containing all attributes.
  While D contains at least one attribute:
    For each attribute, c, in D:
      Remove c from D.
      Perform cross-validation on D.
      Restore c back into D
    Again, remove the attribute whose removal
      resulted in the best accuracy.
```

This algorithm produces a ranked ordering of all the attributes. The first attribute that it removes is the most worthless attribute. The last attribute that it removes is the most important one.

If there are d attributes in D, then this algorithm will require that you perform cross-validation $d + (d-1) + (d-2) + \cdots = d(d+1)/2$ times. That's a lot better than 2^d times!

Note that it may produce different results depending on the learning algorithm you use with it. It does not really identify which attributes are least important. Rather, it identifies which attributes your learning algorithm is least able to exploit in order to achieve higher accuracy. In most cases, that is what you really want to know anyway.

The accuracy scores that you measure with each attribute that you remove can also be useful for determining how many to select. For example, suppose your data contains 6 attributes. The following bar chart might represent the accuracy scores that you obtain:



This shows that attribute 3 was the most worthless (because it was the first one removed), and attribute 2 was the most important (because it was the last one remaining). Note that removing attributes 3 and 1 actually improved the accuracy, so those two should definitely be removed. We can see that attributes 2 and 6 contain nearly all of the important information in this dataset, so those two should definitely be kept. But, what about attribute 4? Should we remove that one? Well, it doesn't contribute very much, but it does appear to improve the accuracy a little bit. So, that is a judgment call that someone needs to make. Perhaps it is worth keeping attribute 4 around for the tiny bit of extra accuracy that it provides. Perhaps, simplicity is more important. It really depends on why you are doing attribute selection.

Now, instead of starting with all the attributes and removing them one at-a-time, what if we start with an empty dataset and add them one at-a-time? Wouldn't that be even faster? Unfortunately, this approach does not usually work very well in practice. Can you guess why not?

Sometimes the value of certain attributes is found in their combination. As a simple example, suppose you had several candidate flying contraptions, and you wanted to train a learning algorithm to predict which ones might successfully fly. Suppose all the contraptions with two wings fly, and those with one or fewer wings do not. Suppose the attributes for your learning algorithm are: 1. Does it have a wing on the left side?, 2. Does it have a wing on the right side?, and 3. Is it painted blue? If we remove these attributes one at-a-time, we will immediately find that the color of the paint is not very relevant to its ability to fly. By contrast, if we start with an empty dataset and add the attributes one at-a-time, we will find that none of them are individually very good predictors of its ability to fly (because it takes two wings to fly). Hence, the attributes that we identify as most important are selected somewhat arbitrarily.

Neither greedy approach is perfect. If we remove attributes one at-a-time, we end up being more arbitrary about the ones we determine to be least-important. If we add them one at-a-time, we end up being more arbitrary about the ones we determine to be most-important. In practice, removing them one at-a-time seems to work better.

Todo: talk about how logistic regression is well-suited for attribute selection because it implicitly assigns a weight (or rank) to all attributes. This significantly reduces the number of times that cross-validation needs to be performed.

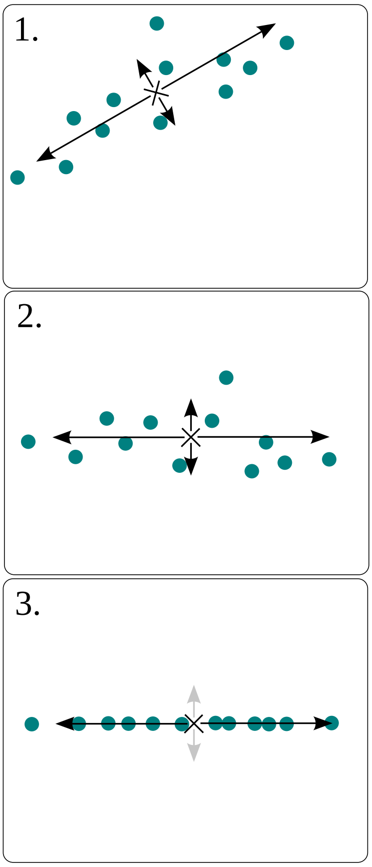
2.7.1.1 Optimization-based approaches

Todo: talk about how beam search can be used for attribute selection problems. Also, evolutionary optimization has been used for this purpose.

2.7.2 Principal component analysis

Instead of seeking the *k* attributes that contain the most information, we could seek the *k* linear combinations of attributes that contain the most information. In general, this allows us to cram more information into just *k* attributes.

In the following illustration, we will suppose that you have some data that consists of two continuous attributes (or dimensions). To visualize this data, we plot each data vector as point, with one attribute on the horizontal axis, and the other attribute on the vertical axis.



PCA can be described intuitively in 3 steps:

1. Compute the centroid of the data. This is shown in the figure with a × symbol. Compute the direction of greatest variance in the data. This is called the *first principal component*. It is shown in the figure with long arrows. Compute the direction of second-greatest variance. This is called the *second principal component*. It is shown in the figure with the shorter arrows.
2. Project the data onto the principal components. This is the same as moving the centroid to the origin, and then rotating the data until the principal components are aligned with the axes.
3. Keep only the dimensions that have a lot of variance.

Whereas attribute selection tries to determine which attributes to discard and which attributes to keep, principal component analysis performs a linear transformation that moves as much of the information as possible into the first several attributes. Then, these first several attributes are obviously the ones to keep, and the last several attributes are the ones to discard.

2.7.3 Multidimensional scaling

2.7.4 Isomap

2.8 Transduction

Todo: write this section

2.9 Agglomerative methods

Todo: write this section

complete link, etc.

2.9.1 Graph-cut transduction

Todo: write this section

2.9.2 k-means

Todo: write this section

todo: for collaborative filtering

2.9.3 Mixture of Gaussians

Todo: write this section

2.10 Incremental learning

Todo: write this section

2.11 Structure prediction

Todo: write this section

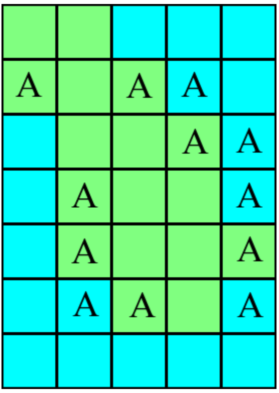
2.12 Active learning

Todo: write this section

3 Statistical learning

3.1 Probability theory

Suppose you are in charge of choosing a location to drill for oil. To keep the problem simple, you have divided your map into a 5×7 grid. Dry land is indicated with green, and water or swampy areas are indicated with blue. Regions that are known to be infested with alligators are indicated with the letter "A". (I don't know why that would matter to a drilling company —it's just a silly example. Perhaps alligators make the land cheaper, but also complicate drilling efforts by bothering the workers.)



🟩	🟩	🟦	🟦	🟦
🟩 A	🟩	🟩 A	🟦 A	🟦
🟦	🟩	🟩	🟩 A	🟦 A
🟦	🟩 A	🟩	🟩	🟦 A
🟦	🟩 A	🟩	🟩	🟩 A
🟦	🟦 A	🟩 A	🟩	🟦 A
🟦	🟦	🟦	🟦	🟦

If you choose a region on the map completely at random, what is the probability that you would choose a location on dry land? That's easy, just count the number of green squares, and divide by the total number of squares:

$$P(\text{land}) = \frac{17}{35}.$$

The uppercase "*P*" means "probability of". We can also count to determine the ratio of regions that are infested by alligators:

$$P(\text{alligator}) = \frac{12}{35},$$

We don't need to count again to determine the probability of choosing a location in water,

$$P(\text{water}) = 1 - P(\text{land}) = \frac{18}{35},$$

or the probability of picking a location without alligators,

$$P(\neg\text{alligator}) = 1 - P(\text{alligator}) = \frac{23}{35}.$$

(The “$\neg$” symbol means “not” or “negate”. “$\wedge$” means “and”, “$\vee$” means “or”.)

We could use counting to identify all of the possible conjuncts:

$$P(\text{land} \wedge \text{alligator}) = \frac{7}{35},$$

$$P(\text{land} \wedge \neg\text{alligator}) = \frac{10}{35},$$

$$P(\text{water} \wedge \text{alligator}) = \frac{5}{35},$$

$$P(\text{water} \wedge \neg\text{alligator}) = \frac{13}{35}.$$

Note that these probabilities are <u>**NOT**</u> simply the product of the two values they combine.

$P(\text{land}) = \frac{17}{35}$, and $P(\text{alligator}) = \frac{12}{35}$, but
$P(\text{land} \wedge \text{alligator}) \neq \frac{17}{35} * \frac{12}{35}.$

Now, suppose the boss decides to let the workers have a vote on whether the next drilling job is performed on land or in water. Suppose they don't really care about finding oil—they just don't want to work near any alligators. So which way will they vote?

Perhaps, the answer depends on how capable they are at working with probabilities. Specifically, the values they are most-interested in should be:

$$P(\text{alligator}|\text{land}),$$

and

$$P(\text{alligator}|\text{water}).$$

The “|” is read “given”, and means that you want to know the probability of what precedes it, given that what follows it is known to be true. If they know this, then whichever gives them a smaller probability of encountering alligators will determine which way they vote. We can use the following identity to calculate these terms,

$$P(a \wedge b) = P(a|b)P(b).$$

Or, if we rearrange the terms a little bit, we get,

$$P(a|b) = \frac{P(a \wedge b)}{P(b)}.$$

So, the workers should calculate,

$$\begin{aligned} P(\text{alligator}|\text{land}) &= \frac{P(\text{alligator} \wedge \text{land})}{P(\text{land})} \\ &= \frac{7/35}{17/35} = \frac{7}{17}, \end{aligned}$$

$$\begin{aligned} P(\text{alligator}|\text{water}) &= \frac{P(\text{alligator} \wedge \text{water})}{P(\text{water})} \\ &= \frac{5/35}{18/35} = \frac{5}{18}. \end{aligned}$$

(You can confirm these ratios by counting on the map. Of the 17 land squares, 7 of them have alligators, and of the 18 water squares, 5 of them have alligators.)

So, since ocean-drilling apparently involves fewer incidences of alligators, the workers who fear alligators will vote for the next drilling job to be in the water.

Let's suppose that the workers believe they have an 80% chance of winning the vote to drill in water. What is the probability that they will be asked to work near alligators?

$$\frac{80}{100}\frac{5}{18} + \frac{20}{100}\frac{7}{17} = \frac{233}{765}.$$

How long is this probability valid? Well, it lasts until they get some more information. A probability is just an expression of belief in the face of uncertainty. Every new piece of information should have some impact on beliefs. (If not, then you have ceased to learn.) So, when the outcome of the vote is known, the probability will change. When the location of the next drilling site is announced, it will change again.

The ultimate tool for machine learning would be a formula that tells us how to update our beliefs whenever some new information becomes available. If we had such a tool, and it was reasonably efficient, then we could use it to implement learning systems. For every new observation, it would update our beliefs to just the right extent—not too much, and not too little.

In the quest for such a formula, let us examine what we know about probabilities a little closer. We know that

$$P(a \wedge b) = P(a|b)P(b).$$

It should also be pretty obvious that

$$P(a \wedge b) = P(b \wedge a).$$

So, if we combine these two rules, we get,

$$P(a|b)P(b) = P(b|a)P(a).$$

This is called Bayes' law. It is basically a probabilistic way of saying, *when you know two things, you know two things, regardless of which one you learned first.*

If we rearrange the terms a little bit, it gets even more interesting,

$$P(a|b) = \frac{P(b|a)P(a)}{P(b)}.$$

Now it is saying, *if you already know $P(a)$, and then you learn something new in the right form, $P(b|a)$ and $P(b)$, then here is the right way to update your beliefs about a.* Bayes' law is almost a complete formula for learning. Its limitations are, 1) It only works if you already know something of interest in the form of a probability distribution, and 2) The new knowledge must also come in a suitable form. For some problems, those are pretty severe limitations. For many problems, however, Bayes' law describes the best possible way to learn. It is, therefore, an essential tool in the field of machine learning.

Each of the four terms in Bayes' law have names:

$P(a)$ is called the *prior*. It describes what you already know or believe, before you obtain the new information.

$P(a|b)$ is the *posterior*. It describes the updated beliefs, after the new information has been obtained.

$P(b|a)$ is the *conditional*. It is the new information that changes your beliefs.

$P(b)$ is the *marginal*. It is essentially a normalization term that scales your distributions to the right size. As we will describe in later sections, we can often operate effectively in machine learning applications without even knowing this term.

[todo: also talk about DeMorgan's law,

$$\neg P(a \wedge b) = P(\neg a \vee \neg b)$$

]

3.2 Probability distributions

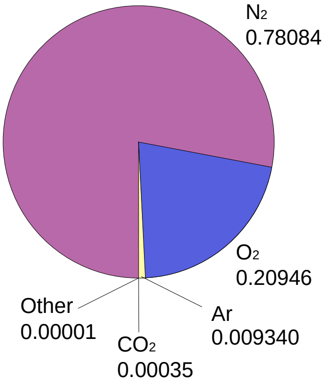
A probability distribution may be compared to a probability as a vector is to a scalar. It is a convenient notation for representing many of them all at once.

Perhaps, the two most important distributions in machine learning applications are the *categorical distribution* and the *normal distribution*. These two distributions are frequently used to describe beliefs pertaining to categorical and continuous values. There are many other important distributions with which you will probably want to eventually become familiar, but understanding these two will give you a good foundation for the basic concepts underlying distributions.

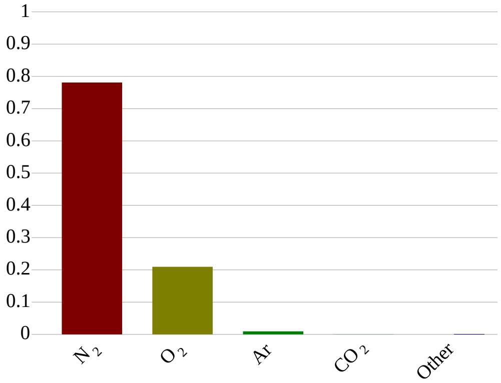
3.2.1 The categorical distribution

The categorical distribution describes the portion of probability that is assigned to each of *k* categories. (Sometimes, the categorical distribution is erroneously called the multinomial distribution, but this is an error. The multinomial distribution is a similar distribution, but it has some significant differences.)

Suppose you draw a random molecule from the air. Will it be a molecule of nitrogen, oxygen, argon, carbon dioxide, or something else? This pie chart describes the probability that the molecule will have each of these possible chemical types.



And, here is the same distribution expressed as a bar chart.



The individual probabilities in a categorical distribution must sum to 1, and like all probabilities, none of them may be less than 0, or greater than 1. You don't have to draw a chart to express a categorical distribution. For

example, we could express the same distribution with a vector of *k* values:
[0.78084, 0.20946, 0.009340, 0.00035, 0.00001].

The *mode* of a categorical distribution is the value with the highest probability. In this case, the mode is “N_2”. The *support* of a distribution is the set of values for which it expresses a probability. A categorical distribution supports *k* values.

3.2.2 The normal distribution

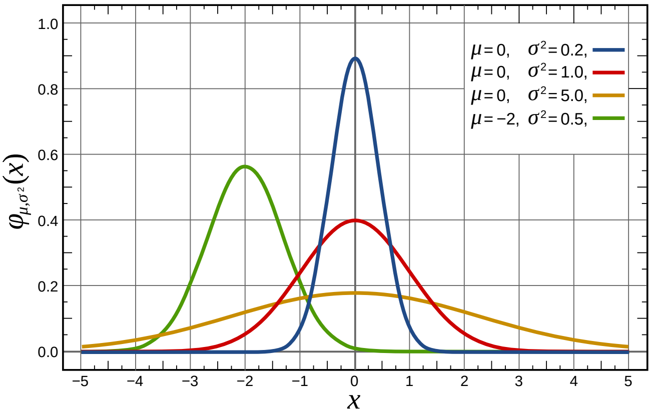
The normal distribution, also called the Gaussian distribution, or the bell curve, supports a continuous range of values, from $-\infty$ to ∞. With continuous distributions, it doesn't really make sense to refer to the probability of a particular value. Since there are infinitely many supported values, the probability of any particular value in a continuous distribution is usually zero. What good is having a distribution at all if, no matter what value we give it, it always tells us that the probability is zero?

So, instead of computing probabilities, continuous distributions compute *probability density*, or *likelihood*, for each supported value.

Probability densities do not sum to 1. (They sum to ∞.) But, they integrate to 1. Integrating to 1 is not the same thing as summing to 1. Integration essentially sums all the values multiplied by an infinitesimal. So, probability density is proportional to probability by a factor of ∞.

Intuitively, you can think of probability density as a number that is only useful for comparisons. Let us use “ϕ” to represent probability density. (This is the Greek letter “phi”. It rhymes with pie, but starts with an “f” sound. Phi is the Greek equivalent of the letter “p”.) If $\phi(a) = 2.7\phi(b)$, then values in the immediate vicinity of a are 2.7 times as likely to be drawn as values in the vicinity of b, even though the probably of drawing the exact value, a, is zero, and the probability of drawing the exact value, b, is also zero. $P(a)$ is just a bigger zero than $P(b)$. Working with zeros of differing sizes is rather bothersome, so we prefer to work with $\phi(a)$ and $\phi(b)$ instead.

This plot shows the probability density function of the normal distribution with various parameters.



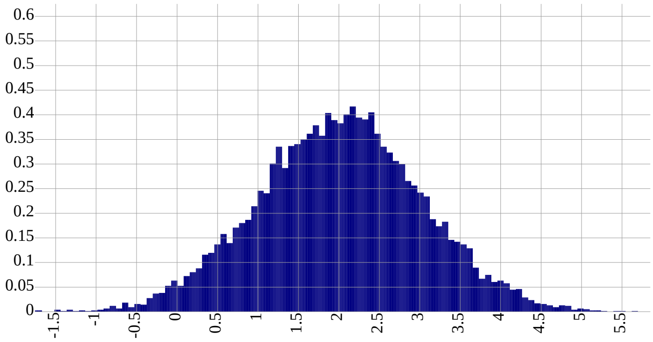
The normal distribution has two parameters. μ is the mean of the distribution. σ is the standard deviation. (σ^2 refers to variance, which is the square of standard deviation. Many implementations parameterize it with σ^2 instead.) When $\mu = 0$ and $\sigma = 1$, it is called the *standard normal distribution*. The equation for the normal distribution is,

$$\phi(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

With the normal distribution, the mean and the mode are always the same value.

3.2.3 Maximum likelihood estimators

Suppose you have some data. You plot a histogram of the data, and it looks something like this:



This data looks a lot like a normal distribution. It looks like it has a mean somewhere close to 2, and a deviation close to 1. But, are those precisely the best parameter values for the distribution? How do you exactly compute the best parameters?

The maximum likelihood parameters are those that maximize the likelihood that data like yours might be drawn from the distribution. We can use the probability density function (PDF) of the distribution to calculate the likelihood of each point in your data. For example, suppose your dataset contains only one attribute. Suppose one of the values is 2.1. Since we have already settled on the normal distribution, we can calculate the likelihood of this point by plugging the value, 2.1, as x into the PDF,

$$\phi(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

Since the points in a dataset are assumed to be independent, we can calculate the likelihood of the whole dataset by multiplying together the likelihood values of each point,

$$\phi(\mathbf{D}) = \prod_i \phi(\mathbf{d}_i).$$

(The “Π” in this equation means “product”. It does not mean 3.14159... The $\mathbf{D}$ represents your dataset, and $\mathbf{d}_i$ is the i^{th} point in your dataset.)

So, the maximum likelihood value for μ is the one that maximizes

$$\prod_i \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(d_i-\mu)^2}{2\sigma^2}}.$$

Since we are maximizing over μ, and the factor at the front of the equation contains no μ, we can drop it without affecting the results. So, now we want to maximize

$$\prod_i e^{-\frac{(d_i-\mu)^2}{2\sigma^2}}.$$

Since logarithm is a continuously increasing function, this is the same as maximizing

$$\sum_i -\frac{(d_i-\mu)^2}{2\sigma^2},$$

which is the same as minimizing

$$\sum_i \frac{(d_i - \mu)^2}{2\sigma^2}.$$

Since we are minimizing over μ, and the denominator contains no μ, this is the same as minimizing

$$\sum_i (d_i - \mu)^2.$$

To minimize this, we take the derivative, and set it equal to zero.

$$\frac{\partial}{\partial\mu} \sum_i (d_i - \mu)^2 = 0.$$

Next, we apply the chain rule.

$$\sum_i 2(d_i - \mu)\frac{\partial}{\partial\mu}(d_i - \mu) = 0.$$

Now we can resolve the ∂ term.

$$\sum_i 2(d_i - \mu)(-1) = 0.$$

We divide both sides by -2.

$$\sum_i (d_i - \mu) = 0.$$

Add $n\mu$ to both sides. (n is the number of data points over which we are summing.)

$$\sum_i d_i = n\mu$$

Finally, we can solve for μ.

$$\mu = \frac{1}{n}\sum_i d_i,$$

In other words, the maximum likelihood value to use for the μ parameter of a normal distribution is the mean of the data.

At this point, you might say, W*hat?! We did all that math just to find out this completely obvious result? The mean of the data? That is the first thing I would have guessed anyway!*

Well, yes, it is rather obvious, but now you are not guessing, and you know why we do it—it is the parameter value that maximizes the likelihood of your data. Even if the result is not that interesting this time, the technique is really quite significant. You can use the same technique to find the maximum likelihood parameters of any distribution.

With other distributions, it may not always work out so nicely. Fortunately, people have already worked them out for most of the well-known distributions, so you can just look them up on Wikipedia. For example, the unbiased maximum likelihood estimator for σ with the normal distribution is

$$\sigma = \sqrt{\frac{1}{n-1}\sum_i (d_i - \mu)^2}.$$

The maximum likelihood estimator for the values in the categorical distribution is quite obvious too. It is

$$P(v) = \frac{1}{n}\sum_i \begin{cases} 1 & \text{if } d_i = v \\ 0 & \text{if } d_i \neq v \end{cases}$$

In other words, you use the same ratio for each value as appears in the data.

3.2.4 Drawing random samples

Most programming languages provide a function that will draw a random number from a uniform distribution between 0 and 1. In this section, we will call this function “random()”.

3.2.4.1 Categorical

Suppose you want to draw a random value from a categorical distribution. Let $P(v)$ be the probability of the value, v. The following algorithm will choose a value, c, according to these probabilities.

```
function draw_from_categorical_distribution()
   $t$ = 0
   for each possible value, $v$
      $t$ = $t$ + $P(v)$
      if random() < $P(v)$ / $t$
         $c$ = $v$
   return $c$
```

This function is very short, but it may take some thinking to fully understand it. It iterates over each supported value, v. The variable, t, is a running total of the probability considered so far. When it returns, t should always be 1. Each time a new value, v, is considered, it decides whether or not to make this be the new chosen value, c. The first value visited will always be chosen, because it represents all of the probability visited so far. If there are more values, however, it might be changed to one of them. When it is done, every value has been given exactly the portion of probability that it should have been given, $P(v)$.

If you need to draw n random variables from a categorical distribution that supports k possible values, this will have a computational cost in $O(nk)$. If you are really worried about computational complexity, you can do it even more efficiently by drawing from a binomial distribution to determine how many of each value are chosen, and then shuffling the values. Unfortunately, an approximation technique must be used to draw from the binomial distribution efficiently, so this approach is a bit more complex than is necessary for most applications.

3.2.4.2 Normal

A simple way to draw from a standard normal distribution uses rejection sampling to draw from a unit circle, and then uses the Box-Müller transform to map to a value from a standard normal distribution. Here is the pseudo-code:

```
function draw_from_normal()
   loop
      x = random() * 2 – 1
      y = random() * 2 – 1
      m = x * x + y * y
      if m < 1 and m > 0
```

```
        return y * sqrt(-2.0 * $\log_e$(m) / m)
```

Usually, this algorithm will return on the first pass through the loop. Sometimes it might take two or possibly more iterations before it finds a value inside the unit circle, but this will not happen often, so it tends to be generally efficient. It falls within the circle $\pi/4$ (nearly 4/5) of the time. The odds of missing 3 times in a row are less than 1%.

To draw a value from some normal distribution other than the standard normal distribution, first draw a value from the standard normal distribution, then multiply the value by the deviation you want, and add the mean.

Todo: there are many other distributions that you can draw from using the normal distribution as a base. Perhaps I should tell how to do it.

Todo: talk about how to invert the CDF of an arbitrary distribution to develop an algorithm to draw a random value.

3.3 Utility theory

3.3.1 Lottery example

Suppose there are two lotteries that you have the option of entering: It costs \$1 to buy a ticket in lottery A. The prize is \$5000, and the odds of winning are 1 in 10000. It costs \$20 to buy a ticket in lotter B. The prize is \$1M, and the odds of winning are 1 in 50000. Let's also say that you live in a place where lottery winnings are only taxed at a rate of 10%. Which lottery is a better deal? (Is your intuition good enough to answer this question without doing any math? It might be fun to try it, and see how well you do.)

To answer this question, we compute the *expected utility* of entering each lottery. Expected utility is the average of all possible utilities, weighted by the probability that it will occur. The process of computing expected utility is sometimes called “integrating out the uncertainty”.

The expected utility of entering lottery A is,

$$\frac{1}{10000}(\$-1+(\$5000-10\%))+$$
$$\frac{9999}{10000}(\$-1)=\$-0.55.$$

The expected utility of entering lottery B is,

$$\frac{1}{50000}(\$-20+(\$1M-10\%))+$$
$$\frac{49999}{50000}(\$-20)=\$-2$$

It turns out that the answer to the question depends on how you evaluate cost. Entering lottery A exactly one time is a better deal than entering lottery B one time. Entering lottery A is like throwing away 55 cents, whereas entering lottery B is like throwing away 2 dollars. So, if you are determined to enter just one lottery, then lottery A would be the better choice. However, entering lottery B is a better deal per dollar spent. For the same cost, you could enter lottery A 20 times, which would be like throwing away $\$11$. So, if you are determined to spend some fixed amount of money on lottery tickets, lottery B would be a much better choice.

Of course, the choice with the highest expected utility is to refuse to play the game. This has an expected utility of $\$0$. If you look closely at these two lotteries, you might also notice that they have some interesting differences in who they benefit. In lottery A, the lottery operators make ten times more profit ($0.50) than the state ($0.05). In lottery B, however, the state is the only sure winner (assuming the lottery is actually fair).

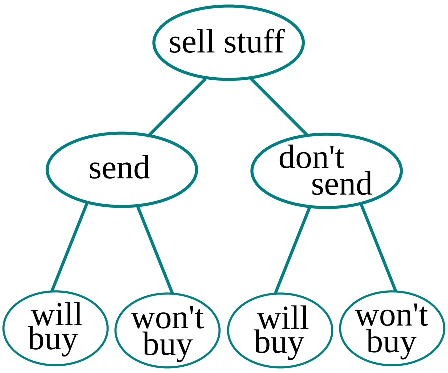
Note that when you enter a lottery, there are two possible outcomes: win and lose. Because the probabilities in a categorical distribution must sum to 1, it is typical to express the distribution with a single probability. You are expected to know that you can compute the probability of the other possibility by subtracting from 1. If you think of it as a categorical distribution with two categories, however, then it becomes more intuitive that the same technique can be applied to compute expected utilities with other types of distributions. Specifically, you compute expected utility as a weighted average over all of the values supported by the probability distribution.

3.3.2 Utility trees

Now, let's see if we can reduce all these principles of utility theory into a single general approach for making decisions. To assist us, here is an example problem to work with:

Suppose a marketing company hires you to help them decide which customers to mail coupons to. By combining data that you scraped from social networks with the company's historical sales data, you obtain enough personal details to train a learning algorithm, *m*, to predict the likelihood that each customer will buy your product without a coupon. Let *m*(*c*) be the predicted probability that customer *c* will buy your product if you send no coupon to customer *c*. Suppose your analysis indicates that receiving a coupon approximately doubles the probability that a customer will buy your product (with a maximum probability of 1, of course), independent of all other conditions (...if only the real world were so simple). Suppose the company makes $3 for each product that they sell, but it costs $0.48 to mail the coupon, and the coupon reduces the profit by 50 cents if the customer buys it. Under what conditions should they send a coupon to customer *c*?

Let's start by building a tree-diagram of the possibilities:



The root node branches on a decision that we get to make: either send a coupon, or don't send a coupon. The two child nodes branch on something we are uncertain about (because it is a decision the customer gets to make). For a more complicated problem, we would continue to build out the tree (with more branches that represent decisions or uncertainty) until we arrived at very simple situations for which we could identify the utility. These are the leaf nodes.

In this case, the utilities of the leaf nodes from left to right are:

$-0.48 + $3 - $0.50,
$-0.48,
$3,
$0.

Next, we systematically collapse the tree until we arrive at the root node. Branches that represent uncertainty are collapsed by integrating out the uncertainty. Branches that represent decisions are collapsed by choosing the option with the highest expected utility. Those are pretty-much the only two operations in utility theory. Since this tree is so small, it will not take very much work to collapse it, but we could in theory handle arbitrarily complex problems in the same manner.

So, the expected utility of not sending a coupon collapses to,

$$m(c) * \$3,$$

and the expected utility of sending a coupon is,

$$\min(2 * m(c), 1) * \$2.50 - 0.48$$

And the answer to the question, *under what conditions should they send a coupon?,* is whenever

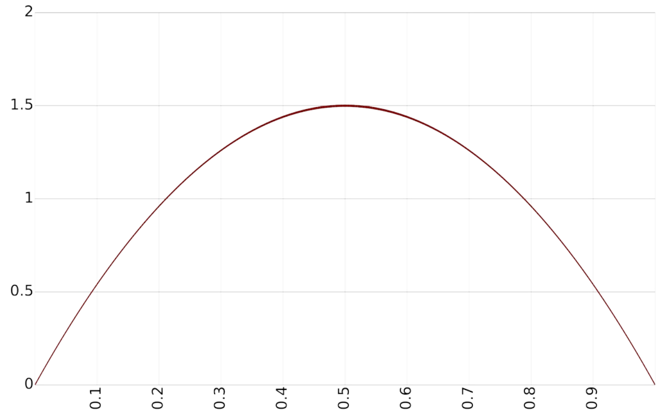
$$\min(2 * m(c), 1) * \$2.50 - 0.48 > m(c) * \$3.$$

3.3.3 Continuous uncertainty

Suppose a 10 million dollar prize is offered to any organization that can biologically engineer a plant or animal to survive on Mars for 150 Earth days. You are hired by a company that is interested in entering the contest. They don't know for sure how long their plants will survive on Mars, but they estimate that the survival rate (in Earth years) of their strongest plant follows the distribution,

$$\phi(x) = 6x - 6x^2,$$

with continuous support from 0 to 1. (In other words, the probability density for any value where $x < 0$ or $x > 1$ is 0.) If we plot this PDF, it looks like this:



The mean falls at 0.5 Earth years (about 182 Earth days), which is good enough to win the prize. However, the company has already invested 5 million dollars to engineer the plant, and it would cost 6 million dollars to send the plant to Mars to even attempt to claim the prize. You can see right away that this company is not going to break even (since $5M + $6M > $10M), but now your job is to decide whether it is worth continuing the effort.

There are two possible actions: send it, and don't send it. The best action is the one with the highest expected utility. So, we should calculate the expected utility of both choices. The expected utility of not sending it is $0. (There is no need to consider the amount the company has already invested, because that will affect the utility of both choices equally. Utility theory only looks forward.) So, the expected utility of sending it is,

$$-\$6\text{M} + \int_0^1 r(6x - 6x^2),$$

where

$$r(x) = \begin{cases} \$10\text{M} & \text{if } x \geq 150/365 \\ \$0 & \text{if } x < 150/365 \end{cases}$$

which evaluates to

$$\begin{aligned} &= \$(-6 + (10 - 30\left(\frac{150}{365}\right)^2 + 20\left(\frac{150}{365}\right)^3))\text{M} \\ &\approx \$0.321\text{M}. \end{aligned}$$

Since sending has a higher expected utility, that is the best choice. It may not be enough to recover their $5M investment, but it is still better than doing nothing.

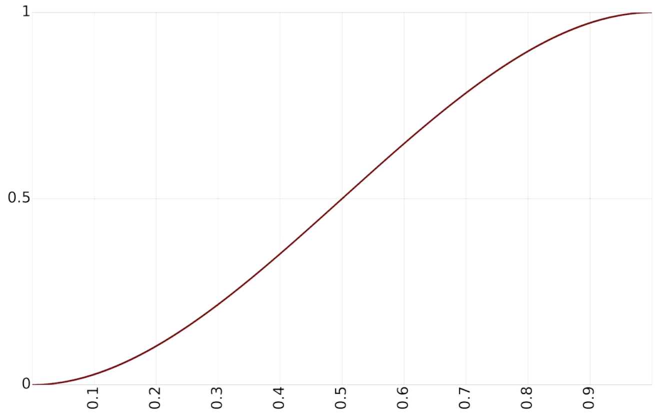
Now, suppose it is possible to test whether the plant will survive on Mars by simulating Mars-like conditions in a tank on Earth. Suppose this test is perfectly reliable—if the plant survives this test, it will survive on Mars (but you would still have to send it to Mars to claim the prize), and if the plant dies in the test, it would die on Mars (so you would never send it to Mars). It costs 2.5 million dollars to set up this tank and perform the test. Should you perform the test, or should you just send the plant directly to Mars without testing it?

We already know the expected utility of just sending it is $\approx \$0.321\mathrm{M}$, so all we need to compute is the expected utility of performing the test. But how do we compute the expected utility of a test, since we don't know whether it will work out or not? We integrate out the uncertainty, of course. In utility theory, that's what you do whenever there is uncertainty. Once you have gotten rid of the uncertainty, the best decision is easy to compute.

We can integrate our probability density function to obtain a cumulative density function, which we represent with an uppercase phi,

$$\Phi(x) = 3x^2 - 2x^3.$$

Here's a plot of this function:



If we set this function equal to 150/365, and solve for x, we obtain,

$$x \approx 0.368.$$

This is the probability that the plant will die on Mars before day 150. So, the probability that it will survive past day 150 is,

$$1 - 0.368 = 0.632.$$

Since the test is perfectly reliable, this is also the probability that we believe it will survive when we perform the test. After we perform the test, of course, we will know for sure whether it passed or not, but we must use the prior probability when deciding whether to perform the test, because that is the best information we have at that time.

So, the expected utility of performing the test is 0.632 times the expected utility of having a surviving plant plus 0.321 times the expected utility of having a plant that dies. More specifically, it is

$$0.632(-2.5 - 6 + 10) + 0.321(-2.5)$$
$$= \$0.146\mathrm{M}.$$

The utility of performing the test is actually lower, so it is better to just sent the plant to Mars untested, and see what happens.

To put it another way, the value of the test is the amount that it improves our expected utility. In this case, the value of the test is

$$(0.632(-6 + 10) + 0.368(0)) - 0.321 = \$2.21\mathrm{M}.$$

Clearly, you should not pay $2.5M for a test that is only worth $2.21M.

At first, it might seem counter-intuitive that to estimate the value of a test, you need to predict how it will turn out, since the very purpose of a test involves determining out how something turns out. But, if you knew exactly how the test would turn out, then performing the test would obviously have no value. Therefore, your uncertainty must play a role in the value of the test, and we express uncertainty in the form of a distribution. Interestingly, tests will have more value when there is more uncertainty about how they will turn out. Note that your uncertainty will not only affect the utility after the test is performed, but also the utility that you have before the test, and the value of the test is the difference between them.

In this example, we showed how to deal with continuous uncertainty (by integrating out the continuous uncertainty). You can also use utility theory to make continuous decisions (by finding the value that maximizes the expected utility). It is really about the same process as with categorical decisions or uncertainty.

3.3.4 Inferring utility values

Suppose you train a machine learning algorithm to determine whether or not individual mushrooms are poisonous. (The “mushroom” dataset is a well-known dataset for testing machine learning algorithms. You could actually try this if you want to.) Then, you go out into the wild and find a mushroom. After describing the various attributes (cap shape, color, odor, gill spacing, stalk shape, etc.) of the mushroom to your classifier, it predicts that the mushroom is edible (as opposed to poisonous). So, what should you do? Eat it, right?

Hold on a moment, let's consider the consequences, first. If your classifier is right, you get to enjoy a tasty mushroom (assuming you like mushrooms). If it is wrong, you might consume a death cap, which might destroy your liver, and you could pay for the mistake with your life. Perhaps you ought to evaluate the accuracy of your classifier before you proceed.

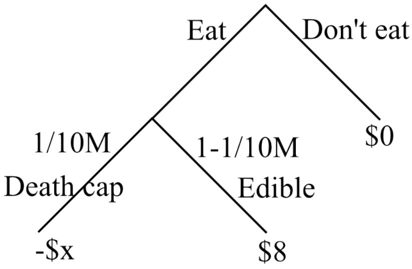
Here is a picture of some death cap mushrooms:

Let's say your classifier is accurate at identifying poisonous mushrooms in 97.6% of cases with a hold-out set. That's pretty good, right? Well, for some applications, that would be amazingly good. For this one, it is probably not be good enough. So, what level of demonstrated reliability would be necessary to persuade you to eat the mushroom?

Suppose that you survey a whole bunch of mushroom connoisseurs. You ask them how much they would pay for a good mushroom. On average, let's suppose they would pay $8. If given a free mushroom, suppose they would eat it if there was less than a 1 in 10 million chance that it might actually be a death cap. (This would require a predictive accuracy greater than 0.9999999.) Now, we might ask, how much do these mushroom connoisseurs value their lives?

In order to calculate this, we will first make some assumptions: We will assume that our mushroom connoisseurs are rational about the value of their lives and about mushrooms. When it comes to topics of obsession, people are often not very rational. Furthermore, matters of life and death have a way of making people behave irrationally. So, in real life, responses to questions like these often do not accurately represent how much people value their own lives, but just for fun, let's proceed with the math as if those numbers were reliable.

Here's the utility tree:



Now, let's calculate the value of their lives:

$$\frac{1}{10,000,000}(-\$x) + \frac{9,999,999}{10,000,000}(\$8) = \$0,$$

$$x = \$79,999,992.$$

3.3.5 Superfluous details

An important skill in evaluating utility problems is seeing past the superfluous details. Let's do another example:

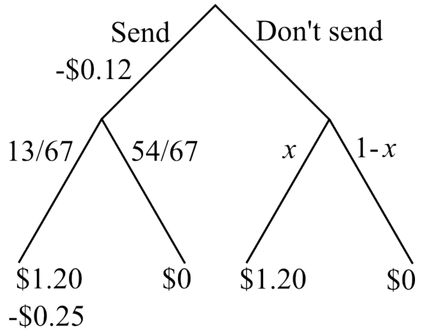
A company named Walmarket already bought 9 million mechanical pencils for $0.15 each. Now, Walmaket intends to sell these pencils for $1.20 each. Walmarket analysts have identified 7 million potential customers, an estimated probability that each customer will buy one of these mechanical pencils, and their home addresses. They also estimate that about 13 out of 67 customers who receive a coupon offering $0.25 off the price of a mechanical pencil will buy one, no matter which customers receive the coupons. With bulk mail pricing, Walmarket can mail a coupon for $0.12. They want to send a coupon to every customer who has a probability less than *x* of buying a pencil. The Walmarket analysts were very thorough and successfully identified all potential customers. No customer will buy more than one mechanical pencil. There is no sales tax in the state where this occurs. Shipping and distribution incur no significant costs. What value for *x* would maximize Walmarket's profit?

Can you solve this problem? Try solving it before you read the solution below.

First, let's comment about the superfluous details in this problem:

- The detail “already bought 9 million mechanical pencils for $0.15 each” is completely superfluous. From a utility perspective, what you did in the past has no bearing on what you should do in the future. No matter what decision you make now, Walmarket still already paid that amount, and it will affect all possible future outcomes equally. In other words, there is no reason to even consider this detail in the solution. (You can if you want to, but it won't change the answer at all.)
- Note that Walmarket bought 9 million pencils, but they are not going to sell them all. Is it possible for Walmarket to even make a profit? If you are asking this question, you have missed the point. Your job is to compute *x*, not to worry about whether the company is making profit.
- When faced with this problem, many people try to calculate Walmarket's expected profit. Don't do that. This problem does not even provide enough information for you to estimate profit. To do that, you would need to know the distribution of probabilities. Apparently, the analysts have that information, but it was not given to you.

Here is the utility tree for this problem:



Note that Walmarket must pay $0.12 to send the coupon whether or not the customer decides to use it. However, the coupon only reduces the profit by $0.25 if the customer uses the coupon.

Now, let's calculate *x*. Walmarket should send a coupon if:

$$\frac{13}{67}(\$1.20 - \$0.25) - \$0.12 > \$1.20x,$$

$$x > 0.053606965.$$

3.3.6 Emeralds example

Shall we do another one? Suppose you want to dig for emeralds in Venezuela. You have already purchased a round-trip plane ticket to Venezuela for $900. If you decide to proceed with your plans, you will still need to purchase digging equipment for $600. You estimate that there is a 0.1 chance that you will find $100,000 worth in emeralds, and an 0.9 chance that you will find only worthless dirt. If you try to bring emeralds back on your return flight, you estimate that there is a 0.95 chance that the customs officer will discover them hidden in your luggage and confiscate them, leaving you with nothing. However, if you pay the customs officer a bribe of $200 before he inspects your luggage, the chances that he will discover your hidden bounty change to *x*. For what values of *x* would it be better to just throw your plane ticket in the garbage and forget about the emeralds?

The answer is *x* > 0.?38. (I replaced one of the digits in the answer with a '?' to avoid completely spoiling the challenge.)

3.3.7 Predicting distributions

If you are using a *k*-NN model, you could predict a distribution by simply measuring the distribution among the *k*-nearest neighbors, instead of merely finding the most-common class. Because this uses a small subset of the data to estimate a distribution, it is important to use Laplacian smoothing so that you don't report 100% confidence in the cases where *k* neighbors just coincidentally agree. In general, if you plan to rely on these distributions for decision-making, it is important to use a value for *k* that is quite large, perhaps much larger than the value that yields the best predictive accuracy. With small values of *k*, this approach produces very unreliable distributions.

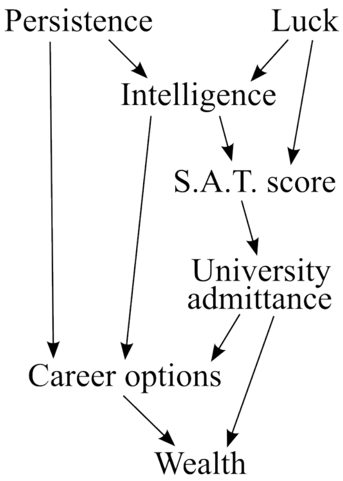
Likewise, ensembles provide a fairly natural approach for predicting distributions. Instead of combining the predictions of all the models into a single prediction, you can measure the distribution among all the predictions. Warning: be careful with this approach. Just having a distribution is not the same thing as having an accurate distribution. In my experience, although ensembles often make accurate classifications, the distributions within an ensemble of predictions are very often poor representations of the actual distributions they are intended to represent, particularly when it is an ensemble of fully-filled-out trees. (For some intuition on this matter, please carefully examine the visualizations in the section about forests.)

A technique called *calibration* has been used to attempt to correct for inaccuracies in predicted distributions. The gist of calibration is that you train a learning algorithm to accept a predicted distribution as input, and predict a corrected distribution as its output. Although calibration certainly can improve upon predicted distributions, it is generally considered to be somewhat of a hack.

Perhaps a better approach is to use an inference technique that is designed from the beginning to predict an accurate distribution. An excellent example of such a technique is Bayesian inference.

3.4 Belief networks

A *probabilistic graphical model* (PGM) is a graph that represents the probabilistic dependencies among a set of variables or attributes. Here is an example:



The edges in this graph show the direct causal relationships among the attributes. For example, those who are persistent tend to develop greater intelligence, so I put an arrow between “Persistence” and “Intelligence”. Of course, real life is much more complicated than this, but even this simple graph expresses many complicated causal relationships. For example, notice the arrow between “University admittance” and “Wealth”. Since prestigious universities tend to have relatively high tuition, being admitted to such a university will have a direct negative effect on one's wealth. Hopefully, attending a prestigious university also has a positive effect on wealth, but this positive effect is indirect (passing through “Career options”). “S.A.T. score” only has indirect influence on one's wealth (assuming any fee required to take the test is negligible), so there is no arrow directly connecting it to “Wealth”.

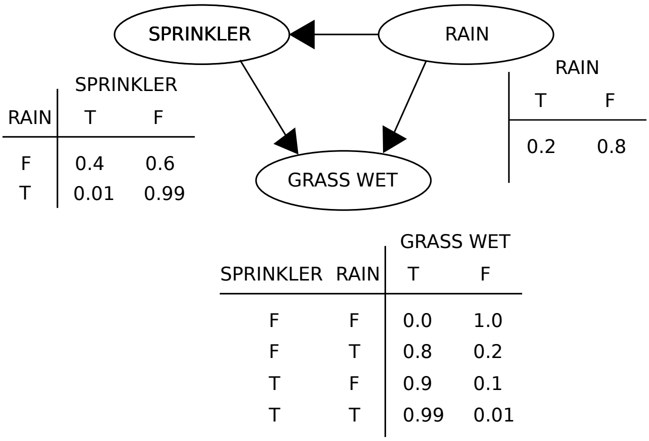
Note that this graph is directed and acyclic. A directed and acyclic PGM is called a *Bayesian network* or a *belief network*. The primary reason for using a belief network is to perform inference. For example, suppose we know the wealth of a certain person, and we also know what that person's S.A.T. score was, can we then infer what university that person probably attended, and how intelligent he or she is?

Since there are other factors, besides intelligence, that probably influenced that person's university attendance and wealth, such as luck, the answer will not be a single university name and intelligence value. Rather, it will be a probability distribution over possible universities and intelligence values. These distributions will depend on the observed values for “Wealth” and “S.A.T. score”, as well as the probability distributions that define all of the causal connections in the belief network. Sounds complicated, doesn't it? Well, if we were going to do all the math necessary to perform exact ineference, it would be insanely complicated. Fortunately, a relatively simple technique called Markov Chain Monte Carlo exists, which enables us to sample from an arbitrary belief network in order to estimate the *conditional joint distribution* of the model.

Joint means “together”, so a joint distribution is the distribution of values for all of the nodes in the entire graph together. A conditional joint distribution is the distribution of all the nodes conditioned on any observed values. In other words, the conditional joint distribution tells us everything we could possibly want to know about the graph, given whatever we already know about it.

Of course, there is always a catch. In this case, the catch is that the belief network must be fully specified. A fully specified belief network involves a lot more than just a bunch of arrows connecting a set of attributes. It requires that you specify a complete probability distribution for every attribute, conditioned on all of its parent nodes (that is, all of the values with incoming arrows). Where does all this information come from? Well, hopefully, most of it is derived from observed data. This is typically done using maximum-likelihood estimators. In practice, however, it often involves a lot of human intuition and a little bit of fudge-factor. In other words, we just make up reasonable numbers for any values we need but don't really know for sure.

Thus, just like most data-driven machine learning techniques, Bayesian inference involves a lot of inexact estimation. The difference, however, is that data-driven machine learning techniques typically perform the inexact estimation within the heuristic learning algorithms. In Bayesian inference, all estimations are done explicitly. So, you can always state precisely how much fudging was done in order to obtain an answer. Another advantage of Bayesian inference methods is that they typically give answers in the form of a probability distribution, rather than just a prediction with a confidence value. These distributions can be very useful for computing utility values. Here is an example of a fully specified belief network:



	SPRINKLER	
RAIN	T	F
F	0.4	0.6
T	0.01	0.99

RAIN	
T	F
0.2	0.8

		GRASS WET	
SPRINKLER	RAIN	T	F
F	F	0.0	1.0
F	T	0.8	0.2
T	F	0.9	0.1
T	T	0.99	0.01

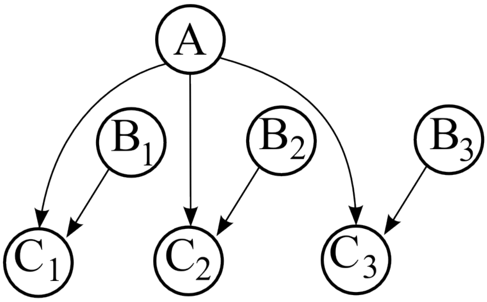
This graph contains three nodes. All three nodes in this graph represent a categorical distribution with two values, {T, F}. (Other distributions may be used in a belief network as well.) The tables next to each node specify the parameter values for the distribution. The table for “Rain” gives values for a single categorical distribution. The table for “Sprinkler” gives parameters for two categorical distributions. One of them is for the case where “Rain” is true, and the other is for the case where “Rain” is false.

Each node in a belief network needs to specify parameters for every permutation of values that its parent nodes might have. In the case of “Grass Wet”, parameters for four categorical distributions are given, because there are four permutations of values that might affect the value of this node.

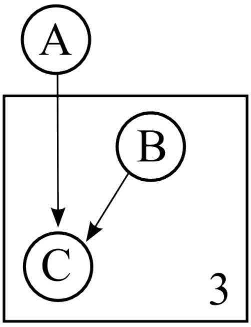
This property is not transitive. That is, each node is only conditioned on its immediate parents. (If it were conditioned on any other node, then there should be an arrow pointing from that node to this one, making it a parent of this one.)

3.4.1 Plate notation

Many real-world belief networks have a lot of nodes in them. It can be bothersome to draw them all. Plate notation is an easy way to indicate that you don't want to bother drawing all of the nodes. A plate is just a rectangle. A number in the corner of the plate may be used to indicate the number of times that its contents are replicated. For example, this network,

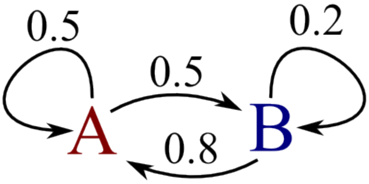


could be more succinctly represented as,



3.4.2 Markov Chain Monte Carlo

Consider a simple machine with one two states: A, and B. When this machine is in state A, there is a probability of 0.5 that it will transition to state B. When it is in state B, there is a probability of 0.8 that it will transition to state A.



This is called a *Markov chain*. A Markov chain is a system that transitions through a sequence of states. At each time, the next state depends only on the current state. (Don't confuse this graph with a PGM. It may look like a PGM, because we are showing it as a graph, but the nodes in this graph represent states, not attributes, and the arrows in this graph represent state transitions, not causal influence. A Markov chain is altogether a different thing from a PGM. It just happens that one is used to do inference with the other, and the other can be designed to model the one—but don't let that confuse you right now—they are different things.)

Suppose this machine begins in state A. We might ask the question, *after n iterations, what is the probability that the machine is in state A?* Intuitively, since this machine stays in state A with higher probability than state B, you might predict that it is a little-bit more likely to be in state A at any moment. That is true, but what if you want to know precisely how much more likely it is?

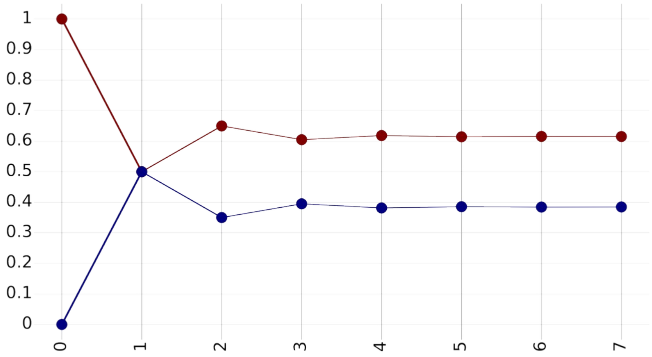
In this case, we can calculate the exact probability of what state it will be in at later time steps:

$$P(A)_{t+1} = P(A)_t * 0.5 + P(B)_t * 0.8$$
$$P(B)_{t+1} = P(A)_t * 0.5 + P(B)_t * 0.2$$

The subscripts indicate the time at which we are computing the probability. The following table shows these probabilities over time:

Time	P(A)	P(B)
0	1.000	0.000
1	0.500	0.500
2	0.650	0.350
3	0.605	0.395
4	0.619	0.381
5	0.614	0.386
6	0.616	0.384
7	0.615	0.385
∞	8/13	5/13

Over time, the probability of being in any particular state converges to a constant (or *stationary*) value. If we plot these values, we can see that these probabilities converge rather quickly.



The horizontal axis shows time. The red dots show the probability of being in state A at a given time. The blue dots show the probability of being in state B at a given time. Even though it theoretically takes an infinite number of iterations before these values arrive at the exact stationary probabilities, they get close enough for most practical applications after only a finite number of iterations.

It has been proven that any Markov chain will eventually converge to a stationary distribution as long as each state is reachable with a non-negligible probability. Some Markov chains might converge to this stationary distribution faster than others, but if you sample long enough, you should eventually get there.

In probabilistic graphical models (PGMs), the conditional joint distribution is often somewhere between difficult and impossible to compute exactly. However, if you happen to know the state of a PGM at a given time step, reasonably simple techniques are known that will draw a sample state for the PGM in the next time step. (In fact, these will be our next topic for study.) If we start with an arbitrary state, and then draw samples in a loop, these samples will eventually start to converge to represent a stationary distribution of the PGM. As it happens, the conditional joint distribution is almost always a stationary distribution.

This technique is called *Markov Chain Monte Carlo* (MCMC). When we draw samples from the PGM, this forms a Markov chain, because each sample depends only on the previous state. This Markov chain should eventually converge to represent the conditional joint distribution of the PGM. A *Monte Carlo* technique is one that involves sampling from a distribution in order to represent that distribution.

So, MCMC can be described in the following steps:

1. Pick an arbitrary initial state for a PGM.
2. Draw a sufficiently long chain of samples from the PGM. These are called the *burn-in* samples. These samples are discarded, because we assume they were drawn before the Markov chain had sufficiently converged to the stationary distribution.
3. Continue drawing more samples. Gather these samples to represent the conditional joint distribution of the PGM.

Because MCMC relies on samples to represent the distribution, it can be difficult to get a lot of precision. If you need more precision, you need to draw exponentially more samples. But, as long as you can be content with just a few digits of precision, it can be a powerful technique. It can handle PGMs of tremendous complexity, that would be much too complex to solve with any exact inference method.

Unlike the state machine with only two states that we used as an example in this section, the Markov chain used by MCMC typically has a lot of states. Specifically, every unique set of values that the PGM could possibly represent is one of its states. That is a pretty-big state machine! If one or more of the nodes follows a continuous distribution, then there is an infinite set of possible states. Nevertheless, it still seems to work in practice, so it is a useful technique.

3.4.2.1 Implementation details

To implement MCMC, every node in the network needs to know its "current value". This is typically done by adding a member variable to the "Node" class. Also, we need to be able to compute the conditional probability that some value might occur in a node given the current values of its parents. This may be implemented by adding a method to the "Node" class, like this:

```
double conditional_probability(double value) {
    Query the current value of each parent node.
    Use the PDF of the distribution of this node
        to compute the probability of value
        given the current parent values.
    Return the probability.
}
```

There are two common types of parent nodes: categorical parent nodes and continuous parent nodes.

Categorical parent nodes create cases for the distribution of the child node. A node that has no categorical parent nodes only needs to implement one distribution. If a node has one categorical parent node that supports *t* possible values, then this node needs to store the parameters for *t* distributions. Thus, the parameters for any node can be implemented in an *m*-by-*k* table, where *k* is the number of parameter values for the distribution, and *m* is the number of cases, or permutations of the values of the parent nodes. The values in this table are often constant, but it is also possible for these values the be variable, given by another node in the graph.

Continuous parent nodes are used to supply variable parameter values for the PDF of the distribution of the child node. In other words, they supply values for elements in the *m*-by-*k* table.

Since each node requires a table that can contain either constant values or variable values, implementation can be tricky. A good solution is to add a special node type that represents a constant value. Such a node is easy to implement. Its “current value” is the constant, and it never changes. Then, the table can be implemented as a table of node references (or pointers). This way, the user is free to mix constant and variable values in the table as desired, and the designer does not need special code for each case.

To summarize, categorical parent nodes increase the size of the table that a node must use to store the parameter values for its distribution. Continuous and constant parent nodes supply the values that fill in this table.

Example:

> Consider a graph where Node A represents a normal distribution. Suppose Node A has two parent nodes, Node B, and Node C. Suppose Node B is a categorical node with 2 values, and Node C is a categorical node with 3 values.
>
> In this example, Node A needs to store an internal table of size 6× 2, to represent 6 possible cases of a Normal distribution. (Each Normal distribution requires 2 parameters, mean and variance.) Perhaps, the mean parameters are all constant. In that case, 6 constant nodes are allocated to fill these slots. Perhaps, the variance parameters are variable. In that case 6 nodes that represent some continuous distribution may be allocated to fill those slots. (Nodes with an Inverse-Gamma distribution are often used to represent variance when it is variable.) Thus, it may be said that Node A actually has 14 parent nodes, although constant nodes are not usually counted.
>
> Suppose the “conditional_probability” method is called, and the value “0.1” is passed in. The task for this method is to compute the conditional probability that this value (0.1) might occur at this node (Node A) given the current conditions (the current values of Nodes B and C). So, it first queries Nodes B and C for their current values, to determine which of the 6 possible cases to use. Then, it looks in the table to obtain the mean and variance values. Since the mean is constant, it will always have the same value. The variance will be the “current value” of whatever node fills that slot in the table. After the mean and variance values are obtained, it plugs the value 0.1 into the probability density function of a Normal distribution to compute the probability. This probability is then returned.

The following pseudo-code can be used to calculate the number of rows in the table that a node uses to store its parameter values:

```
m=1
for each categorical parent node, b
        m = m * the total number of
                values supported by node b
return m
```

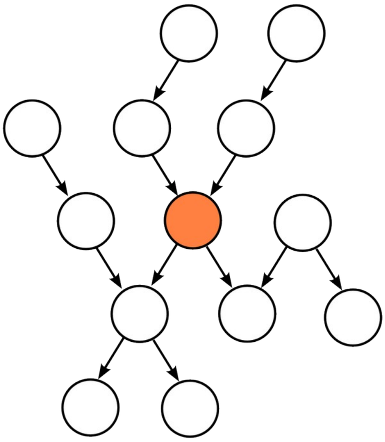
Similarly, the following pseudo-code might be helpful in implementing the “conditional_probability” method. It returns a unique value from 0 to *m*-1 to represent the case encoded in the current values of the parent nodes:

```
$v$=0
$m$=1
for each categorical parent node, $b$
    $v = v + m$ * the current value of node $b$
    $m = m$ * the total number of
              values supported by node $b$
return $v$
```

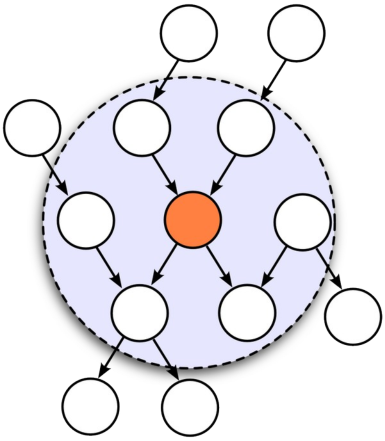
3.4.3 Gibbs sampling

To complete an implementation of MCMC, we need to describe a method for sampling the next state in a PGM. Perhaps, the easiest way to do this is with *Gibbs sampling*. Gibbs sampling involves visiting each node in a PGM, one at-a-time, and drawing a new sample for that node. When you have drawn a new sample for every node, those samples are a new sample for the entire network.

Suppose we want to draw a new sample for the orange node in this graph:



There are obviously many causal dependencies in this graph. If we didn't know the values of all these nodes, then computing a new sample for this orange node would involve a lot of complex math. Fortunately, we do have values for all of the nodes. These values might be somewhat arbitrary at first, but they should eventually converge to meaningful values that actually follow the conditional joint distribution of the PGM. It turns out that when the state of every node in the PGM is known, the next value for any node depends only on the current values of its parents, each of its children, and all of their parents.



This set of nodes is called a *Markov blanket*. Because the next value of the orange node is independent of all nodes outside of the Markov blanket (given the current values of all the nodes in the Markov blanket), we can compute a new sample for this node without having to do an unreasonable amount of computation.

In this example, it may look like the Markov blanket still covers a significant portion of the network, but imagine a network with thousands of nodes—having to consider only the neighbors really speeds things up a great deal!

Let n be the node we wish to sample. (In the diagram, this is the orange one.)
Let $V = \{v_1, v_2, \cdots\}$ be the set of values supported by n.
Let $B = \{b_1, b_2, \cdots\}$ be the parent nodes of n. (That is, an arrow points from each $b \in B$ to n.)
Let $C = \{c_1, c_2, \cdots\}$ be the child nodes of n.
(That is, an arrow points from n to each $c \in C$.)
Let D be the set of parent nodes of some child, c_j.
(So, n will always be one of the nodes in D.)
Then, to draw a new Gibbs sample for n, we just need to draw a value from the distribution where

$$P(v_i) = \frac{P(v_i|B)\prod_j P(c_j|D)}{\sum_k P(v_k|B)\prod_j P(c_j|D)}.$$

The Σ means "sum". The "Π" symbol means "product". (It does not mean 3.14159...) The probabilities in this equation are given by the fully specified belief network. That is, you take the given values (the part after the "|") and plug them into the distribution given with the belief network to obtain the probability of the value v_i.

Note that the denominator is approximately the same as the numerator, except that it sums over all possible values. In other words, it is just a normalizing factor to ensure that all of the probabilities sum to 1. It looks a lot nicer if we express it like this:

$$P(v_i) \propto P(v_i|B)\prod_j P(c_j|D).$$

The "$\propto$" symbol means "proportional to". It implies that the probability we are computing is not yet normalized. So, after you compute the probabilities for each v_i, just don't forget to normalize them all so they sum to 1.

When implementing this equation, it is important to behave as if the value of n was already set to v_i. (This occurs in the product, where n is one of the nodes in D.) If you use the previous value of n, instead of the value whose probability you are computing, you will get an incorrect probability.

Sometimes it helps to see another implementation, so here is some pseudo-code to implement Gibbs sampling with a categorical node.

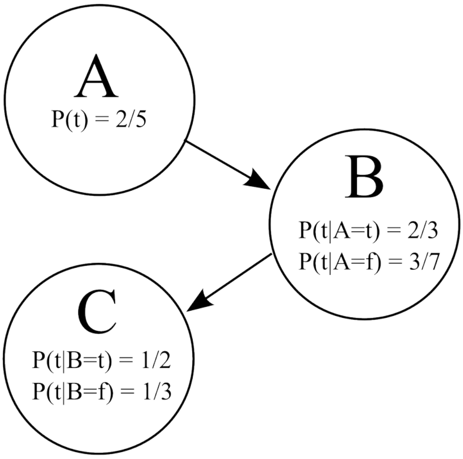
```
Let z be a vector of size $|V|$.
for i from 0 to $|V|$-1
    Temporarily set the value of $n$ to $v_i$.
    z[i] = n.conditional_probability($v_i$)
    for each $c \in C$
        Let w be the current value of node $c$.
        z[i] = z[i] * c.conditional_probability(w)
Normalize the values in z so they sum to 1.
Draw a new value for $n$ from the probabilities
given in z.
```

It should be clear that this code does the very same thing as the equations given before it.

So, to do Gibbs sampling with a belief network, you just repeatedly resample every non-observed node in the network. It is generally a good idea to visit the nodes in a random order in each pass through the network.

3.4.3.1 A simple example

When you implement something complex, it is a good idea to first test it with something very simple, so you can make sure that it works as expected. Here is a very simple fully-specified belief network. It has 3 nodes that support the binary values {t,f}:



Note that we did not specify the probability that any of these nodes will ever take the value “f”. This is because we already specified that they are binary nodes, so they only support two values, “t” and “f”. Since probabilities must sum to 1, you can easily figure out the probabilities for the other value by subtracting from 1.

For testing purposes, let us ask the following two questions of this network:

- What is the probability that B is true given that A and C are both true?
- What is the probability that B is false given that A and C are both

false?

This network is so simple that we can actually perform exact inference. That is, we can calculate the answers to these questions in closed form:

To answer the first question, we compute

$$P(B = t|A = t, C = t) \propto \frac{2}{3} * \frac{1}{2},$$
$$P(B = f|A = t, C = t) \propto \frac{1}{3} * \frac{1}{3},$$

then we normalize these to sum to 1, and we get

$$P(B = t|A = t, C = t) = \frac{3}{4} = 0.75.$$

To answer the second question, we compute

$$P(B = t|A = f, C = f) \propto \frac{3}{7} * \frac{1}{2},$$
$$P(B = f|A = f, C = f) \propto \frac{4}{7} * \frac{2}{3},$$

then we normalize these to sum to 1, and we get

$$P(B = f|A = f, C = f) = \frac{16}{25} = 0.64.$$

With most belief networks, it is not reasonable to perform exact inference like this. That's why we are testing with a very simple network. Now, just to be absolutely certain that we did our math correctly, let's simulate it too. Here is some code that will generate a sample from the joint distribution of this simple network:

```
if rand.uniform() < 2.0 / 5.0
    then a = true; else a = false
if a == true
    if rand.uniform() < 2.0 / 3.0
        then b = true; else b = false
else
    if rand.uniform() < 3.0 / 7.0
        then b = true; else b = false
if b == true
    if rand.uniform() < 1.0 / 2.0
        then c = true; else c = false
else
    if rand.uniform() < 1.0 / 3.0
        then c = true; else c = false
```

In this code, “rand.uniform()” draws a random floating point value uniformly distributed between 0 and 1.

We could easily run this code in a loop and draw several million samples from the joint distribution of this network. Once we had all those samples, we could analyze them to answer probabilistic questions. To answer the first question, we find all of the samples where A and C are both true. Among these samples, we find that B is true in about 75% of them. To answer the second question, we find all of the samples where A and C are both false. Among these samples, we find that B is false in about 64% of them. The more samples you collect, the closer to these values you will get. This confirms that we did our math correctly.

So, why can't we use this approach as a general solution for Bayesian inference? Well, theoretically you could, but it is not very practical. Unless the pattern of given values occurs very frequently, most of the samples you draw will simply be rejected. If your given values contain any continuous values, then the odds are extremely low that you will draw the exact value that you are looking for. Gibbs sampling, on the other hand, samples the conditional joint distribution of the belief network much more efficiently.

So, now we test our Gibbs sampler, and we find that it computes the same values that we obtained by the other two methods. Yay! Now, we are ready to move on to more challenging problems.

3.4.4 Metropolis

If you try to use the pseudo-code that we previously presented for Gibbs sampling with a continuous distribution (such as a Normal distribution, or a Beta distribution), you would encounter a big problem: the number of possible values is infinite! How do you draw from a distribution if you cannot reasonably compute the probability of every possible value?

Metropolis is one algorithm that solves this problem. Here is pseudo-code to draw a sample for this node with Metropolis:

```
Let $g$ be the current value of this node.
Draw a candidate sample, $s$, from a normal distribution with a mean of $g$.
If the distribution of this node is discrete,
    round $s$. That is, $s = \lfloor s + 0.5 \rfloor$.
Temporarily set the value of this node to $s$.
$e = \phi(s|B)$, where $B$ is the current values of all the parent nodes of this node.
For each child node, $c$
    $e = e * \phi(c|D)$, where $D$ is the current values of all the parent nodes of $c$, and $\phi$ computes a probability density or likelihood.
Restore the value this node to back to $g$.
$f = \phi(\mu|B)$
For each child node, $c$
    $f = f * \phi(c|D)$
Let r be a random value between 0 and 1.
if r < e / f
    $g = s$
```

Basically, this algorithm does Gibbs sampling with two values: once with μ, the current value of this node, and once with s, a candidate new value that was drawn from the normal distribution (with mean μ). It then compares the likelihoods of these two values, and probabilistically chooses which one to use for the next value of this node, in such a way that these samples will converge to represent whatever distribution this node represents.

Todo: explain why this works

Todo: A good way to adjust \sigma is to track the variance in the samples

3.4.5 Metropolis-Hastings

Metropolis-hastings is an improvement to the Metropolis algorithm that enables one to use an arbitrary distribution, instead of the normal distribution, to draw candidate samples.

Todo: describe the algorithm

3.4.6 Naïve Bayes

Todo: write this section

4 Artificial Neural Networks

Todo: introduce the topic

4.1 Basic optimization techniques

Consider the equation, $e = f(w)$. If we think of this equation as a machine, then w goes in, and e comes out. We might draw this equation with a box diagram:

$$w \rightarrow \boxed{f} \rightarrow e$$

In optimization, f is called the *objective function*, or *target function*. It evaluates the thing being optimized, w, and returns an error term, e. Optimization involves trying to find w, such that e is minimized.

Alternatively, e could represent “goodness” instead of error. In that case, the objective would be to find w, such that e is maximized. It would be a trivial matter to negate the objective function, so there is no need to make a big deal about whether optimization involves minimization or maximization. When maximizing, the objective function is sometimes called a *fitness function*. When minimizing, it is sometimes called an *error function*, or *loss function*.

Optimization can be described as seeking *optima* or *extrema*, which terms include both maxima and minima. (O*ptima* is the plural form of *optimum*. *Extrema* is the plural form of *extremum*. And so forth.) Sometimes it is nice to use these terms so you do not have to specify whether you are seeking to maximize or minimize the objective function.

Often, the objective function has multiple inputs (or *variables*), $e = f(w_1, w_2, w_3, \cdots, w_n)$. This is more easily written using vector notation, $e = f(\mathbf{w})$. In this case, the bold-face font indicates that $\mathbf{w}$ is a vector. Often, the size of $\mathbf{w}$ is not even specified. In these cases, the reader is expected to understand that f operates on some number of input values, and it is probably not necessary to know how many inputs there are in order to understand the point being made. Most optimization techniques are designed to be able to handle any number of inputs.

4.1.1 Example

Objective functions can be mathematical equations, but they do not have to be. You can optimize over other things too. The inputs to the objective function can be continuous scalar values, but the do not have to be. As the following example illustrates, sometimes we optimize other types of values.

4.1.1.1 Non-traditional example

Suppose you had a computer program that could simulate a very simple cellular organism if you provide an encoding of its DNA. (The technology for such a sophisticated computer program may not yet exist, but we are getting closer at a rapid rate.) Suppose you are interested in seeking an organism that can survive as long as possible in a simulated environment.

To describe the DNA, you provide a list of nucleotides from the set {G, A, T, C}. For example, you might provide an input string like, “GGTATCCAGGAACCTTAGACTATATAC...”. After some amount of computation, the program reports how long the simulated organism survived.

In this example, the computer program that simulates organisms is the objective function. Since this objective function is relatively computationally expensive, we will probably want to give careful attention to how frequently the optimization technique we choose uses it.

In this example, the nucleotides are the inputs. If the DNA for a simple organism consists of 50000 nucleotides, then our objective function has 50000 inputs. That means there are 4^{50000} possible unique inputs. We couldn't possibly try them all, even if our objective function were extremely efficient, so we really need to make sure that we choose an optimization technique that is deliberate and careful, rather than systematic and thorough.

Often, the set of possible inputs is called the *input space*. In this example, we have a really big input space. Each input is a separate *dimension* in the input space. Since each dimension has 4 possible values that have no implicit ordering, these are *categorical* (also called *nominal*) dimensions (as opposed to *continuous* dimensions, which would have real values).

In this case, it is unlikely that we will actually find the optimum. Even if we did stumble across the very best DNA sequence, we would probably never know for sure that it was the best one, because it would take too long to try all the others. So, in this case, the objective of optimization is not really to find the optimum, but to find the best input vector (DNA sequence) that we can find. With many interesting optimization problems, finding the true optimum is not really reasonable. This does not mean that optimization is futile. It just means that the goal of optimization is to find the best input vector that can be found with limited computational resources.

4.1.2 Grid search

Grid search, also known as *brute force search*, is one of the most basic optimization techniques. It can be described as, *try all reasonable input vectors, and choose the one with the lowest error (or highest goodness).*

It might be said that grid search is easy on the programmer, but hard on the machine. It is trivial to implement, but often takes a very large amount of computation to run.

It is well-suited for optimizing over categorical inputs, because it can try each possible value. For continuous inputs, it is common to choose some range and granularity. For example, for a continuous input, you might try all values between 0 and 1 at increments of 0.1.

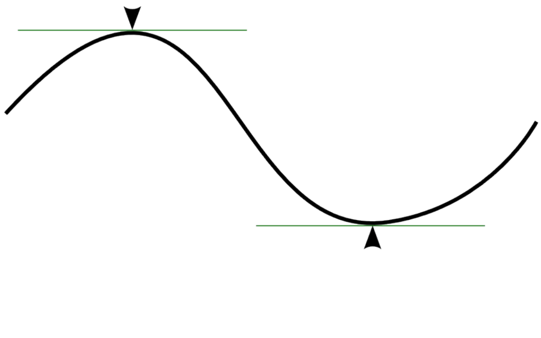
The biggest problem with grid search is that it scales very poorly with the dimensionality of the input space. For example, suppose the input space has 100 continuous dimensions. Even if you only try 2 possible values in each dimension, which would be very poor granularity for most applications, you would have to use the objective function 2^{100} times, which is far too much for a modern computer. On the other hand, if your objective function only has 2 continuous inputs, you could sample each input with a granularity of 1000 values, and still only call the objective function 1 million times, which is fairly reasonable if the objective function is not too complex.

4.1.3 The calculus approach

One approach for optimization is frequently taught with calculus. This approach can be described in three steps:

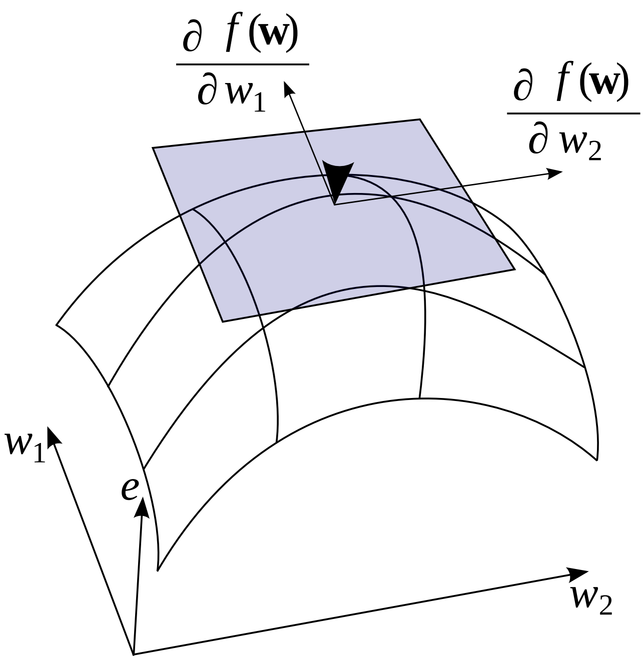
1. Differentiate the objective function. That is, find a formula for $f'(x)$.
2. Solve to find all possible values for x that satisfy the equation $f'(x) = 0$. These are called “critical points”.
3. Test each critical point to see which maximize or minimize $f(x)$.

The intuition commonly given for this approach is that the slope of a line tangent to f will be 0 at potential extrema.



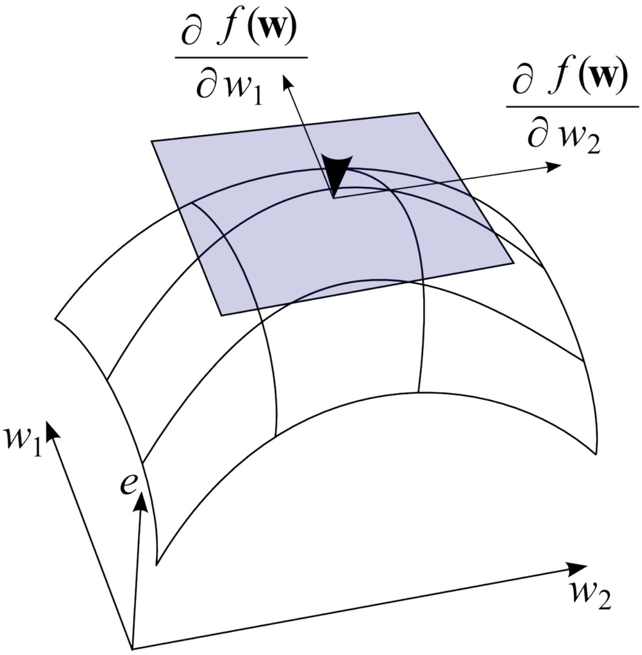
If the derivative equation is linear, then there will be only one critical point. Objective functions with only one critical point are said to be *convex*. If the target function is a 1000th order polynomial (meaning a polynomial where the biggest exponent is 1000), then there may be 1000 solutions, and hence, 1000 points that must be tested.

Note that extrema may also occur at $+\infty$ and $-\infty$, so it is necessary to also test these two values in addition to all of the solutions to the derivative equation.



$\frac{\partial f(\mathbf{w})}{\partial w_1}$
$\frac{\partial f(\mathbf{w})}{\partial w_2}$
w_1
e
w_2

Now, let us consider how this approach generalizes for a function with two inputs. In this case, the tangent of the objective function would be a plane instead of a line.


$\frac{\partial f(\mathbf{w})}{\partial w_1}$
$\frac{\partial f(\mathbf{w})}{\partial w_2}$
w_1
e
w_2

It takes two values to describe the slope of a plane. Visually, it can be observed that critical points occur only where the plane is parallel to both the w_1 and w_2 axes. (*Axes* is the plural form of *axis*. *Axes* is pronounced ax-eze.)

The slope of the tangent plane of the objective function, also called the *gradient*, is described by the partial derivatives of the objective function with respect to each of the input variables. So, critical points occur where

$$\frac{\partial f(\mathbf{w})}{\partial w_1} = 0 \quad \text{and} \quad \frac{\partial f(\mathbf{w})}{\partial w_2} = 0\,.$$

∂ is the symbol for a partial derivative. It refers to an infinitesimal amount of change in a single value. Partial derivatives are almost always used in ratios. These equations are read, *the partial derivative of the objective function with respect to w_1 equals zero*, and *the partial derivative of the objective function with respect to w_2 equals zero.*

Partial derivatives are found by treating all the other variables as if they were constants, and then finding the derivative as if it were a function with only one variable. For example, suppose

$$f(\mathbf{w}) = 1 + 2w_1 + 3w_2 + 4w_1w_2 + 5w_1^2 + 6w_2^2.$$

Then,

$$\frac{\partial f(\mathbf{w})}{\partial w_1} = 2 + 4w_2 + 10w_1, \text{ and}$$

$$\frac{\partial f(\mathbf{w})}{\partial w_2} = 3 + 4w_1 + 12w_2.$$

To find the critical point(s), therefore, we solve the set of equations,

$$2 + 4w_2 + 10w_1 = 0,$$

$$3 + 4w_1 + 12w_2 = 0,$$

and we arrive at the solution,

$$w_1 = -\frac{3}{26}, \text{ and } w_2 = -\frac{11}{52}.$$

In this example, there was only one solution to our set of equations because both derivative equations were linear. If they had both been 3rd order polynomials, then there may have been as many as 9 critical points. If they were $k^{\mathrm{t}h}$ order polynomials, then there may have been k^2 critical points. Additionally, we must also test the points,

$$w_1 = +\infty, w_2 = +\infty,$$

$$w_1 = +\infty, w_2 = -\infty,$$

$$w_1 = -\infty, w_2 = +\infty, \text{ and}$$

$$w_1 = -\infty, w_2 = -\infty.$$

So, what if our objective function has d inputs? Well, then there will be about k^d critical points, and 2^d points at infinity that also need to be tested. This is very poor scalability. For example, suppose $k = 3$ and $d = 20$. (These are very conservative values. Many interesting problems involve equations with many more than 3 possible solutions, and it is quite common to optimize over objective functions with many more than 20 inputs.) In this case, we would need to test more than 3 billion points! ($3^{20} + 2^{20} = 3487832977$).

While the computational cost of grid search is exponential in the number of inputs (or variables), the computational cost of the calculus approach is exponential in the complexity of the objective function. Grid search is only tractable iff the number of variables is small. The calculus approach is tractable iff the number of variables with an order of 3 or higher is small. That's better, but it really only solves the scalability problem for very simple objective functions.

Another limitation of the calculus approach for optimization is that it can only handle objective functions that are differentiable. In the real world, many of the functions we wish to optimize over are difficult to differentiate. Many of them are not even differentiable at all. Some of them are not even describable with a simple math equation. Suppose your objective function involves running a simulation. How are you going to differentiate that?

4.1.4 Hill climbing

Suppose you were brought to the base of a hill, blindfolded, and challenged to make your way to the top of the hill. How might you do it?

You might feel the surrounding ground with your foot, decide which direction seems to ascend, and walk in that direction. Before each step, you would probably feel around some more, to make sure you do not step over a ledge. Eventually, there is a good chance you would be able to make your way to the top of the hill.

Hill climbing is a very simple optimization approach. It can be summarized as:

1. Try stepping in each possible direction. (Undo each step after trying it.)
2. Redo the step in the direction that produced the biggest improvement.
3. Repeat until no further improvements can be found.

We assume that the objective function forms a sort of “hill” if we are maximizing or a “bowl” if we are minimizing. If we are minimizing, then it is sometimes called the *error surface*. The objective is to find the highest point on the hill, or the lowest point on the error surface. Because of its simplicity, hill climbing can be an attractive optimization technique.

4.1.4.1 Local optima

Hill climbing requires the user to provide a starting point. If the error surface is convex (meaning it forms only one hill-like or bowl-like shape), then the starting point does not really matter. No matter where it starts, it will end up in the same place. It may take longer to get there from some starting points, but the final results will still be the same.

If the error surface is non-convex, then the starting point can have a significant influence on the quality of the results. Some starting points may cause it to find the *global optimum*. That is, the vector of input values that make the objective function give the smallest error (or largest goodness value). Unfortunately, some starting points may only cause it to find a *local optimum*. This is an input vector with a better error value than any other point close to it, but is not necessarily the best point overall.

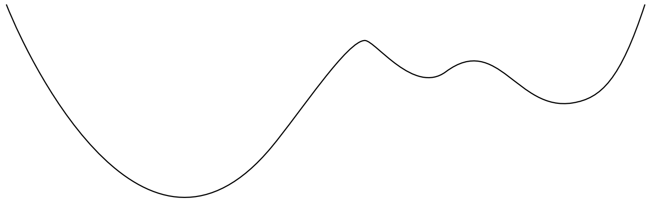
Of course, it is always preferable to find the global optimum, but there are many applications where finding a good local optimum may be good enough. These are applications for which hill climbing is well-suited. Furthermore, it is not always trivial to determine whether a local optimum is the global optimum. Often, the best we can do is ask, *is this one good enough?*

The region immediately around a local optimum is, by definition, locally convex. That means the error surface forms a bowl-shape immediately around the local optimum.

4.1.4.2 Random restarts

A simple improvement to hill climbing is to do it multiple times, each time starting from a different random position. If one starting point causes it to fall into a local optimum, perhaps the next one will cause it to find the global optimum.

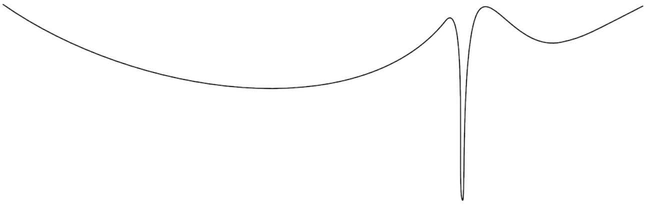
The shape of the error surface has a big impact on the effectiveness of this approach. For example, suppose you want to use hill climbing to minimize over the following error surface.



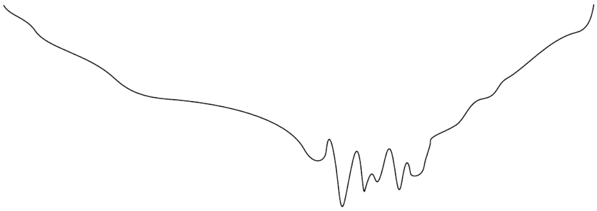
If the random starting

point falls on the right-side of the surface, the hill climber will find one of the local optima. But if it falls anywhere on the left-side, it will find the global optimum.

By contrast, suppose the error surface was more like this,



or like this,



In these cases, most starting points would cause it to fall into a local optimum. It may take a great many random restarts before a starting point is found that results in finding the global optimum. When the dimensionality is high, there might even be such extreme cases that no reasonable number of random restarts will ever suffice.

4.1.4.3 The non-intuitive effects of dimensionality

The effects of dimensionality on optimization techniques are not always intuitive. Obviously, the size of the input space is exponential in the number of inputs, but humans often have difficulty comprehending the significance of exponential growth.

In the case of grid search, dimensionality (meaning lots of input variables) makes the algorithm completely intractable. With hill climbing, however, this is not necessarily the case. Even though the space may be very big, hill climbers still follow a single path through this space. Although this path winds through all of the input dimensions, the path itself is essentially one-dimensional.

Another effect of dimensionality is that it impacts the likelihood of local optima. Local optima occur only where they are bounded on all sides by locations that represent less-effective input vectors. In one-dimensional space, one only needs to find a value, such that making that value bigger would make the error value worse, and making that value smaller would also make the error value worse. If there are 1000 variables, then local optima occur only where all of them are locally optimal at the same time. What are the chances that a small adjustment to any of the 1000 variables in either direction would always make the error value worse? Well, that probably depends on the nature of the input variables themselves. If they are all highly correlated, then they will behave as if there were far fewer of them. If they all represent completely independent concepts, then such a situation is unlikely to occur unless this local optimum is also the global optimum. Consequently, hill climbing seems to work very well with problems that involve simultaneously optimizing many independent variables. This effect is interesting because it seems so counter-intuitive.

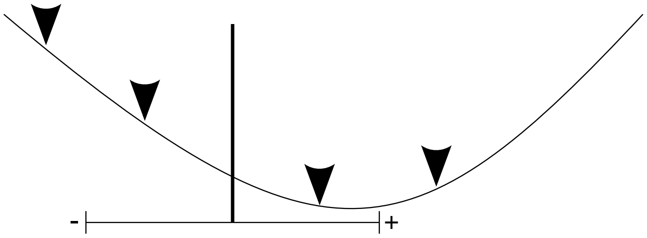
Another counter-intuitive effect of dimensionality is that it reduces the effectiveness of random restarts. In low-dimensional spaces, it is often a good idea to try many random restarts. In high-dimensional spaces, however, it becomes increasingly intractable to sufficiently sample the potential starting points. In the limit, as the error surface becomes increasingly complex, doing random restarts becomes just as expensive as grid search. Further, dimensionality exacerbates the problem of optima that are difficult to find. If some optimum has only a small convex region around it, then that region will be much more difficult to find in a large space. Hence, if there are lots of variables to optimize, there is a good chance that performing hill climbing in one shot will give pretty-good results, and additional attempts may not improve your results very much.

4.1.4.4 Continuous variables

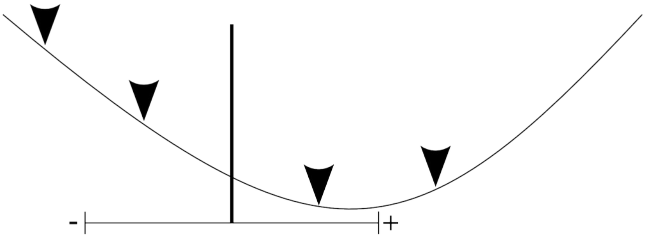
Although hill climbing is most-intuitive with categorical values, it can also be done with continuous values. Perhaps the most common approach for implementing a continuous-space hill climber is to step in a random direction, and then undo the step if it does not improve the error value.

Unfortunately, this is a very poor approach. When it starts out, approximately half of the steps it takes will improve the input vector, and about half of them will fail to improve it. Consequently, it works very well for getting into the general vicinity of a good value. As it approaches a locally-convex region, however, the chances decrease of randomly stepping in a direction that improves the input vector. Consequently, this approach tends to get stuck much sooner than it really needs to. Further, it is unlikely that each dimension will require refinements of approximately the same size. Some dimensions require very big refinements, while others require very small fine-tuning.

In order to do effective hill-climbing in continuous space, it is essential to maintain a unique dynamic step size for every dimension. This requires a little bit more work on the part of the programmer, but can make a significant difference in the effectiveness of the hill climber.



My favorite way to implement hill climbing in continuous spaces is to try 4 candidate steps in each dimension. One step is a little-bit smaller than the current step-size for this dimension in the positive direction. One step is a little-bit bigger than the current step-size for this dimension in the positive direction. One step is a little-bit smaller than the current step-size for this dimension in the negative direction. One step is a little-bit bigger than the current step-size for this dimension in the negative direction.



Whichever of the 4 candidate steps gives the lowest error value will be accepted, and the step size will be adjusted to match the size of the step taken. This way, the hill climber will accelerate when the bigger steps give the better error value, and it will decelerate when the smaller steps give the better error value. Perhaps more significant, some dimensions will accelerate, while others decelerate, such that the entire system will quickly hone in on the precise position of the nearest local optimum.

4.1.4.5 Ridges

Sometimes multiple variables need to be adjusted together in order to minimize the error function, while adjusting them individually has little beneficial effect. This results in a narrow alley in the error surface (or ridge in the goodness surface), which is not axis-aligned. In such cases, hill climbers will generally begin taking very small steps. This has the effect of making optimization take a very long time.

4.1.5 Gradient descent

Gradient descent (also known as gradient ascent) is an improvement to hill climbing. It may be considered to be a hybrid approach between the calculus approach and hill climbing.

Like a hill climber, gradient descent begins at an arbitrary point, and it takes small steps to move toward the nearest local optimum. The difference is that instead of adjusting each dimension individually, it calculates the direction that will most directly descend. Sometimes, this approach is called *tobogganing*, because it can be analogous with riding a toboggan (or sled) in the most downhill direction.

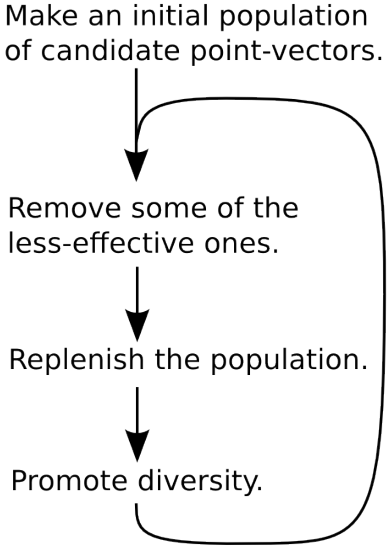
The gradient can be thought of as a vector that points in the direction that moves most directly uphill. So, if you want to minimize the error, you should move in the opposite direction of the gradient vector. The gradient vector is calculated as the partial derivative of the error with respect to each input value. This is the same set of values we calculate to use the calculus approach, but instead of solving for where all the gradient values are zero, we will simply take a small step in the gradient direction (or the opposite of the gradient direction), and repeat until we arrive at a local optimum.

Unlike hill climbing, gradient descent requires only a single step-size, because the gradient itself will cause each dimension to be updated with suitable proportions. Also unlike hill climbing, gradient descent does not struggle with narrow alleys (or ridges), because it can “travel” in any direction, whether axis-aligned or not. (It does still struggle, however, with narrow alleys that curve sharply or spiral, but these are somewhat less common.)

We will discuss gradient descent in much more specific detail in subsequent chapters.

4.1.6 Evolutionary optimization

Evolutionary optimization is a class of optimization techniques loosely inspired by natural evolution. There are many variations of evolutionary optimization, but they all follow a similar flow:


Make an initial population of candidate point-vectors.
Remove some of the less-effective ones.
Replenish the population.
Promote diversity.

There isn't really a “right way” to implement any of these steps, but we will discuss some common themes.

4.1.6.1 Initialization

The initial population is usually generated completely at random. However, if some prior knowledge is available, evolutionary optimization can benefit greatly from having a few members of its population initialized with better values.

If all of the elements in the point-vectors are categorical, then each member of the population may be analogous with a “genome”, and the optimization technique may be called a “genetic algorithm”. However, the same technique works for optimizing continuous values, so we refer to it using the slightly broader term “evolutionary optimization”.

4.1.6.2 Selection

Killing off the weaker member of the population and letting the more fit members survive is called *selection*. The two most-common ways to implement selection are *fitness-proportionate selection*, and *tournament selection*. As with any optimization technique, these methods require that some mechanism exists to evaluate the *fitness* of any point-vector.

In fitness-proportionate selection, the fitness of every member of the population is evaluated. Each member is then deleted with a probability proportional to its fitness. One limitation of fitness-proportionate selection is that its effectiveness depends heavily on the mechanism used to evaluate fitness. That is, the ratios between evaluations of fitness will carry over into the optimization. As the population increases in fitness, these ratios will tend to even out, leading to greater diversity within the population and slower progress toward the global optimum. Is that good or bad? Well, it's not entirely clear.

In tournament selection, two members of the population are picked to fight with each other. These two members are evaluated. With some probability (greater than 50%), the better point survives. Sometimes, however, the weaker point prevails and the better point dies.

Why do both of these techniques ensure that there is some probability of the weaker points surviving? Because we don't want to get stuck in a local optimum. Sometimes, you have to take a few steps back before you can take many steps forward. One of the the key ideas of evolutionary optimization is to always have a few members of the population exploring the less-desirable regions of the space, just in case they find a much better optimum. If you select the better members of the population too strongly, then evolutionary optimization will behave much like a hill-climber, always gravitating toward better locations—except much less efficiently, since you have to take care of a whole population of candidate points. If you want it to behave like a hill climber, you'd really be better off just using a hill climber.

4.1.6.3 Replenishing the population

After you kill off some members of the population, you typically want to fill the missing slots. (Of course, you could use a growing or diminishing population, if that would be advantageous for some reason, but the most common approach is to use a static population size.)

One simple approach is to randomly generate a new point-vector. Another approach is to randomly pick another members of the population and clone it to fill a missing slot. You could use a fitness-proportionate approach to choosing which member of the population gets to be cloned. By far, however, the most common approach for replenishing the population is *crossover*.

Crossover is inspired by sexual reproduction. It chooses two parents, and combines them to produce a new child. For example:

```
Parent 1: GATACATTACATACCTGGA
Parent 2: AATTCCACATAGGCCTAAC
Child:    GATACATCATAGGCCTAAC
```

In this case, the first 7 values for the child came from Parent 1, and the remaining values came from Parent 2. The cross-over point (7, in this case) is usually chosen randomly. Of course, you could use multiple crossover points, if you want,

```
Parent 1: GATACATTACATACCTGGA
Parent 2: AATTCCACATAGGCCTAAC
Child:    GATACATCACATGCCTAAC
```

but this could also be accomplished by using single-point cross-over and running for several generations. If the order of the values in the genome is not relevant, as is the case with point-vectors in continuous space, then it makes sense to choose a random parent for each element:

```
Parent 1: GATACATTACATACCTGGA
Parent 2: AATTCCACATAGGCCTAAC
Child:    GATACCTCACATGCCTGGC
```

In continous space, another good option for producing offspring from two parents is interpolation. This is done by picking a random value, w, between 0 and 1, and computing the child as

$$\mathbf{c} = w\mathbf{p}_1 + (1 - w)\mathbf{p}_2.$$

If you allow w to be less than 0, or greater than 1, then it will perform extrapolation. Instead of picking a point between the two parents, this will pick a point farther away from one of them.

Many other approaches for filling missing slots in the population have been attempted. Many of them are inspired by biology in some way. For example, one approach simulates continental separation. (For example, species in Africa would be unlikely to breed with species in Australia.) Another approach simulates social rules that influence which members of the population will breed. Ultimately, since there are many ways to fill the missing slots, and no one really knows which way is best, a good approach is to implement them all and randomly pick which one to use when there is a slot to fill. In evolutionary optimization, adding lots of complex ideas is generally considered to be beneficial. Some work better under some conditions, and some work better under different conditions, so if we implement them all, then we can let the selection step sort out what works best at any time.

4.1.6.4 Promoting diversity

The primary reason for choosing evolutionary optimization over faster optimization techniques is to avoid problems with local optima by exploring diverse regions of the space. If the entire population converges to the same point, then the space is not explored very effectively. So, it is necessary to use a few techniques to promote diversity.

The most common way to promote diversity is *mutation*. This is implemented by randomly picking an element of the genome and randomly changing it.

```
Before:  GATACCTCACATGCCTGG
After:   GATACCTCTCATGCCTGG
```

If the values are continuous, mutation can be implemented by perturbing an element by a random amount. This is typically implemented by drawing a random value from a Normal distribution and adding it to the element. What value should be used for the deviation of the Normal distribution? Well, if we knew the answer to anything that specific, then we probably wouldn't be using evolutionary optimization, so this would be yet another parameter picked by the implementor.

Another option is to implement the possibility of catastrophic mutations, which perturb elements by a much larger amount, or perturb all (or a large portion) of the genome. Those who prefer continuity may even randomly choose how catastrophic a mutation event will be, or make different slots in the population more or less susceptible to mutations.

One useful way to promote diversity may be to automatically kill off members of the population if they approach too closely to another member. This will ensure that no two members of the population begin exploring the same region of the space.

Ultimately, the three steps that we described in the flow chart for evolutionary optimization do not really need to be three distinct steps at all. They are more like guiding principles. For example, cross-over is a mechanism for replenishing the population, but it is also a mechanism to promote diversity. Selection can also be used to promote diversity. Essentially, any algorithm that uses a population of candidate vectors to explore diverse regions of the space, and draws some indirect inspiration from evolution, can be termed evolutionary optimization.

4.1.7 Comparison

Let us compare the properties of the simple optimization techniques that we have discussed so far:

	Scalability with dimensionality	Optimality	Limitations
Grid search	Terrible	Approximates global optimum	Imprecise.
Calculus	Terrible	Always finds global optimum	Requires differentiable model.
Hill climber	Good	Gets stuck in local optima	Needs a starting point.
Gradient descent	Great	Gets stuck in local optima	Needs a starting point. Requires differentiable model.
Evolutionary optimization	Good	May find global optimum	Depends on many heuristics.

	Rate of convergence	Parallelizability	Ease of implementation
Grid search	N/A	Great	Easy
Calculus	N/A	Great	Hard. Need a math solver.
Hill climber	Good	Poor	Easy
Gradient descent	Great	Poor	Medium. Model-specific
Evolutionary optimization	Poor	Great	Easy

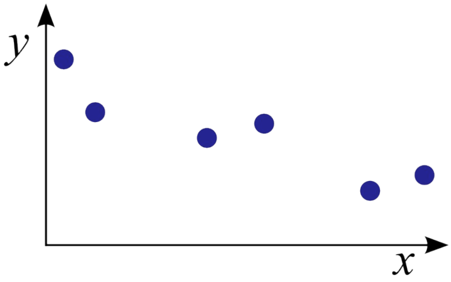
It should be fairly clear that no optimization technique is best for every problem. There are, of course many other optimization methods, and several hybrid methods that attempt to combine various optimization methods to obtain desirable properties.

4.2 Regression

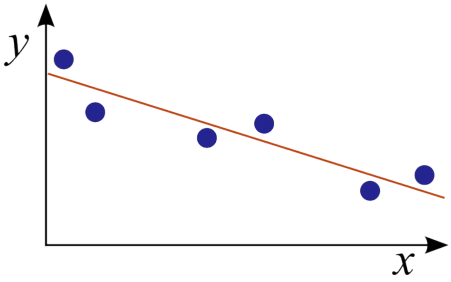
Regression involves using some optimization technique to fit a model to some data. For example, suppose we decide to use the model,

$$\hat{y} = mx + b.$$

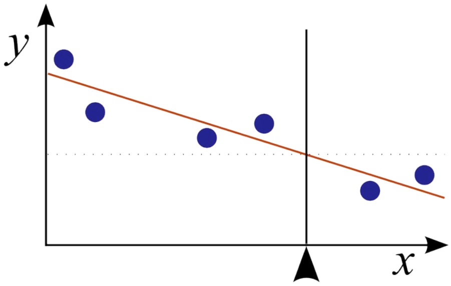
You might recognize this as the equation for a line. Using this model with regression is called *linear regression.* For example, if we had some data, like this,



then linear regression would find the line that fits to the data, like this:



Why do we want to find a line that fits to our data? Well, we could use it to make predictions. If we only knew a value for x, the equation for the line would enable us to predict a corresponding value for y,



The trained model is called a *hypothesis*. A hypothesis is a function that maps from a vector of input values, $\mathbf{x}$, to a corresponding vector of output values, or labels, $\mathbf{y}$. We could draw the hypothesis as a machine or flow diagram like this:

$$\mathbf{x} \rightarrow \boxed{h} \rightarrow \hat{\mathbf{y}}$$

In the case of linear regression, the hypothesis consists of the values m and b. So, how exactly do we calculate these values?

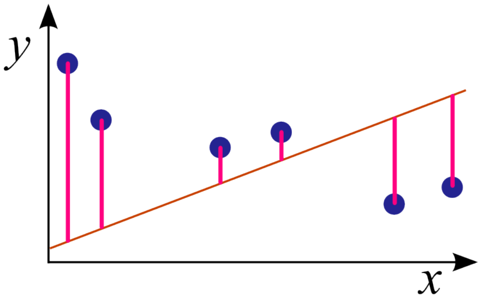
First, we need an objective function. A common objective function for regression is *sum squared error* (SSE),

$$\text{SSE} = \sum_i (y_i - \hat{y}_i)^2,$$

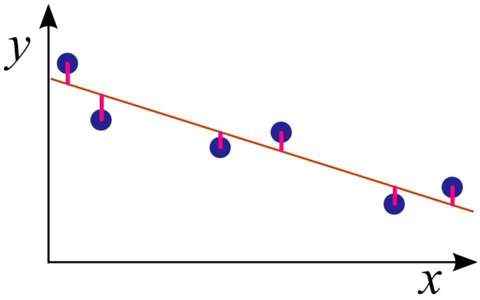
or

$$\text{SSE} = \sum_i (y_i - h(x_i))^2.$$

This objective function will return a very big value if the model fits the data poorly,



and will return a very small value if the model fits the data well.



Note that there are multiple functions involved in this process. Be careful not to confuse the model with the objective function. They are both functions, but they have different purposes. The model is the function we want to fit to the data. The objective function tells us how good (or bad) of a job we have done. In this example, the objective function is,

$$\text{SSE} = \sum_i (y_i - h(x_i))^2,$$

and the model is

$$y = mx + b.$$

Note that this model has two values, m and b, which need to be "*trained*". These are called *model parameters*. Training is done by using some optimization technique to find model parameters that cause the model to fit to the data well. When we settle on good values for the model parameters, it is called a hypothesis. Our hypothesis in this example might be,

$$\hat{y} = h(x) = -\frac{1}{5}x + 2.$$

Note that x and y are not trained. They are called *variables*. We leave their values free so that our hypothesis has something to operate on. In mathematical notation, x and y are often used to refer to variables, meaning values that change frequently, and other letters are often used to refer to parameters, which change less frequently. However, they are all really just values. You could just as well train x and leave m free, if you wanted to. (With this model, that would make no difference at all.)

Other models, besides lines, can also be used for regression. For example, we might fit the model

$$\hat{y} = a + bx + cx^2 + dx^3 + ex^4$$

to our data. This would be called polynomial regression. In that case, the coefficients, a through e would be the model parameters that we train, or optimize to obtain a hypothesis.

4.2.1 Multivariate linear regression

(Before reading this section, it might be a good idea to look thorough the appendix that reviews essential linear algebra concepts.)

A good regression technique should be able to support any number of input dimensions as well as any number of output dimensions. Linear models generalize well to multiple dimensions. For example, suppose our training data has 3 input dimensions, $\langle x_1, x_2, x_3 \rangle$, and 2 output dimensions, $\langle y_1, y_2 \rangle$. A linear model for this problem would look like this:

$$\hat{y}_1 = m_1 x_1 + m_2 x_2 + m_3 x_3 + b_1$$

$$\hat{y}_2 = m_4 x_1 + m_5 x_2 + m_6 x_3 + b_2.$$

We could write this in matrix form,

$$\hat{\mathbf{y}} = \mathbf{M}\mathbf{x} + \mathbf{b}.$$

This equation looks quite similar to the scalar form,

$$\hat{y} = mx + b,$$

except the m is capitalized, and all the letters are bolded. This implies that $\mathbf{M}$ is a matrix, and $\mathbf{y}$, $\mathbf{x}$, and $\mathbf{b}$ are vectors.

4.2.1.1 Linear models have convex error surfaces

When SSE (or MSE, or RMSE) is used as the objective function with a linear model, the resulting error surface is always convex. This is a very nice property. It means that we can use any reasonable optimization technique to compute $\mathbf{M}$ and $\mathbf{b}$, and we will always arrive at approximately the same hypothesis. Furthermore, this hypothesis will be the optimal for this model. In other words, there are no values for $\mathbf{M}$ and $\mathbf{b}$ that will result in a lower SSE with this data. (Of course, there may be still be better hypotheses that could be found with non-linear models.)

4.2.1.2 Ordinary least squares

With linear models, it is actually possible to use the calculus approach to compute the optimal hypothesis in closed form.

Although this is elegant from a mathematical perspective, it is not really very practical. It is usually faster to approximate these values with an iterative optimization technique than to compute them directly. Furthermore, it is more difficult to maintain numerical stability when computing them in closed form. Hypotheses trained with numerical optimization techniques tend to be slightly more precise than those computed in closed form. Most non-linear models are too complex to solve in closed form, so they must be trained using a numerical optimization technique. (So, this section is only here for completeness. You can skip reading this if you don't care about computing the optimal values for $\mathbf{M}$ and $\mathbf{b}$ for a linear model in closed form.)

Training a linear model by computing the model values in closed form is called *Ordinary Least Squares* (OLS). To derive OLS, let us start with the equation for SSE,

$$\text{SSE} = \sum_i ||\mathbf{y}_i - (\mathbf{M}\mathbf{x}_i + \mathbf{b})||^2.$$

To keep it simple for now, we will temporarily assume that both the inputs and outputs are already centered at the origin. This simplifies the equation to

$$\text{SSE} = \sum_i ||\mathbf{y}_i - \mathbf{M}\mathbf{x}_i||^2.$$

We want to find the $\mathbf{M}$ that minimizes this equation, which will occur where the derivative of the equation with respect to every element of $\mathbf{M}$ is zero. We express this by picking an arbitrary element, $m_{r,c}$, where r is any arbitrary row, and c is any arbitrary column, and setting it equal to zero:

$$\frac{\partial \sum_i ||\mathbf{y}_i - \mathbf{M}\mathbf{x}_i||^2}{\partial m_{r,c}} = 0.$$

You can interpret this to mean that changing any element in $\mathbf{M}$ by a very small amount will have no effect on SSE. In other words, slope is zero in all possible directions at the same time.

Since

$$||\mathbf{a}||^2 = \mathbf{a} \cdot \mathbf{a} = a_1^2 + a_2^2 + a_3^2 + \cdots,$$

we can rewrite the equation as

$$\frac{\partial \sum_i \sum_j (y_{i,j} - \mathbf{m}_j \cdot \mathbf{x}_i)^2}{\partial m_{r,c}} = 0,$$

where j iterates over each of the label dimensions. Since none of the squared terms can ever be negative, this equation can only be satisfied when all of them are exactly equal to 0. So, we could break it up into multiple equations,

$$\frac{\partial \sum_i (y_{i,1} - \mathbf{m}_1 \cdot \mathbf{x}_i)^2}{\partial m_{r,c}} = 0,$$

$$\frac{\partial \sum_i (y_{i,2} - \mathbf{m}_2 \cdot \mathbf{x}_i)^2}{\partial m_{r,c}} = 0,$$

$$\vdots$$

Keeping track of so many equations would be cumbersome, so instead, we express them all with a single equations as,

$$\frac{\partial \sum_i (\mathbf{y}_i - \mathbf{M}\mathbf{x}_i)^2}{\partial m_{r,c}} = \mathbf{0}.$$

This equation looks a lot like the one we started with, so you might think that we haven't really done anything yet, except change the magnitude bars to parentheses. But if you are paying careful attention, there is actually another pretty-big difference. Initially, we only had one equation. The numerator evaluated to a single scalar value. This one, however, represents multiple equations. The bold-face zero on the right-side of the equation indicates that each of these equations is individually equal to zero.

Further, these equations are in a form that is easier to work with. So, the next step is to use the chain rule,

$$\frac{\sum_i 2(\mathbf{y}_i - \mathbf{M}\mathbf{x}_i)\partial(\mathbf{y}_i - \mathbf{M}\mathbf{x}_i)}{\partial m_{r,c}} = \mathbf{0}.$$

We can divide both sides of the equation by 2. We can also drop the "$\mathbf{y}_i$" inside the ∂ expression, because changing an arbitrary element of $\mathbf{M}$ will have no effect on its value.

$$\frac{\sum_i (\mathbf{y}_i - \mathbf{M}\mathbf{x}_i)\partial(-\mathbf{M}\mathbf{x}_i)}{\partial m_{r,c}} = \mathbf{0}.$$

Now, we can work out the ∂ expression,

$$\sum_i (\mathbf{y}_i - \mathbf{M}\mathbf{x}_i) \times (-\mathbf{x}_i) = \mathbf{0}.$$

There are some significant details here that are easy to miss when you solve all of the equations at once, like this. If you want to really see how it all works out, I recommend writing out the individual equations (but I am not going to do that here). Even though we started out with just one equation for each label dimension, when we work out the ∂ term, it compounds each of them by the number of input dimensions. So, we are actually working with a matrix of equations. This matrix has one row for each label dimension, and one column for each input dimension. The $\mathbf{y}_i$ and the first $\mathbf{x}_i$ in the equation above are both column vectors, but the second $\mathbf{x}_i$ is actually a row vector.

Anyway, our objective, here, is to solve for $\mathbf{M}$. So, first we distribute,

$$\sum_i -\mathbf{y}_i \times \mathbf{x}_i + \mathbf{M} \times \mathbf{x}_i \times \mathbf{x}_i = \mathbf{0},$$

and then we can solve for $\mathbf{M}$:

$$\mathbf{M} = \left(\sum_i \mathbf{y}_i \times \mathbf{x}_i\right) \times \left(\sum_i \mathbf{x}_i \times \mathbf{x}_i\right)^{-1}.$$

So, we have solved for the case where the patterns and labels are both centered at the origin. But, what if they are not centered at the origin?

Well, we can adjust the equation for $\mathbf{M}$ subtract the centroids from both $\mathbf{x}$ and $\mathbf{y}$ to move them to the origin before it uses them. Likewise, we can use $\mathbf{b}$ to place the center where it is supposed to be. So, the model equations become

$$\begin{aligned}\mathbf{M} = &\left(\sum_i (\mathbf{y}_i - \bar{\mathbf{y}}) \times (\mathbf{x}_i - \bar{\mathbf{x}})\right) \times \\ &\left(\sum_i (\mathbf{x}_i - \bar{\mathbf{x}}) \times (\mathbf{x}_i - \bar{\mathbf{x}})\right)^{-1}, \\ \mathbf{b} = \bar{\mathbf{y}} &- \mathbf{M}\bar{\mathbf{x}}.\end{aligned}$$

We can use these equations to train a a linear model to fit to data with any centroid. In these equations, $\mathbf{x}_i$ refers to the input pattern of sample i in the training data, and $\mathbf{y}_i$ is the corresponding output, or label vector. $\bar{\mathbf{x}}$ is the centroid of the inputs, and $\bar{\mathbf{y}}$ is the centroid of the outputs.

4.2.2 Multivariate polynomial regression

Linear equations are a special case of polynomials. In the previous section, we examined how linear regression can be extended to handle multiple input variables. We found that linear models scale nicely to handle many input variables. Now, let us examine the more general class of polynomials for handling multiple input variables.

To begin, here is a familiar equation for a polynomial:

$$\hat{y} = a + bx + cx^2 + dx^3.$$

This polynomial has a degree of 3. The *degree* is the largest exponent in any of the terms. We can see that the equation for a line is just a polynomial of degree 1. Before we start talking about multivariate polynomials, let us rewrite our equation a little bit:

$$\hat{y} = w_1 x_1^0 + w_2 x_1^1 + w_3 x_1^2 + w_4 x_1^3.$$

This equation really says the same thing as the previous one. We just enumerated the coefficients, so we won't run out of alphabet letters. Also, we added a subscript 1 to each variable, x, so we could distinguish it from other input variables (such as x_2). It may look a little confusing to see both a subscript and a superscript on the same variable, but this represents nothing new. The subscript just specifies which variable we are talking about, and the superscript is the exponent.

Okay, let's look at an equation for a polynomial with two variables:

$$\begin{aligned} y = {} & w_1 x_1^0 x_2^0 + w_2 x_1^1 x_2^0 + w_3 x_1^2 x_2^0 + w_4 x_1^3 x_2^0 + \\ & w_5 x_1^0 x_2^1 + w_6 x_1^1 x_2^1 + w_7 x_1^2 x_2^1 + w_8 x_1^3 x_2^1 + \\ & w_9 x_1^0 x_2^2 + w_{10} x_1^1 x_2^2 + w_{11} x_1^2 x_2^2 + w_{12} x_1^3 x_2^2 + \\ & w_{13} x_1^0 x_2^3 + w_{14} x_1^1 x_2^3 + w_{15} x_1^2 x_2^3 + w_{16} x_1^3 x_2^3. \end{aligned}$$

Don't panic! This equation may look nasty, but it is really not very complex. There are 16 terms in this equation because we need one term for each pair: $\{x_1^0, x_1^1, x_1^2, x_1^3\} \times \{x_2^0, x_2^1, x_2^2, x_2^3\}$. (If we added a third variable, there would be 64 combinations. If we added a fourth one, there would be 256.) Making a term for each combination is all it takes to make a multivariate polynomial.

As with single variable polynomials, the degree of the polynomial is the largest degree of any term. But the *degree* of each term is actually the sum of its exponents. So, this is actually a 2-variable polynomial of degree 6. If we want to limit our polynomial to a degree of 3, then we need to throw out all the terms of higher degree:

$$\begin{aligned}\hat{y} = {} & w_1 x_1^0 x_2^0 + w_2 x_1^1 x_2^0 + w_3 x_1^2 x_2^0 + w_4 x_1^3 x_2^0 + \\ & w_5 x_1^0 x_2^1 + w_6 x_1^1 x_2^1 + w_7 x_1^2 x_2^1 + \\ & w_8 x_1^0 x_2^2 + w_9 x_1^1 x_2^2 + \\ & w_{10} x_1^0 x_2^3\end{aligned}$$

Well, that made it a little bit simpler! Now, we can also see that the multivariate linear model is just a polynomial of degree 1. If we throw out all the terms with a degree greater than 1, then we end up with the multivariate linear model.

So, are multivariate polynomials the answer to solving the world's most difficult problems? As we discuss in the next section, they are actually an excellent example of a very poor model.

4.2.3 What makes a good model?

In general, what is the best model to use for regression? One that contains a “true” hypothesis, of course. After all, the best-possible case of regression is to find the true or ideal hypothesis.

But, what is a “true” or “ideal” hypothesis? It is one that accurately describes the very Universe that produced the data. For example, suppose your data consists of measurements of the positions of falling objects. In that case, the ideal hypothesis would be the physical equations that describe the behavior of gravity. Since these equations are not very complex (well, if we assuming Newtonian physics and ignore relativity, that is), then finding a model capable of representing them may be quite reasonable. However, what if your data contains measurements of biological phenomena, such as the social behaviors of Rhesus monkeys?

In the case of predicting biological phenomena, the ideal hypothesis would essentially have to be equations sufficient to simulate the complete environment and biological creatures that you measured to obtain the data. It would take a pathologically flexible model to contains a hypothesis that complex! And, even if you had a model that flexible, training it (by searching its model-space to find the right hypothesis) would probably be a big nasty job. And, even if you had enough computing power to do it, there is yet another big complication: How will you know when you find the right hypothesis? We could optimistically assume that any hypothesis that matches our observations is probably a good one, right? Actually, no. If your model space is big enough to contain the complex ideal hypothesis, it is probably also big enough to contain a lot of pathological hypotheses that just happen to fit all known observations, but behave pathologically in other cases. Such optimism is an excellent example of human intuition failing to comprehend the utter vastness of high-dimensional space.

With complex problems, the answer seems to be to stop trying to find the true hypothesis, and settle for one that makes predictions that are good enough, because you will never be enough data to validate the true hypothesis anyway.

William of Ockham (c. 1287-1347) coined the famous maxim, “Entia non sunt multiplicanda praeter necessitatem”, which translates to approximately, “Don't add complexities to your explanations without necessity.” It has become known as Ockham's razor.

In the context of regression, Ockham's razor (sometimes spelled “Occam's razor”) might say, “Don't use a model that is more complex than is necessary to explain the observed data.”

Sometimes, people criticize Occam's razor by pointing out that “truth” is often not very simple. If we use Occam's razor, we will choose the simplest hypothesis that is consistent with the observed data. Then, if we compare it with the hypothesis that turns out to be “true”, the truth will usually be much more complex than the one Occam's razor told us to pick. In other words, Occam's razor does not necessarily lead us to choose the “true” hypothesis. Isn't that a problem?

There is a big difference between having a hypothesis that is ideal as far as you know, and knowing that your hypothesis is ideal. The former can (and usually will) make wild predictions. So, if you don't have enough data to validate a complex hypothesis, it really is better to have a simple one. The simple one will make better predictions.

Intuitively, the training data may be thought of as a set of equations,

$$\hat{\mathbf{y}}_1 = h(\mathbf{x}_1),$$

$$\hat{\mathbf{y}}_2 = h(\mathbf{x}_2),$$

$$\begin{aligned}\hat{\mathbf{y}}_3 &= h(\mathbf{x}_3),\\ &\vdots\\ \hat{\mathbf{y}}_n &= h(\mathbf{x}_n).\end{aligned}$$

Training a model essentially involves trying to solve for all of the parameter values in h, after plugging the data into the hypothesis. From algebra, we know that if you have n unknown values, then you need to have at least n equations, or else you cannot constrain any them to any particular value. So, if your hypothesis has, say, 214 parameters in it, you had better have at least 214 data points in your training set, or else there is no hope of identifying the right hypothesis.

For example, suppose you want to use a linear model,

$$y = mx + b.$$

This model has 2 parameters, m and b. Suppose you only have one data point, <2, 2>. When you train the model, you find $m = 1000$ and $b = -1998$. Is this a good hypothesis? Well, it perfectly fits all of your training data because
$2 = 1000 * 2 - 1998$. Does this hypothesis make good predictions? Technically you don't really know, because you only have one data point. In reality, however, you probably actually do know—it makes terrible predictions. How do you know that? Because the model is under-constrained. The odds of this hypothesis being worth anything, when its values really depend more on how you initialized your optimizer than on the data itself, are pathological. If you don't have a reason to know that a hypothesis is good, it is almost certainly not.

With most real-world problems there is some amount of noise in the training data, so we actually prefer our model to be over-constrained. Hence, we really want a lot more than n data points to constrain our model.

Now, let's examine how well polynomials scale. The number of coefficients in a polynomial of degree d with v variables is,

$$\frac{(d+v)!}{d!v!}.$$

Here is a table showing the number of coefficients for polynomials of a small degree with few variables:

Degree Vars	1	2	3	4	5
1	2	3	4	5	6
2	3	6	10	15	21
3	4	10	20	35	56
4	5	15	35	70	126
5	6	21	56	126	252
6	7	28	84	210	462
7	8	36	120	330	792
8	9	45	165	495	1287
9	10	55	220	715	2002
10	11	66	286	1001	3003

The values in this table are the same as those in Pascal's triangle, where each element is

$$\binom{d+v}{v}.$$

It can be observed that whenever the degree is greater than 1, the number of coefficients grows faster than the number of variables, and the rate of growth accelerates as the number of variables gets bigger. This is not a good property.

Multivariate polynomials tend to make terrible predictions. Why? Because they are too flexible. It takes a lot of parameters to specify how they bend in every dimension, and no one has enough data to constrain all those variables.

For general-purpose regression, we need a model that can handle the non-linearities that often occur in real-world data (like polynomials), but where the number of parameters scales linearly with respect to the number of variables (like linear model). Mutilayer perceptrons achieve both properties.

4.3 Multilayer Perceptrons

4.3.1 Logistic regression

One way to make a non-linear model is to wrap a linear model with some non-linear function. So, the equations for a linear model,

$$
\begin{aligned}
\hat{y}_1 &= m_{1,1}x_1 + m_{1,2}x_2 + m_{1,3}x_3 + b_1,\\
\hat{y}_2 &= m_{2,1}x_1 + m_{2,2}x_2 + m_{2,3}x_3 + b_2,\\
&\vdots
\end{aligned}
$$

become

$$
\begin{aligned}
\hat{y}_1 &= a(w_{1,1}x_1 + w_{1,2}x_2 + w_{1,3}x_3 + b_1),\\
\hat{y}_2 &= a(w_{2,1}x_1 + w_{2,2}x_2 + w_{2,3}x_3 + b_2),\\
&\vdots
\end{aligned}
$$

where a is some function that takes a scalar value as input and returns another scalar value for its output. We call this an *activation function*. (We also changed the m to a w, because these values are called “weights” when used with logistic regression.)

This is a strict generalization of linear models. If a is a linear function, then it is a linear model. If a is some nonlinear function, then the model is also nonlinear.

We can write these model equations in vector form:

$$
\begin{aligned}
\hat{y}_2 &= a(\mathbf{w}_1 \cdot \mathbf{x} + b_1),\\
\hat{y}_2 &= a(\mathbf{w}_2 \cdot \mathbf{x} + b_2).\\
&\vdots
\end{aligned}
$$

(The ^ over the y indicates that it is a predicted label, $\hat{\mathbf{y}} = h(\mathbf{x})$, rather than an observed label.)

We could also write the model equations in matrix form,

$$\hat{\mathbf{y}} = a(\mathbf{W}\mathbf{x} + \mathbf{b}).$$

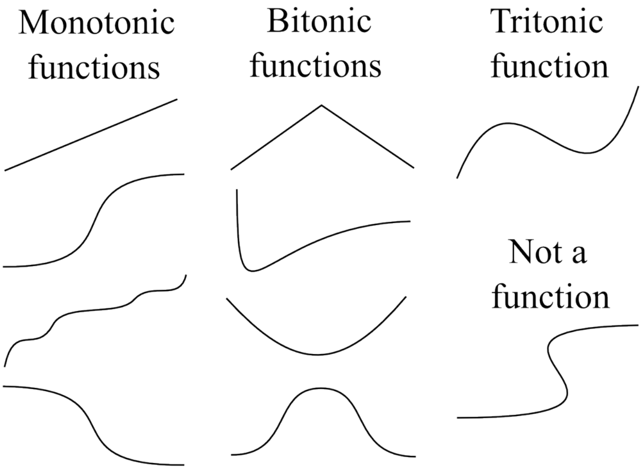
This is a slight abuse of notation, since the activation function, a, technically only accepts one scalar value as input. In order to use this notation, therefore, the reader needs to understand that a is intended to be applied individually to each element of its input vector to produce its output vector.

The $\mathbf{b}$ terms are called the *biases*, or the *bias weights*. They are important so that the model is not constrained to pass through the origin. If we add an additional element to $\mathbf{x}$ with a constant value of 1, then the biases can be combined with the weights. So, we can rewrite the equations as,

$$\begin{aligned}\hat{y}_1 &= a(\mathbf{w}_1 \cdot \mathbf{x}),\\ \hat{y}_2 &= a(\mathbf{w}_2 \cdot \mathbf{x}).\\ &\vdots\end{aligned}$$

When you see it written in this form, it is important to remember that there is still a bias weight—it has just been combined into the vector $\mathbf{w}$.

One advantage of using an activation function to warp a linear model into a nonlinear model, instead of using a polynomial of degree > 1, is that the activation function adds no additional coefficients (or parameters) beyond those of the linear model. Also, if the activation function is *monotonic*, then, like a linear model, the error surface is still guaranteed to be convex. So, logistic regression scales very well with dimensionality, and it is also easy to train. These are big advantages.



A monotonic function is a function whose output either never decreases or else never increases as the input is increased.

Although any function could be used as an activation function, it is common to use a *sigmoid function*. A sigmoid is a monotonic function shaped somewhat like the letter “S”. Sometimes sigmoid functions are called *squashing functions* because the input may be any value from $-\infty$ to ∞, but the output falls within a finite range. Hence, they “squash” the value. Because such functions are monotonic, the error surface is guaranteed to be convex, and because their output is constrained within a finite range, they are well-suited to be trained with gradient descent (which we discuss in the next section).

A common choice for a sigmoid-shaped activation function is the logistic function,

$$a(x) = \frac{1}{1 + e^{-x}}.$$

That's why it is called “logistic regression”. However, a slightly better choice is the *hyperbolic tangent*,

$$a(x) = \frac{2}{1 + e^{-2x}} - 1 = \tanh(x).$$

The hyperbolic tangent actually only differs from the logistic function by linear transformation. Since the weights perform a linear transformation on the inputs, they are functionally identical. However, for some tasks, the math turns out to be a little bit easier if you use the hyperbolic tangent, so it is generally preferred over the logistic function.

4.3.2 Gradient descent

Since the the error surface of logistic regression is guaranteed to be convex, no matter what data it is trying to fit, pretty-much any optimization technique is suitable for training them. Nevertheless, gradient descent is a good choice because it is commonly used for this purpose, and we will build upon it in the coming section on backpropagation.

To use gradient descent, we need to compute the gradient of the error surface. (That is, we need to compute the direction that goes up-hill, so we can step in the opposite direction to make the error smaller.) The gradient is the partial derivative of the average error with respect to weights,

$$\frac{\partial \text{MSE}}{\partial \mathbf{w}}.$$

This expression basically tells how MSE will change if we adjust the weights. It is the multi-dimensional generalization of slope. It is a vector of the same size as $\mathbf{w}$. Positive elements indicate that making the corresponding element in $\mathbf{w}$ bigger will make MSE bigger. Negative elements indicate that making the corresponding element in $\mathbf{w}$ bigger will make MSE smaller.

Another way to think about the gradient intuitively is to imagine a hyperplane that is tangent to the error surface at the point specified by the current values in $\mathbf{w}$. Each element in the gradient describes the slope of this hyperplane along the axis that represents one value in $\mathbf{w}$. The most efficient way to decrease MSE by adjusting the weights at this point is to adjust the weights by an amount inversely proportional to the values in the gradient. This corresponds to moving “directly downhill” on the error surface.

To use the gradient, we need to be able to evaluate it, so we need to get the ∂ terms out of the equation. Let's begin by expanding the expression:

$$\frac{\partial \text{MSE}}{\partial \mathbf{w}} = \frac{\partial \frac{1}{n} \sum_i (y_i - h(\mathbf{x}_i))^2}{\partial \mathbf{w}}.$$

Next, we want to move as many terms as possible outside of the ∂ expression. To do this, we will use the chain rule:

$$\frac{\partial \text{MSE}}{\partial \mathbf{w}} = \frac{\frac{2}{n} \sum_i (y_i - h(\mathbf{x}_i)) \partial (y_i - h(\mathbf{x}_i))}{\partial \mathbf{w}}.$$

This may look more complex, but the part inside the ∂ expression is a little bit simpler than before, and that is the part we are trying to get out of our equation. We can now drop the “y_i” inside the ∂ expression, because y_i does not depend on $\mathbf{w}$.

$$\frac{\partial \text{MSE}}{\partial \mathbf{w}} = \frac{-\frac{2}{n} \sum_i (y_i - h(\mathbf{x}_i)) \partial (h(\mathbf{x}_i))}{\partial \mathbf{w}}.$$

By contrast, we cannot drop the “$h(\mathbf{x}_i)$”, because it does depend on $\mathbf{w}$. So, let's expand the $h(\mathbf{x}_i)$ term inside the ∂ expression to see if we can simplify further:

$$= \frac{-\frac{2}{n}\sum_i (y_i - h(\mathbf{x}_i))\partial(a(\mathbf{w}\cdot\mathbf{x}_i))}{\partial\mathbf{w}}.$$

We can now use the chain rule to peel off another layer of what is inside the ∂ expression. Specifically, we can get the a out of there:

$$= \frac{-\frac{2}{n}\sum_i (y_i - h(\mathbf{x}_i))a'(\mathbf{w}\cdot\mathbf{x}_i)\partial(\mathbf{w}\cdot\mathbf{x}_i)}{\partial\mathbf{w}}.$$

(In this expression, a' is the derivative of a. So, our activation function had better be differentiable, or else that could stop us from being able to evaluate this equation. Fortunately, there are many differentiable activation functions to choose from. We will visit this topic in greater detail later.) Now, we have finally reached a point where we can work out that ∂ expression!

$$\frac{\partial\text{MSE}}{\partial\mathbf{w}} = -\frac{2}{n}\sum_i (y_i - h(\mathbf{x}_i))a'(\mathbf{w}\cdot\mathbf{x}_i)\mathbf{x}_i.$$

Just to finish it off, let's expand the other $h(\mathbf{x}_i)$ too:

$$\frac{\partial\text{MSE}}{\partial\mathbf{w}} = -\frac{2}{n}\sum_i (y_i - a(\mathbf{w}\cdot\mathbf{x}_i))a'(\mathbf{w}\cdot\mathbf{x}_i)\mathbf{x}_i.$$

This equation evaluates to a vector of the same size as $\mathbf{w}$. This vector tells us how changing $\mathbf{w}$ will impact MSE. Since the gradient points uphill, we want to step in the opposite direction. This is done by updating $\mathbf{w}$, like this:

$$\mathbf{w} = \mathbf{w} - \eta\frac{\partial\text{MSE}}{\partial\mathbf{w}}.$$

η is the lowercase Greek letter “eta”. We use it to represent the *step size*, also called the *learning rate*. (Often, in machine learning literature, the “2” is dropped in these equations, because it is folded into the value of η .)

4.3.2.1 Empirical gradient approximation

Imagine that you were standing blindfolded on a sloping surface. How could you determine the gradient of the surface at your current location? Well, you could reach out with your foot and feel around. It probably wouldn't take very long for you to determine which way went up, and which way went down.

Another way to obtain the gradient is to measure it empirically. Let δ be some small value. (This is the distance that you extend our toe to feel the gradient around you.) Let c be MSE with the current weights, and let d be MSE when one of the weights, w_j, is temporarily incremented by a small amount, δ. Then,

$$\frac{\partial \text{MSE}}{\partial w_j} \approx \frac{d - c}{\delta}.$$

Slope can be described as “rise over run”. In this equation, the numerator is the rise in MSE, and the denominator is the run.

Understanding how to approximate the gradient empirically is a good way to develop an intuition for the gradient. Unfortunately, it is not a very efficient way to approximate it. Suppose your model contains 1000 weights. To measure the entire gradient empirically, you would have to evaluate MSE 1001 times. That requires a lot more computation than just evaluating the equation we derived for the gradient. So, doing that math saved us a lot of unnecessary computation.

If you want to verify that we actually did the math correctly, a good check is to pick an arbitrary model and make sure the gradient we computed in closed form gives approximately the same results as when you measure the gradient empirically. (And, by the way, they do return approximately the same values—I tested it.)

4.3.2.2 Stochastic gradient descent

I once worked with a company that was so concerned about building a high-quality product that they held design meetings at excessively frequent intervals. Everyone in the company was invited to express their preferences before any design decisions were made. So much time was spent planning that little time remained for development, and since progress was so slow, each meeting had little more context to work with than the previous one. Unsurprisingly, funding ran out before the product was completed. It is possible to be too deliberate.

Instead of consulting with every sample in the training data before taking a single step, it turns out that it is significantly more efficient if we present the training samples in random order, and take a step after each one.

So, instead of computing the gradient for the mean squared error of the whole training set,

$$\frac{\partial \mathrm{MSE}}{\partial \mathbf{w}} = -\frac{2}{n} \sum_i (y_i - a(\mathbf{w} \cdot \mathbf{x}_i)) a'(\mathbf{w} \cdot \mathbf{x}_i) \mathbf{x}_i,$$

we only need to compute the gradient for the squared error of a single sample, $\langle \mathbf{x}_i, y_i \rangle$, as

$$\frac{\partial \mathrm{SE}}{\partial \mathbf{w}} = -2(y_i - a(\mathbf{w} \cdot \mathbf{x}_i)) a'(\mathbf{w} \cdot \mathbf{x}_i) \mathbf{x}_i,$$

and we update the weights after each sample is presented.

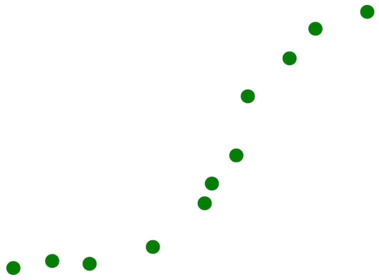
If there is noise in the training data, then some samples may pull the weights in one direction, and other samples pull the weights in another direction. Aren't such tug-of-wars counter-productive? Isn't that a problem? Well, not really. Until the weights arrive at an optimum, the tug-of-war will never be evenly matched in all directions. So, even though progress may go back and forth in some directions, it will yet advance in directions where there is some component of agreement. Further, that advancement may change the balance in directions that previously only went back and forth. So, even though the path it follows may wind or zig-zag, it benefits immediately from every refinement, instead of waiting until all the refinements have been aggregated.

When the gradient is computed over the entire training set, this is called *batch gradient descent*. When the gradient is computed and the weights updated for each sample, we call it *on-line gradient descent*, or *stochastic gradient descent*. (The term “on-line” predates the Internet. In machine learning, it refers to the ability to steadily refine a model, and to potentially use the model before it is fully refined. The term “stochastic” means random. In this case, it refers to the random order in which the patterns are presented.)

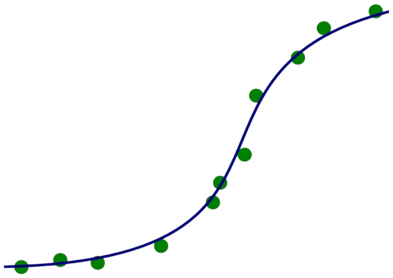
4.3.3 Multiple layers

Linear regression can be thought of as fitting a hyperplane to some training data. Logistic regression, sometimes called a *layer of sigmoid units,* or *perceptrons,* can be thought of as fitting an activation function to the training data. For example, Here is a plot of the logistic function:

So, if your training data looks like this,

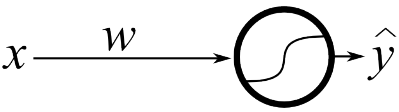


then a perceptron with a logistic activation function can fit to the data quite nicely:



Perceptrons can also fit linear data quite well (by using a nearly-linear segment of the activation function to fit the data). So, perceptrons are slightly more capable than linear models. If your data does not resemble some portion of the activation function, however, then a perceptron may not be able to fit the data well.

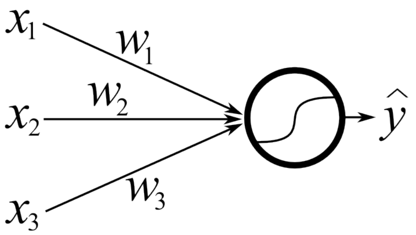
A diagram of this perceptron might look like this:



x is the input value. This input value is multiplied by w, and the bias, b, is added. (The bias is implicit, so it is not depicted in the diagram.) This value is then passed through the activation function to predict y. So, this diagram represents the equation,

$$\hat{y} = a(wx + b).$$

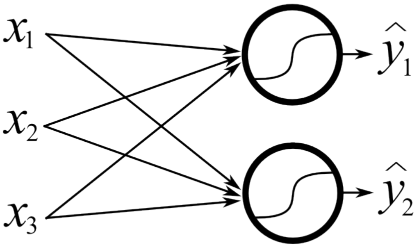
We can also draw a perceptron with multiple inputs:



When there are multiple arrows feeding into a circle, the values they represent are all summed together before the activation function is applied. So, this diagram represents the equation,

$$\hat{y} = a(w_1x_1 + w_2x_2 + w_3x_3 + b).$$

There can also be multiple outputs. This is still a single-layer perceptron:



This diagram represents the equation,

$$\hat{\mathbf{y}} = a(\mathbf{W}\mathbf{x} + \mathbf{b}),$$

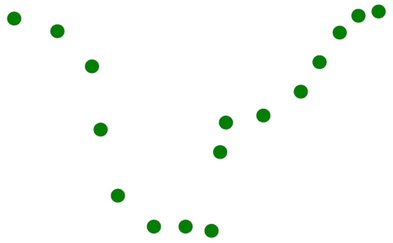
where $\mathbf{x}$ is a 3-element column vector, $\mathbf{W}$ is a 2×3 matrix of weights, $\mathbf{b}$ is a 2-element column vector (because each activation function has one implicit bias value), and $\hat{\mathbf{y}}$ is a 2-element column vector. (Note that the activation function, a, is applied element-wise, not to the whole vector.)

Some implementations might use a 2×4 matrix to store both $\mathbf{W}$ and $\mathbf{b}$, and add an extra element to $\mathbf{x}$ with a value of 1. In this case, the equation would be just,

$$\hat{\mathbf{y}} = a(\mathbf{W}\mathbf{x}).$$

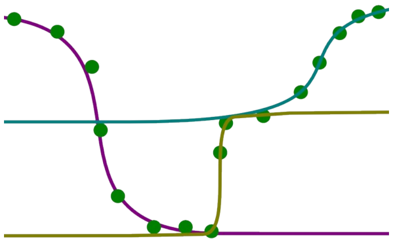
These are just two different ways to express the same thing.

Single-layer perceptrons are great models as long as your data has a shape that matches some part of the activation function, but, what if your data looks more like this?

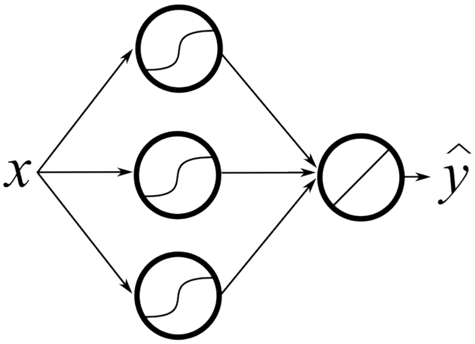


No part of any monotonic activation function will be sufficient to fit this data, because it clearly follows a bitonic trend. Perhaps with a lot of trial-and-error, we could eventually find a bitonic activation function that fits this data, but that would require a lot of human effort, and we would prefer an automated approach.

Although one monotonic activation function cannot fit this data well, we could fit it quite nicely by combining 3 perceptrons in a piecemeal manner, like this:



Here is a diagram of a model that combines 3 perceptrons in this manner:



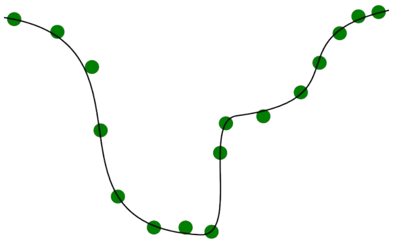
Notice that there is just one input and one output in this model, but it uses 4 total perceptrons: 3 of them to fit the data, and 1 to combine them into a single curve. This is a multi-layer perceptron (MLP). Each perceptron in an MLP is called a *node* or a *unit*. We used the identity function in the output unit, so that it will just combine the other perceptrons without adding any distortion. It is slightly more common to use the same activation function in all units, but I think using a linear activation function on the output unit is a little easier to understand intuitively. The three sigmoid units constitute a *hidden layer*. This is a layer between the inputs and the outputs. There is also an *output layer*. In this case, the output layer contains only one linear unit. Thus, the model as a whole is a 2-layer network of perceptrons. (Some people refer to the inputs as another "layer". They would call this a 3-layer network. I do not count the inputs as a layer, because they have no activation function, and no weights.)

So, the mathematical equation for this whole model would be,

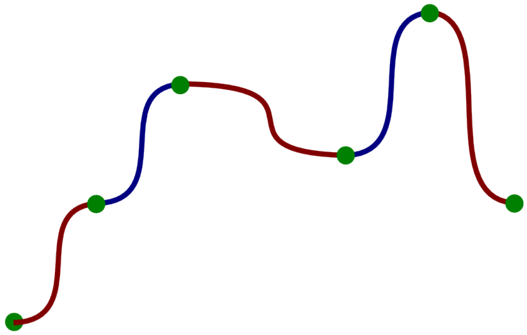
$$\begin{aligned}\hat{y} = {} & w_4 a(w_1 x + b_1) + w_5 a(w_2 x + b_2) + \\ & w_6 a(w_3 x + b_3) + b_4.\end{aligned}$$

A good exercise would be to enter this equation into a function-plotting tool. (There are many convenient web sites that do this.) Try adjusting the weights to see how they affect the plot.

b_1, b_2, and b_3 will shift each of the sigmoid-shaped regions left or right. b_4 will shift the entire curve up or down. w_1, w_2, and w_3 will adjust the steepness of each sigmoid-shaped region. w_4, w_5, and w_6 will adjust the vertical scale of each sigmoid-shaped region. Altogether, these values provide all the degrees of freedom necessary to arbitrarily combine three sigmoid-units into a single curve. See if you can find weights that approximate the curve for this data:

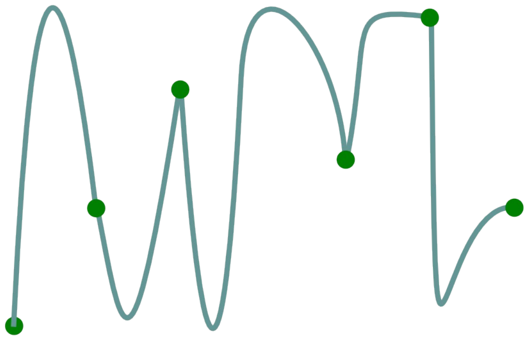


Now, someone might argue, *hey, you just designed that data to fit easily with a combination of sigmoids.* Oh, Did I? Well, then I challenge you to produce some data that a combination of sigmoid units cannot approximate with high precision. In the worst-possible case involving n points, I can simply connect each of them with $n - 1$ sigmoid units, plus one more to combine them all into a single smooth and continuous curve. Are you impressed? (Don't be—it's not very impressive. It's just a cheap trick.)



We can also add more inputs and outputs as necessary to fit a function with any number of inputs and any number of outputs. Thus, a 2-layer perceptron is an *universal function approximator*. An universal function approximator, sometimes called an *arbitrary function approximator*, is any model that can predict any training data with arbitrary precision (assuming there are no samples that assign conflicting labels to the same pattern). *k*-NN is an arbitrary function approximator, because it simply stores the training data. Decision trees are arbitrary function approximators because the tree can divide until the labels are perfectly homogeneous. Multivariate polynomials are, because you can increase the degree until they achieve arbitrary flexibility. One-layer perceptrons are not arbitrary function approximators, but two-layer perceptrons are, because they can combine more sigmoid units until every training point is precisely fitted.

So, why do we want to use a universal function approximator in machine learning applications? (Be careful, it's a trick question.) In machine learning, the goal is usually about generalizing, or making accurate predictions. This is not necessarily the same thing as fitting the training data. Flexibility in the model is only good as far as you need it, and thereafter it is a bad thing. To state it pessimistically, the property of being a universal function approximator is really a guarantee that no matter how simple or complex the training data may be, the model can represent a hypothesis that perfectly fits all the training data while still making horrible predictions for any pattern not in the training data.



So, it may be nice to know that our models can be made to represent any problem that we might encounter, but we cannot really expect any hypothesis to make good predictions unless we also find a way to keep it simple.

The real reason MLPs make good models in machine learning, is because they can often fit complex training data with far fewer than n hidden units. Additionally, you can even add multiple hidden layers. These give the network significantly more representational power in ways that are not yet fully-understood. Only a few extra units in an additional hidden layer can often compensate for many extra units in just one hidden layer. One theory is that these additional hidden units somehow exploit redundant patterns in the data to reuse the units in other layers to fit different regions of the data.

4.3.4 Conditions for successful regression

In general, regression will be successful if the following conditions are met:

1. The model is capable of fitting the problem,
2. Enough training data is available to constrain the model to fit the problem, and
3. Optimization of model parameters succeeds.

If the model is a universal function approximator, then condition 1 is satisfied. After all, a universal function approximator can fit any function, no matter how complex.

Like neural networks, multivariate polynomials are also universal function approximators. So, why are multivariate polynomials rarely used for regression? Because they exacerbate condition 2. As dimensionality increases, multivariate polynomials have almost exponentially more parameters, which means exponentially more data is needed to constrain them. By contrast, neural networks achieve universal function approximation without scaling badly in their number of parameters as dimensionality increases. So with neural networks, “enough” data is much more likely to be an amount that is practical to obtain. With multivariate polynomials, it is often nearly impossible to obtain “enough” data.

Neural networks are not the only model that scales well in the number of parameters as dimensionality increases. Linear models can scale to handle very large dimensionality, without incurring the cost of too many parameters. However, linear models are not universal function approximators. Linear models are great for condition 2, but often fail at condition 1.

Neural networks may be characterized as a having the scalability of linear models with the universal approximation capabilities of multivariate polynomials. They achieve both properties by adding a parameter-free nonlinear activation function on top of a linear model, and then adding multiple layers.

Condition 3 can be solved in theory by using grid search, or some other optimization technique that systematically searches the entire parameter space of the model. Of course, such solutions are not very practical. One might wonder, will we ever find an optimization technique that is both practical and guarantees to find the local optimum? Since global optimization can be trivially shown to find the solutions to many NP-complete problems, that seems unlikely. Consequently, a practical solution to condition 3 may be as good as can ever be achieved. With neural networks, stochastic gradient descent tends to work pretty well at finding good solutions to many problems (although it is, admittedly, rather computationally expensive).

So, altogether, neural networks strongly satisfy two of the conditions for successful regression, and weakly satisfy the third. Some refinements to neural networks, such as convolutional layers (which we cover later), further strengthen their capabilities for satisfying criterion 2. Some training methods, such as unsupervised pretraining and GPGPU parallelization, strengthen their case for criterion 3. Overall, it appears unlikely that some other model will be replacing neural networks anytime soon as the best general-purpose model for regression problems.

4.3.5 The hyperbolic tangent

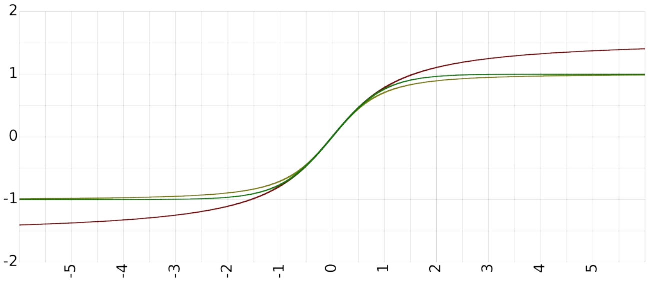
There are many possible functions that one could use for an activation function. Some possible choices include:

$$a(x) = \frac{x}{\sqrt{1+x^2}},$$

$$a(x) = \operatorname{atan}(x), \text{ and}$$

$$a(x) = \tanh(x).$$

Here is a plot of these 3 functions together (respectively in red, yellow, and green).



These functions all have the following properties that are desirable in an activation function:

1. They are sigmoid-shaped.
2. They are monotonic.
3. They can be computed efficiently.
4. They pass through the origin.
5. They are differentiable.
6. The derivative at 0 is 1.

Of these activation functions, my favorite is the hyperbolic tangent. In addition to these properties, it also has many other elegant properties.

It can be expressed in a couple different forms,

$$a(x) = \frac{2}{1 + e^{-2x}} - 1 = \tanh(x)$$

It ranges between -1 and 1, which is easy to remember.

It is invertible,

$$\begin{aligned} a^{-1}(y) &= \frac{1}{2}(\log_e(1 + y) - \log_e(1 - y)) \\ &= \mathrm{atanh}(y) \end{aligned}$$

It is differentiable,

$$a'(x) = \left(\frac{2}{e^x + e^{-x}}\right)^2 = \mathrm{sech}^2(x),$$

and this derivative can even be expressed in terms of the activation function itself,

$$a'(x) = 1 - \tanh^2(x) = 1 - a(x)^2,$$

which property can be used to simplify implementations of the backpropagation algorithm somewhat.

Its derivative is differentiable,

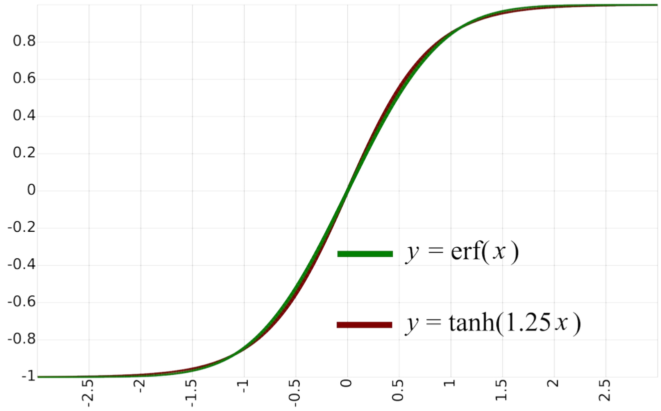
$$a''(x) = (2\tanh^2(x) - 2) * \tanh(x),$$

and it is differentiable too, but I don't feel like writing out the formula here.

The hyperbolic tangent can be integrated,

$$\int a(x) = \log_e(\cosh(x)).$$

The shape of the hyperbolic tangent function is very similar to that of the cumulative density function (CDF) of the Normal distribution.



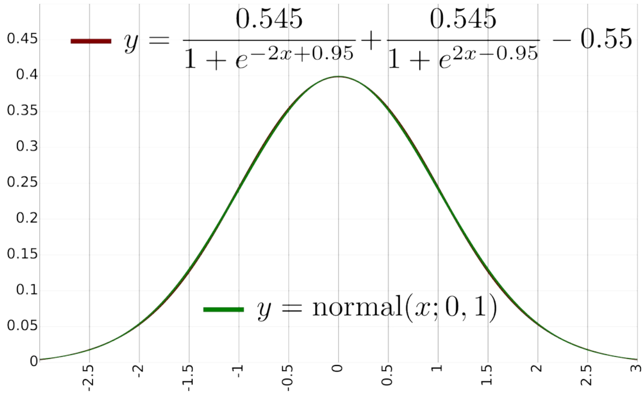
The CDF of the normal distribution is

$$\Phi(x) = \frac{1}{2}\left(\text{erf}\left(\frac{x}{\sqrt{2}}\right) + 1\right),$$

where $\text{erf}(x)$ is called the “error function”. The error function is a very important function because it occurs in nature, but it is somewhat difficult to work with because it can be estimated, but cannot be computed exactly in closed form (because its closed-form representation contains an integral that does not resolve to any elementary functions),

$$\text{erf}(x) = \frac{1}{\sqrt{\pi}} \int_{-x}^{x} e^{-t^2} dt.$$

Also, the sum of just two parameterized hyperbolic tangent functions can approximate the Normal distribution very closely.



(If you cannot see the red curve, it is because the green curve is mostly covering it up.)

Despite all of these nice properties, there are some good reasons to use other activation functions. [todo: discuss]

4.3.6 Backpropagation

Like single-layer perceptrons, multi-layer perceptrons can also be trained with gradient descent. In this section, we define the details about how this is done.

On-line training is performed by presenting training samples one at-a-time in random order. Each time a training sample, i, is presented, the following 3 high-level steps are performed to refine the weights of the MLP:

1. Use forward-propagation to predict labels.
2. Compute a “blame term” for each unit in the network.
3. Update all the weights.

Step 1 (forward-propagation) is performed by feeding the input vector, $\mathbf{x}_i$, into the network to compute a predicted label vector, $\hat{\mathbf{y}}_i$. To describe this step, we define the following terms:

Let $w_{t,s}$ be the weight that feeds the output of unit s into unit t.

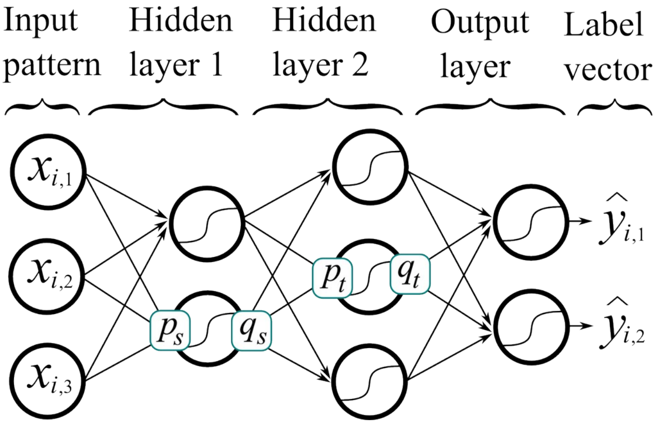
Let $w_{t,1}$ be the bias weight that feeds into unit t.

Let p_t be the net input into unit t.

Let a_t be the activation function on unit t.

Let q_t be the output of unit t.

For reference, here is an example diagram of an MLP:



This example has 3 layers. (The input pattern is not usually counted as a layer.) For this example, I arbitrarily picked one unit in hidden layer 1 to be called unit s, and another unit in hidden layer 2 to be unit t. Unit t is one of the 3 units into which s feeds its output. The arrow between unit s and unit t could be labeled $w_{t,s}$, but that made the diagram look too busy, so I removed it. Also, the activation function in unit s could be labeled a_s , and the activation function in unit t could be labeled a_t.

In each layer, $\mathbf{p}$ (a.k.a "the net") is computed with the equation,

$$p_t = \left(\sum_s w_{t,s} q_s\right) + w_{t,1}$$

The summation in this equation sums over each unit, s, that feeds into unit t.

Then, the values in $\mathbf{q}$ are computed with the equation,

$$q_t = a_t(p_t).$$

At the start of this step, where the input pattern feeds into the very first hidden layer, $\mathbf{q}$ is the same as $\mathbf{x}_i$. At the end of this step, where the output layer computes the predicted label vector, $\mathbf{q}$ is the same as $\hat{\mathbf{y}}_i$.

<u>**Step 2**</u> computes a value that represents the extent to which each output

unit is responsible for the prediction error. I call this the *blame term*. (Sometimes it is called an “error term”, but this can be confusing terminology because it is not the same as “error”, which we defined as $y - \hat{y}$. Therefore, I will call it “blame”.)

Let δ_u be the blame term for unit u.

We can compute the blame term for each output unit as,

$$\delta_u = (y_{i,u} - \hat{y}_{i,u})a'_u(p_u),$$

which is the same as

$$\delta_u = (y_{i,u} - q_u)a'_u(p_u).$$

Now we need to compute the blame term for all of the hidden units in the network. This step is called *backpropagation*. (The other steps of gradient descent were known long before backpropagation was discovered. Consequently, this step was historically significant because it enabled gradient descent to be used with multi-layer perceptrons. Often in machine learning literature, the term “backpropagation” is used to refer to the entire process of refining weights by gradient descent. Although this usage is confusing, it is so common that it is not considered to be a mistake. For clarity, I only use the term “backpropagation” to refer to the step of computing the “blame” for each unit, but you will likely encounter other literature that uses the term to refer to the entire 3-step process.)

Backpropagation is performed by recursively applying the formula,

$$\delta_s = \sum_t w_{t,s}\delta_t a'_s(p_s),$$

until the blame term has been computed for every unit in the network. The summation in this equation sums over each unit, t, into which unit s feeds.

The reason for computing all these blame terms is that they make the gradient very easy to compute.

<u>Step 3</u> updates all of the weights in the network by stepping in the opposite direction of the gradient. The gradient for any weight can be computed as,

$$\frac{\partial \mathrm{SE}}{\partial w_{t,s}} = -\delta_t q_s.$$

The gradient for any bias weight is simply,

$$\frac{\partial \mathrm{SE}}{\partial w_{t,1}} = -\delta_t.$$

The gradient for the inputs is,

$$\frac{\partial \mathrm{SE}}{\partial x_{i,u}} = -\sum_j w_{j,u}\delta_j.$$

The summation in this equation sums over each unit, j, into which $x_{i,u}$ feeds. This equation is not used in applications where the inputs are considered to be fixed and only the weights are updated. There are, however, applications where the inputs are also adjusted. We will discuss those applications in other sections. In this section, this equation is not used at all.

So, the weights are updated by taking a small step in the opposite direction of the gradient,

$$w_{t,s} = w_{t,s} - \eta\frac{\partial \mathrm{SE}}{\partial w_{t,s}},$$

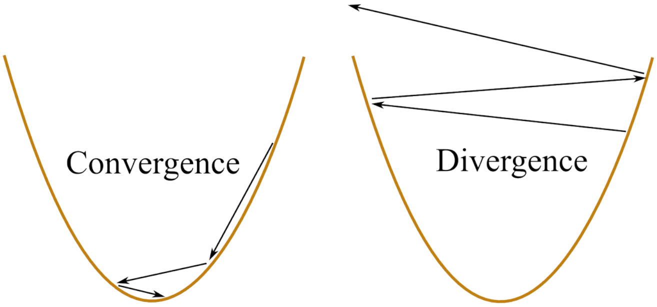
and

$$w_{t,1} = w_{t,1} - \eta\frac{\partial \mathrm{SE}}{\partial w_{t,1}}.$$

In many implementations, steps 3 and 4 are combined, such that both steps are performed in a single backward pass over the network units. After the weights are updated, the next training sample in random order is presented, and the 4 steps are repeated until convergence is detected.

4.3.6.1 Setting the learning rate

In general, if the learning rate is very small, then it will take many steps to get to the optimum. However, if the learning rate is too big, it is possible to diverge instead of converge.



Even if η is constant, the actual size of the step we take still depends on the gradient, which changes with each step. This generally has the very nice effect of causing it to take big steps when far away from the optimum, and smaller steps as it approaches an optimum. If η is too big, however, this can actually exacerbate divergence, because each step moves to a position where the gradient is even steeper, making the next step even bigger. Hence, it is much better to choose a learning rate that is too small than to choose one that is too big.

Suppose we use the hyperbolic tangent function for the activation function of our MLP. The output of this MLP can only range from -1 to 1. Presumably, the training data has also been normalized so that its values fall within this range. (If not, then our MLP is going to fail miserably to fit with this data.) If both the target labels and the predicted labels always fall between -1 and 1, then the worst possible MSE is likewise bounded. This makes it unlikely for the gradient to ever grow out-of-control and lead to divergence.

When a linear activation function is used on the output layer, the learning rate must be carefully chosen. Sometimes, it must be very small in order to avoid divergence. When a sigmoid-shaped activation function is used, however, gradient descent is much more robust to the learning rate. For this reason, it is common to use a sigmoid-shaped activation function on every unit in an MLP.

With many simple problems, the value $\eta = 0.1$ generally works well. With very difficult problems, the value $\eta = 0.01$ may work better. Problems that require a smaller learning rate than that are uncommon, but they certainly exist.

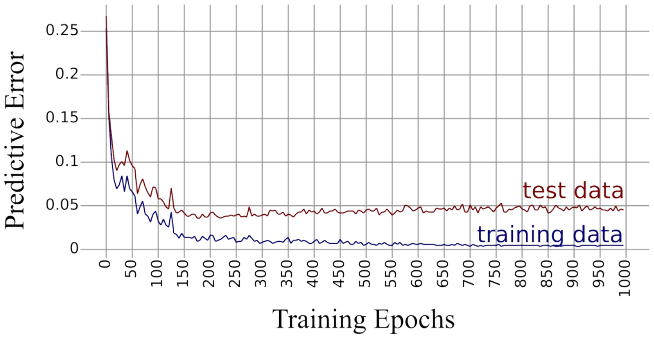
Sometimes, a good option is to decay the learning rate. For example, you might start out with a value, such as $\eta = 0.1$, and slowly reduce the learning rate until it reaches a value, such as $\eta = 0.01$. In theory, the gradient automatically scales the step size for you, but in practice, I find that I get slightly better results when I use a decaying learning rate.

A fairly appealing idea is to attempt to dynamically adjust the learning rate. For example, consider the following intuitive approach: One could train two copies of the MLP in parallel, one with a learning rate somewhat larger than the current value, and the other with learning rate somewhat smaller than the current value. After each pass through the training data, the two copies are evaluated to determine which one achieved greater improvement. The learning rate is then adjusted to match that of the winner. The winner then copies its weights over the loser, so they both have the same starting point for the next pass through the training data. Unfortunately, in practice, this approach does not usually work very well. It tends to prioritize immediate improvements over long-term improvements, which causes the learning rate to become prematurely small. Perhaps, this may be due to a preference of not riding up the walls of narrow alleys as it rolls downhill. In practice, using a manual decay rate seems to work better than simple dynamic tuning methods. Complex dynamic tuning methods have been developed, but who wants to deal with one of those?

4.3.6.2 Detecting convergence

The longer you train an MLP, the better it will become at predicting the training data. It might seem natural to conclude, therefore, that an MLP should be trained for as long as possible. This, however, is erroneous reasoning. In machine learning, the goal is not typically to predict the training data well. Rather, it is to generalize well.

The following chart compares error measured on the training against error measured on a test set as a neural network is trained. (Specifically, this is the “cars” dataset, and we trained an MLP with one hidden layer of 16 nodes. We used a learning rate of 0.03.)



It is easily observed that the MLP does better with the training data that the test data (which it has never seen before). After 2000 training epochs, it mis-classifies fewer than 1% of instances in the training data, whereas it mis-classifies almost 4% of instances in the test data. The difference between these curves is due to overfitting to the training data.

Notice that both lines are somewhat jittery, as opposed to being smooth. Such is almost always the case with test data, because it is not exactly the same as the data on which the MLP is being trained. There are three reasons why the training data line is also jittery in this graph: 1. We used on-line training instead of batch training, 2. We measured predictive accuracy (which differs slightly from the sum-squared-error that backpropagation optimizes), and 3. Our learning rate was somewhat larger than infinitesimal.

Despite the non-smoothness of the curves, it can still be observed that the training error generally decreases with additional training epochs. Interestingly, however, the test error improves to a certain point, and then actually starts to get a little worse. This behavior is typical with most datasets. So, how do we know when to stop training? Since the goal is to generalize well, the best time to stop training is when the test error is as low as it will ever get. (In this case, it is somewhere around 650 epochs.) Unfortunately, since the measurement is not smooth, this point can be tricky to detect. Sometimes, test error will even plateau for a while (as can be observed between 200 and 400 epochs) before it begins to improve again.

There is no way to know for sure when to stop if you are using all of the available training data to train your MLP. It is necessary, therefore, to divide this data into two portions: one portion to train the MLP, and one portion to evaluate it. This portion is often called a *validation set*.

One good method is to make a copy of the weights whenever the test accuracy is better than has ever been seen before. After you have trained for a long time, restore the best weights ever found. Unfortunately, it can be expensive to copy the weights at frequent intervals. Sometimes, it is more efficient to simply store the epoch number when the weights are best, and then train a second time until that number of epochs is reached.

How do you know when you have trained for a sufficiently long time the first time? A good approach is to measure accuracy over a window of epochs. For example, you might stop if test accuracy has improved by less than 0.1% over a window of 200 epochs. Such criteria is guaranteed to eventually stop. In this case, it would stop after about 900 epochs, which reasonable since the best epoch occurred before this point. Larger windows and larger improvement thresholds will be more robust against jittery measurements, but smaller windows and smaller improvement thresholds will be more effective at preventing superfluous training.

No matter how good your stopping criteria may be, however, it won't be sufficient to prevent all overfit by itself. As we examine these two curves, we can see that they actually began to separate from each other after as few as 50 epochs, and that would definitely be too soon to stop. So, what do we do about that?

The best solution for preventing overfit is to use more training data. Unfortunately, this solution is not always practical, because more data is not always available. Another option is to reduce the number of hidden nodes. In this case, we probably used more hidden nodes than was really necessary to model this data. Another option is to use regularization. (We will discuss this topic more in a later section.) If we take the time to figure out the best number of hidden nodes, and the best regularization parameters, then the stopping criteria will have less of an impact on generalization accuracy. But, if we really want to generalize well, we might as well use every trick we can find to improve it.

4.3.6.3 Deriving backpropagation

(Before reading this section, it might be a good idea to take a look at the appendix that reviews the chain rule.)

So, where did all those equations come from that define gradient descent with an MLP? And most-importantly, where did the equation for the backpropagation step come from?

Several people independently invented backpropagation. Arthur E. Bryson and Yu-Chi Ho appear to have been first. They described it as an optimization technique in 1969. Paul Werbos discovered it for an application in economics. David E. Rummelhart is often credited for bringing it into mainstream use with neural networks. Others, including Geoffrey E. Hinton, and Ronald J. Williams were also influential in that work.

Let $h(\mathbf{x}_i)$ refer to the output of the MLP when $\mathbf{x}_i$ is fed into it. We can decompose the MLP into many functions,

$$\hat{\mathbf{y}}_i = h(\mathbf{x}_i),$$
$$h(\mathbf{x}_i) = f_6(f_5(f_4(f_3(f_2(f_1(\mathbf{x}_i)))))).$$

Each layer in the MLP becomes two functions. In this case, all of the odd-numbered functions are linear equations,

$$f_1(\mathbf{x}) = \mathbf{W}\mathbf{x},$$

and all of the even-numbered functions are some activation function,

$$f_2(\mathbf{x}) = a(\mathbf{x}).$$

Each of these functions is easy to differentiate:

$$f_1'(\mathbf{x}) = \mathbf{W},$$
$$f_2'(\mathbf{x}) = a'(\mathbf{x}).$$

Admittedly, we didn't really do anything with f_2, because we don't know what activation function will be used. But, the point is, we can plug in the derivative of a whenever we need it. So, now, all we need to do is use the chain rule to metaphorically peel the onion, and we will obtain the derivative of our MLP.

At this point, you'd probably be better-off getting out some paper and doing it yourself, but I'll walk through the process, anyway. For simplicity, we will do it for the case of a one-dimensional label vector. I will try to do it in relatively small steps, and attempt to err on the side of providing too much explanation, rather than too little. If it looks scary because there are so many equations, then, as I said, you'd probably be better-off getting out some paper and doing it yourself.

We start by expressing the partial derivative of the squared error with respect to an arbitrary weight,

$$\frac{\partial E}{\partial w_{t,s}} = \frac{\partial ||\mathbf{y}_i - \hat{\mathbf{y}}_i||^2}{\partial w_{t,s}},$$

and then we need to work the “∂” out of the expression. The only reason this wasn't discovered for multi-layer perceptrons earlier is because the equations get a little-bit scarier, and no one earlier was brave enough to work it out. In hindsight, however, things are generally much easier. So let's derive gradient descent for a multi-layer perceptron.

We start by applying the chain rule to obtain,

$$\frac{\partial E}{\partial w_{t,s}} = \frac{2(\mathbf{y}_i - \hat{\mathbf{y}}_i)\partial(\mathbf{y}_i - \hat{\mathbf{y}}_i)}{\partial w_{t,s}}.$$

(If that last step was unclear, see the derivation of Ordinary Least Squares. It starts off in a similar manner, but we describe it in a bit more detail there.) Since changing the weights does not affect the target value, we can drop $\mathbf{y}_i$ from the ∂ term:

$$\frac{\partial E}{\partial w_{t,s}} = \frac{-2(\mathbf{y}_i - \hat{\mathbf{y}}_i)\partial\hat{\mathbf{y}}_i}{\partial w_{t,s}}.$$

Our goal is to get rid of the ∂ term, because that is what is stopping us from evaluating it, so next we will expand the “$\hat{\mathbf{y}}$” in that term. If we were to expand it as far as possible, it would be a pretty big and scary-looking equation. This is probably why no one successfully worked it out before 1969. But, if we just do it one layer at-a-time, it turns out to not be so bad. We get

$$\frac{\partial \mathrm{SE}}{\partial w_{t,s}} = \frac{-2(\mathbf{y}_i - \hat{\mathbf{y}}_i)\partial a(\mathbf{p}_l)}{\partial w_{t,s}},$$

where l is the number of layers, and $\mathbf{p}_l$ is the vector of net input values into the output layer of activation functions. Note that the activation function only accepts a single scalar value as its input, so $a(\mathbf{p}_l)$ is intended to mean that the activation function is applied element-wise, rather than to the vector as a whole.

Now, we can apply the chain rule to peel that activation function out of the ∂ expression,

$$\frac{\partial \mathrm{SE}}{\partial w_{t,s}} = \frac{-2(\mathbf{y}_i - \hat{\mathbf{y}}_i)a'(\mathbf{p}_l)\partial \mathbf{p}_l}{\partial w_{t,s}}.$$

Now, we need to fork our derivation. So remember this point, because we will come back to it in a little while. If our model is a single-layer perceptron ($l = 1$), then we can expand $\mathbf{p}_l$ to get

$$\frac{\partial \mathrm{SE}}{\partial w_{t,s}} = \frac{-2(\mathbf{y}_i - \hat{\mathbf{y}}_i)a'(\mathbf{p}_1)\partial \mathbf{W}\mathbf{x}_i}{\partial w_{t,s}}.$$

Now, we can work out the ∂ expression,

$$\frac{\partial \mathrm{SE}}{\partial w_{t,s}} = -2(\mathbf{y}_i - \hat{\mathbf{y}}_i)a'(\mathbf{p}_1)x_{i,s}.$$

(To see why “$\mathbf{W}\mathbf{x}_i$” reduces to “$x_{i,s}$”, you can expand “$\mathbf{W}\mathbf{x}_i$” into a bunch of products. The only term in this expansion that involves $w_{t,s}$ is “$w_{t,s}x_{i,s}$”.)

This equation is precisely what we get if we combine the equations from steps 2 and 4 for doing gradient descent with an MLP. Therefore, we have shown that those equations are correct for the case of only one layer.

We still need to show the case where there is more than one layer, so let us return to the point where we forked our derivation. For convenience, we will copy the equation from that point here:

$$\frac{\partial \mathrm{SE}}{\partial w_{t,s}} = \frac{-2(\mathbf{y}_i - \hat{\mathbf{y}}_i)a'(\mathbf{p}_l)\partial \mathbf{p}_l}{\partial w_{t,s}}.$$

Since there is more than one layer, expanding $\mathbf{p}_l$ gives

$$\frac{\partial \mathrm{SE}}{\partial w_{t,s}} = \frac{-2(\mathbf{y}_i - \hat{\mathbf{y}}_i)a'(\mathbf{p}_l)\partial \mathbf{W}_l\mathbf{q}_{l-1}}{\partial w_{t,s}},$$

where $\mathbf{W}_l$ is the weights of the last layer, and $\mathbf{q}_{l-1}$ is the output (or activation) of the layer that feeds into it.

If $w_{t,s}$ is in $\mathbf{W}_l$, then we can work out the ∂ term right now (and it works out the same as with a single-layer perceptron). If $w_{t,s}$ is not in $\mathbf{W}_l$, then we need to keep expanding,

$$\frac{\partial \mathrm{SE}}{\partial w_{t,s}} = \frac{-2(\mathbf{y}_i - \hat{\mathbf{y}}_i)a'(\mathbf{p}_l)\mathbf{W}_l \partial a(\mathbf{p}_{l-1})}{\partial w_{t,s}}.$$

We can apply the chain rule again,

$$\frac{\partial \mathrm{SE}}{\partial w_{t,s}} = \frac{-2(\mathbf{y}_i - \hat{\mathbf{y}}_i)a'(\mathbf{p}_l)\mathbf{W}_l a'(\mathbf{p}_{l-1})\partial \mathbf{p}_{l-1}}{\partial w_{t,s}}.$$

Notice the term, $\partial \mathbf{p}_{l-1}$, is similar to $\partial \mathbf{p}_l$, which we have seen before. If we keep going, we will get to $\partial \mathbf{p}_{l-2}$, then $\partial \mathbf{p}_{l-3}$, and so forth. Basically, we are working our way backwards layer-by-layer until we arrive at the layer that contains $w_{t,s}$. When that finally occurs, we will be able to work out the ∂ term.

Let's arbitrarily suppose that $w_{t,s}$ is in layer $l - 5$. Then, we would work it out to,

$$\begin{gathered} = -2(\mathbf{y}_i - \hat{\mathbf{y}}_i)a'(\mathbf{p}_l)\mathbf{W}_l a'(\mathbf{p}_{l-1})\times \\ \mathbf{W}_{l-1}a'(\mathbf{p}_{l-2})\mathbf{W}_{l-2}a'(\mathbf{p}_{l-3})\mathbf{W}_{l-3}a'(\mathbf{p}_{l-4})\times \\ \frac{\mathbf{W}_{l-4}a'(\mathbf{p}_{l-5})\partial \mathbf{W}_{l-5}\mathbf{q}_{l-6}}{\partial w_{t,s}}. \end{gathered}$$

Now, we can finally resolve the ∂ term,

$$\begin{gathered} = -2(\mathbf{y}_i - \hat{\mathbf{y}}_i)a'(\mathbf{p}_l)\mathbf{W}_l a'(\mathbf{p}_{l-1})\times \\ \mathbf{W}_{l-1}a'(\mathbf{p}_{l-2})\mathbf{W}_{l-2}a'(\mathbf{p}_{l-3})\mathbf{W}_{l-3}a'(\mathbf{p}_{l-4})\times \\ \mathbf{W}_{l-4}a'(\mathbf{p}_{l-5})\mathbf{q}_s \end{gathered}$$

Now, we have something that we can actually evaluate. Since this equation has a lot of repeating parts, it would make sense to write a function to handle those parts. The repeating part can be implemented with the recursive function,

$$\delta_s = \delta_t \mathbf{W} a'(\mathbf{p}).$$

If we expand it to be just a little more explicit, it is

$$\delta_s = \sum_t w_{t,s}\delta_t a_s'(p_s).$$

This is the equation for backpropagating the blame. So, we see that “blame” is really just a temporary value that we store so that we don't have to compute it again for the other units in the network. It is nothing more than a term in the equation for the gradient. So, plugging this in simplifies our equation all the way until we arrive at the output units, where we can describe their “blame” as,

$$\delta_u = 2(y_{i,u} - q_u)a'_u(p_u).$$

Since we are going to use a learning rate to optimize, we can absorb the “ 2” into that learning rate to obtain,

$$\delta_u = (y_{i,u} - q_u)a'_u(p_u).$$

With these abstractions, the only parts of the gradient equation that remain unaccounted for are a minus sign and a “q_s”. So, the equation for the gradient reduces to

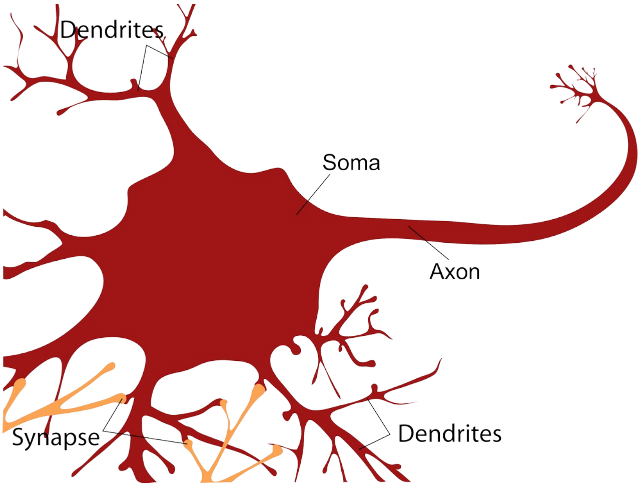
$$\frac{\partial \mathrm{SE}}{\partial w_{t,s}} = -\delta_t q_s.$$

That's it. We just derived all of the equations for backpropagation. To summarize, we started by expressing the gradient of the error surface with respect to an arbitrary weight. We used the chain rule to reduce it to an equation that we could evaluate. Then, we implemented this equation by abstracting much of the computation with “blame” terms. This resulted in all of the equations for computing the gradient of a multilayer perceptron.

4.3.7 Analogs with biological neural networks

Some of the terms and ideas used used with artificial neural networks have historical origins in neuroscience.

The human brain is a network of brain cells, called *neurons*. Here is a simple diagram of a neuron:



The main body of the neuron is called the *soma*. A long protrusion, called the *axon* extends from the soma. The axon carries pulsing or “spiking” electrical signals that the neuron sends to other neurons in the network. The axon may be thought of as the output of the neuron. *Dendrites* are the inputs into they neuron. The axons of other neurons connect with the dendrites through connections called *synapses*. The synapses regulate how much of the signal from the sending neuron is received by the receiving neuron. They perform a task analogous to that of the weights in an artificial neural network.

At a high level, the function of a single neuron is rather simple. It essentially accumulates charge from the pulsing spikes that it receives through its dendrites as inputs signals. When the accumulated charge reaches a certain threshold, the neuron activates, sending a spike of electrical voltage down its axon to other neurons in the network.

At a low level, neurons are living cells, and are therefore very complex. An important unanswered question regarding neurons is, could the simple high-level function of a neuron be sufficient to implement human-like intelligence, or do the complex low-level operations of the neuron play an important role in making the human brain so effective?

The answer to this question is not yet known, for sure, but we can look to similar situations that have occurred in the past. For example, birds once provided inspiration for people to build flying machines. Some of our earliest designs for such machines incorporated feathers and flapping wings. Now that we understand the principles of physics that govern flight to a much greater extent, machines with flapping wings just seem comical, and no one would even think twice about trying to implement mechanical analogs of a bird's digestive or circulatory systems. Yet, although our modern airplanes deviate so much from birds, they can fly higher and faster, they can carry much heavier cargo, and they can drop missiles with much greater destructive power.

With artificial neural networks, science is still trying to figure out just how much it can do with networks of simple units that only simulate neurons at a very high level. Attempts have been made to incorporate very complex more-neuron-like units into artificial neural networks, but they have generally hurt more than they have helped. Since our capabilities with networks of simple units are still growing, it is not surprising that extra complexity has been of little benefit overall in our artificial neural networks.

4.3.7.1 Spiking versus non-spiking neurons

An artificial neuron that attempts to simulate a biologically-plausible neuron is called a *spiking neuron*, because biological neurons send signals in pulses or spikes of electrical voltage.

The biggest simplifying assumption we typically use to reduce complexity in artificial neurons is to eliminate the element of time. This is done by representing the frequency of the spikes, instead of their voltages. We call neurons of this type *non-spiking*. For example, if the values 0.7 and -1.4 feed into a non-spiking neuron, and the value 0.13 comes out, we can say that 0.7 and -1.4 are a representation of the rates at which other neurons are firing signals that feed into this one, and 0.13 is a representation of the rate at which this neuron is firing. When these values change, we say the neurons have changed their firing rate. Non-spiking neurons are much simpler to implement because there is no need to worry about time.

In theory, amplitude modulation and frequency modulation are just two different ways of encoding information. Anything that can be encoded one way can be encoded the other way, as well. (As an example, the Fourier transform bidirectionally converts between these two ways of representing information.) So, it is commonly believed that the general proficiency of the human brain probably does not depend on using a spiking representation of information.

Much research has been done to explore both methods for representing artificial neurons. Although spiking neurons are more biologically plausible, non-spiking models have surged ahead in demonstrating effectiveness, and spiking models seem to be constantly trying to catch up with algorithms for training non-spiking neurons. In many ways, this is consistent with Occam's razor: the simplest model capable of representing the data appears to be the better choice. However, we must ultimately admit that general intelligence has already been successfully implemented in the human brain with the spiking model, and artificial general intelligence has not yet been achieved by any model at all. Until that point is actually achieved, it is difficult to be certain whether there may be subtle advantages to using the spiking model that we have not yet figured out how to utilize.

4.3.7.2 The “one algorithm” theory

In the field of neuroscience, experiments have been performed in which the nerves connecting the eyes and ears of ferrets were disconnected from their brains, and reattached to different regions. The ears were reconnected to the visual cortex of the brain, the part responsible for seeing, and the eyes were reconnected to the audio cortex, the region responsible for hearing. Over time, the ferrets eventually regained their abilities to hear and see. Their audio cortexes learned to hear, and their visual cortexes learned to see.

Similar experiments were repeated by other researchers, on other continents, and using different animals. In another experiment, a third eye on the back of a frog's head was wired into its brain. This frog eventually learned to see with its additional eye, giving it the ability to respond to visual stimulus that a normal frog would not even be capable of detecting.

Even human brains have demonstrated similar capabilities. People who lose their sight often develop augmented abilities for hearing. Human “cyborgs” have augmented their existing senses with devices that play sounds, induce tactile sensations, or create other forms of stimulus in response to devices that measure things they cannot physically sense. These people often report that the experience eventually stops feeling like the sound or touch sensations used in the interface, and that they learn to experience the augmented sensory abilities as if they were an innate part of their bodies.

All of these experiments suggest that the brain adapts to the signals it is given. Rather than being specifically hard-wired to experience audio, visual, tactile, olfactory, and gustatory stimuli, it learns to work with the signals it receives, and the various parts of the brain all seem to have similar capabilities. In other words, general intelligence may depend much more on a single algorithm for learning than a complex architecture for cognitive function.

4.3.7.3 Hebbian learning

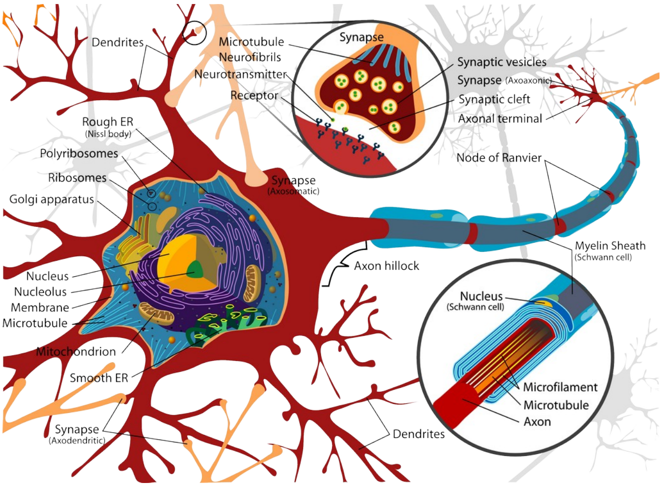
[todo: Discuss the Hebbian mantra, “neurons that fire together wire together”]

[todo: ponder, is backpropagation a frequency-modulated approximation of Hebbian learning in amplitude-modulated spiking representations? Have we already found the “one algorithm”?]

4.3.7.4 Brain structure

Since we do not yet know for sure which aspects of the human brain are necessary for general intelligence, and which aspects are merely arbitrary byproducts of evolution, it may be a good idea to learn as much as possible about it.

Here is a diagram of a neuron I lifted from Wikipedia showing a little more detail than the previous one:



One notable aspect is the myelin that surrounds the axon. This is a protective covering composed primarily of cholesterol, and serves a purpose similar to the plastic insulation that we often use to protect electrical wires. It is believed that myelin enables brains to grow much larger by facilitating axons that traverse much greater distances.

Axons coated with myelin are white in appearance, whereas the somas of neurons are typically more of a gray color. When a human brain is dissected, we find that the outer surace, called the cerebral cortex, is composed of “gray matter”, and the inner bulk of the brain is primarily “white matter”. In other words, most of the neurons are found in the rippled outer layer of the brain, and most of the inner bulk consists of myelinated axons connecting various regions of the brain.

[todo: complete this section]

4.3.7.5 Animals by numbers of neurons

The following table shows the estimated number of neurons found in the complete nervous systems of a few animals:

Caenorhabditis Elegans Nematode Roundworm	302
Jellyfish	800
Fruit Fly	100,000
Ant	250,000
Mouse	71,000,000
Human	86,000,000,000
African Elephant	300,000,000,000

If we count the total number of neurons, the African Elephant has us beat. However, many of these neurons are part of its system for relaying signals throughout its large body. If we only count neurons in the cerebral cortex, or gray matter of the brain, humans have about twice as many neurons:

Mouse	4 Million
Dog	160 Million
Cat	300 Million
Bottlenose Dolphin	5.8 Billion
Chimpanzee	6 Billion
False Killer Whale	10.5 Billion
African Elephant	11 Billion
Humans	22 Billion

4.4 Some advanced optimization concepts

Todo: transition

4.4.1 Why we minimize SSE

Todo: derive why SSE is a maximum likelihood estimator if we assume noise is Gaussian (normal). Also talk about why the central limit theorem suggests that it is reasonable to assume noise is distributed normally.

4.4.2 Regularization

Overfit is a significant problem in machine learning. With neural networks, common ways to mitigate overfit include:

- Get plenty of training data.
- Limit the number of network units.
- Limit the amount of training that occurs.
- Regularize the weights.

The first three of these are straightforward. Now, we will discuss regularization.

When we initialize a neural network, we typically use small weights. This causes the net values that feed into the activation function to also be small. Since tanh (and most other common activation functions) approximate the identity function at values close to zero, the whole network initially behaves much like a linear model. As training continues, the weights tend to become larger. Now, warps and bends start to appear in the model as it stretches in various directions to meet each of the points in the training data. The larger the weights become, the sharper and bends become. This is when overfit starts to become a problem. So, we see that overfit may be related to having large weights. Regularization attempts to limit the weights in some way.

4.4.2.1 Weight decay

The error function most commonly used with neural networks is sum-squared-error,

$$\sum_i (\mathbf{y}_i - h(\mathbf{x}_i))^2.$$

In addition to penalizing “error”, weight decay adds an additional term to also penalize large weights,

$$\sum_i (\mathbf{y}_i - h(\mathbf{x}_i))^2 + \lambda \sum_j w_j^2.$$

The variable λ (lambda) is called the regularization term. It can be tuned to adjust the balance balance between the importance of having low error and the importance of having small weights. Typically, small values λ, such as 0.0001, tend to give desirable behavior.

To update the weights in our network, we start with this new regularized target function, and re-derive the gradient. The first summation term works out the same as always, and the second summation term is quite easy to differentiate:

$$\frac{\partial \text{SE}}{\partial w_k} \lambda \sum_j w_j^2 = 2\lambda w_k.$$

We can absorb the "2" into the λ. Also, remember that to update the weights, we subtract the gradient times a step size, η. So, the weight update rule with weight decay works to to be the same as before, except we subtract $\eta\lambda w_j$ from each w_j just before we update w_j in the usual manner. And that is the same as multiplying all the weights in the network by $1 - \eta\lambda$ just before updating the weights in the usual manner.

For example, if $\eta = 0.1$ and $\lambda = 0.001$, then you would multiply all the weights by 0.9999 after you back-propagate the error terms, but before you update the weights based on those error terms. So, in other words, you "decay the weights" just a little bit while doing regular training.

Weight decay is easy to implement, and it works reasonably well. It is the oldest and most well-established method for regularizing a neural network. Overall, it tends to promote networks that fit to the training data while giving approximately equal magnitude to each weight in the network.

When used with other models, besides neural networks, weight decay is called L^2 regularization (especially when it is compared with L^1 regularization, which we discuss next), Euclidean Norm regularization, or Tikhonov regularization. Regression that uses this type of regularization is also called ridge regression.

4.4.2.2 L^1 regularization

L^1 regularization penalizes weights according to their absolute value,

$$\sum_i (\mathbf{y}_i - h(\mathbf{x}_i))^2 + \lambda \sum_j |w_j|$$

If we work out the weight-update rule for L^1 regularization, it turns out to be to subtract $\eta\lambda$ from all the positive weights, and to add $\eta\lambda$ to all the negative weights, just before you update those weights in the usual manner.

L^1 regularization is similar to L^2 regularization in many ways, but its effect is quite different. Assuming the value of λ is small, both of them apply a gentle pressure on the weights to move closer to zero during training. The difference is that the pressure applied by L^2 regularization is proportional to the size of each weight, whereas the pressure applied by L^1 regularization is constant across all weights. So, whereas L^2 regularization tries to find a set of weights that are all approximately equally small, L^1 regularization tries to find a set of weights containing as many zero-magnitude weights as possible.

In other words, if you want a lot of small weights, use L^2 regularization. If you want sparse weights, use L^1 regularization.

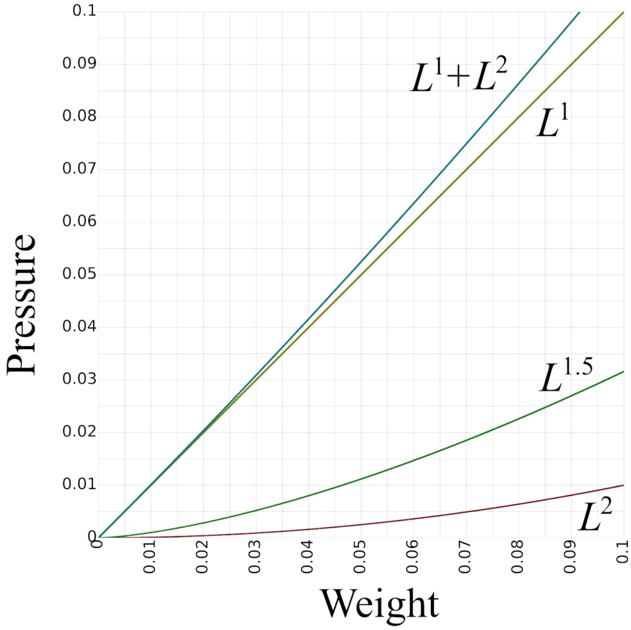
L^1 is also called LASSO (least absolute shrinkage and selection operator), especially when used in regression with other models besides neural networks.

4.4.2.3 Combining L^1 and L^2 regularization

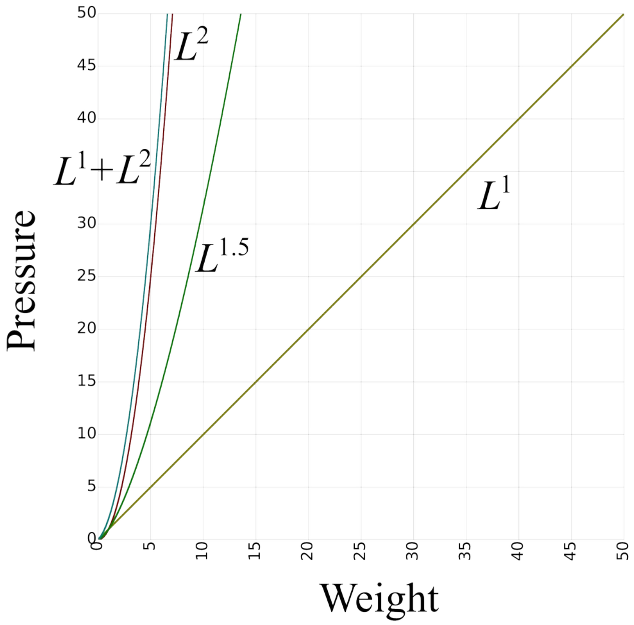
One can easily generalize these two concepts to form L^p regularization, where p is 1, 2, or any other value. This is done by subtracting $\eta\lambda w_j^{p-1}$ from the positive weights, and adding $\eta\lambda w_j^{p-1}$ to the negative weights, just before updating them in the usual manner.

Intuitively, one might expect that going part-way between L^1 and L^2 regularization, perhaps with $L^{1.5}$ regularization might give the best of both worlds. That is, one might hope to both prevent weights from getting big, and promote sparsity. Unfortunately, it actually does the opposite. When weights are small, it behaves more like L^2 regularization by failing to promote sparsity, and when weights are large, it behaves more like L^1 by failing to put much pressure on the weights to remain small.

A generally better solution is to apply both L^1 and L^2 regularization together. Doing this is called Elastic Net. It is implemented by adding $\eta(\lambda_2 w_j + \lambda_1)$ to each negative weight, w_j, and adding $\eta(\lambda_2 w_j - \lambda_1)$ to each positive weight, w_j. One drawback is that now there are two regularization terms, λ_1 and λ_2, instead of just one.



The preceding figure shows that ElasticNet (L^1+L^2) behavies like L^1 regularization with small weights, whereas $L^{1.5}$ regularization behaves more like L^2 regularization in this region.



The preceding figure shows that ElasticNet (L^1+L^2) behavies like L^2 regularization with large weights, whereas $L^{1.5}$ does something part way between L^1 and L^2 regularization (which may or may not keep up with the rate at which large weights grow during training).

4.4.2.4 Contractive regularization

Contractive regularization is a type of regularization commonly applied to the encoder of an autoencoder. It causes the encoding to fill up its available space, and minimize abrupt changes. It is called “contractive” because this behavior can be viewed intuitively as if the space contracted around the encoding ...sort of.

Contractive regularization is implemented by subtracting $\eta\lambda a'(n)w_j$ from each weight, w_j, where a' is the activation function of the unit into which weight w_j feeds, and n is the net input value that currently feeds into that unit.

Intuitively, $a'(n)$ is close to zero when the surface represented by the neural network is flat, and it is large when it is “bendy” or curved. So, this type of regularization essentially tries to promote small weights where the surface of the network changes. In other words, it tries to promote an encoding that avoids abrupt changes.

It can be derived by adding a penalty term of $\lambda\frac{\partial h}{\partial x}$ to the error function, where h is the function represented by the neural network, and x is the input that feeds into it. Thus, it penalizes cases where the function represented by the neural network would change a lot when the inputs are slightly perturbed. (A more crude way to achieve the same effect is to inject random perturbations into the input, and train the autoencoder to predict the original noise-free pattern. However, contractive regularization is generally considered to be a more elegant solution.)

4.4.2.5 Max norm

Max norm regularization does not specify how the weight ends up being distributed. Instead, it simply places a cap on the magnitude of the weight vector that feeds into each unit. If $\mathbf{w}_i$ is the vector of weights that feeds into unit i, then max norm regularization can be implemented by doing,

$$\text{if } ||\mathbf{w}_i|| > \tau \text{ then } \mathbf{w}_i = \frac{\tau}{||\mathbf{w}_i||}\mathbf{w}_i,$$

immediately after the weights are updated in the usual manner, where τ is some threshold.

Naturally, max norm regularization could be implemented with any L^p-norm magnitude, but L^2 is the most common since it is invariant to rotation.

4.4.2.6 Dropout and DropConnect

One way that biological brains differ from artificial neural networks is that biological neurons are not entirely reliable. Sometimes, they don't fire when they should. Also, they have a recovery period that often renders them unable to fire when they otherwise would. Simulating this behavior in an artificial neural network is called Dropout.

Dropout has a regularizing effect on artificial neural networks. It has been reported to be more effective than weight decay with big neural networks on many problems.

It can be implemented as follows: After the activations for a layer are computed, randomly pick a portion of the neurons, and set their activations to zero. When weights are updated, the blame term attributed to neurons that did not fire will be zero, so their weights are not updated. A common drop-out rate is 50%, meaning about half of the neurons will fail to fire.

Naturally, this causes the artificial neural network to not rely too much on any one neuron. Since any neuron may fail to do its job, they all try to spread the burden around.

After training is complete, dropout is typically not used anymore. If nodes were dropped with a probability of *p*, then all the weights must be multiplied by (1-*p*) to compensate for all of the nodes now being allowed to participate.

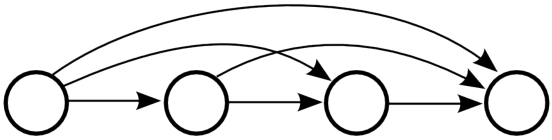
DropConnect is an obvious variation in which weights (a.k.a. connections) are randomly dropped instead of nodes. It is not as well-studied as dropout, probably because it is not as trivial to implement, but it claims to yield slightly better results.

4.4.3 Connections

4.4.3.1 Fully-connected layers

The most common type of neural network is the multilayer perceptron, which consists of multiple layers of fully-connected feed-forward units. The term “feed-forward” suggests that there are ny cycles. Every connection feeds in a forward direction. The term “fully-connected” suggests that every unit in a layer is connected to every unit in the next layer.

However, such a network is not really as “fully-connected” as possible. If we wanted to, we could connect every unit in the network to every downstream unit, and still have a feed-forward network.



Such a network would not really have “layers” at all. You could still use backpropagation to implement gradient descent with such a network. In many ways, this idea is appealing because it would essentially allow gradient descent to find the best topology, and would spare the user from having to specify a topology at all. (The user would only need to specify the total number of units to use in the network.)

In practice, however, all those extra weights tend to do more harm than good. In contrast with this approach, some of the more effective neural networks have been those that take measures to limit the number of weights that are available for the network to tune, while still providing a lot of connections. The next two sections discuss models that use fewer weights with many connections.

4.4.3.2 Restricted Boltzmann machines

A restricted Boltzman machine (RBM) is a type of autoencoder. It uses one layer of units for the encoder, and one layer of units for the decoder. However, instead of giving these two layers their own sets of weights, they share a common set of weights.

Specifically, the matrix of weights for the encoding layer is the transpose of the weights for the decoding layer. Although both layers share the same weights matrix, they each have their own bias vector. With RBMs, the encoder maps from “visible units” to “hidden units”, and the decoder maps from “hidden units” to “visible units”. So, an RBM with 7 visible units and 5 hidden units would have a total of 47 weights: 35 for the shared matrix of weights, 7 bias weights for the visible units, and 5 bias weights for the hidden units.

If you feed a vector through an RBM in the encoding direction, a “hidden” representation is produced. If you feed that hidden representation back through in the decoding direction, a “visible” representation is reconstructed. If you continue encoding, decoding, encoding, decoding, encoding decoding, encoding, decoding, etc., eventually the two representations will probably stop changing. That is, the visible representation will converge to something, and the hidden representation will converge to something. (This is true with other kinds of autoencoders too, but it is more common to think about RBMs in this way.) An RBM is generally considered to be well-trained if it eventually converges to a “good” vector no matter what vector you initially feed into it. In this case, a “good” vector is one like those found in the training data.

In other words, an RBM is often thought of as a type of distribution approximator. The training data is thought of as a collection of samples from the target distribution, and the RBM is a model of that distribution. You can draw a new sample from this model of the distribution by feeding a random vector into the RBM, and then encoding it, decoding it, encoding it again, decoding it again, etc., until it converges to something.

So, how do we train an RBM? Because the weights for the encoder and decoder are tied together, it is difficult to calculate the exact gradient for updating the weights of an RBM. For this reason, they are not typically trained by gradient descent. Instead, they are trained using a method called *contrastive divergence.*

Contrastive divergence is based on the notion that if an RBM is well-trained, and we seed it with a sample from the training data, then it should converge immediately without changing that sample (because it came from the training data, which is the best representation of the target distribution we have). Anything the RBM does to change one of the training samples is considered “wrong”, and the more it changes it, the more wrong it is. So, the difference between any vector that we present from the training data to the RBM, and the vector it reconstructs after *k* encoding-decoding cycles provides an error signal that we can use to update the weights.

[todo: describe formally]

[todo: finish this section]

4.4.3.3 Convolutional layers

4.4.4 Lagrange multipliers

Sometimes we need to optimize something, but not all possible solutions are acceptable. For example, if we are trying to model a distribution, then only acceptable solutions consist of probabilities that sum to 1 and are never negative. These limitations are called *constraints*. Solving an optimization problem with constraints is called *constrained optimization*.

Some optimization techniques, such as linear programming, are designed to work within constraints, but these are typically limited to working with simplistic models, such as linear models. Most general-purpose optimization techniques, such as hill climbing, are not designed to work with constraints.

Lagrange multipliers are a method for converting a constrained optimization problem into an unconstrained optimization problem. It does this by adding some extra variables, called *Lagrange multipliers*, to the problem. After you solve the unconstrained problem with extra variables, you can throw out those extra variables, and the remaining values are the solution to the constrained optimization problem.

[todo: finish]

4.5 Generative and transductive variants

Todo: write this section (distinguish from discriminative and supervised models)

4.5.1 Autoencoders

Todo: discuss bottle-neck, weight-decay, noise-reducing, whitening, and contractive variants

4.5.2 Principal component analysis

Todo: write this section

4.5.3 Matrix factorization

Todo: write this section

4.5.4 Nonlinear matrix factorization

Todo: write this section

4.5.5 Transductive learning

Todo: write this section

4.6 “Deep” neural network learning

Todo: write this section

4.6.1 Parallel implementations

Todo: discuss general-purpose graphical processing units.

Todo: discuss dedicated hardware

4.6.2 Unsupervised pretraining

Todo: write this section

4.7 Temporal models

Todo: write this section

todo: EKF, particle filter, Gaussian processes (kriging), Fourier nets, etc

4.8 Related and historical models

Todo: discuss Hopfield networks, self-organizing map, reservoir networks, support vector machines,

4.9 Deep neural networks

Todo: write this section

Todo: Eventually, this should reach all the way to AI, reinforcement learning, planning methods (breadth-first, A*, POMDPs, etc.), cognitive architectures, etc

5 Appendixes

This section contains some important content that did not fit very well with the rest of the book.

5.1 Review of some essential linear algebra concepts

To make sure you are comfortable with the matrix form of this formula, let's do a quick review of basic vector and matrix operations.

In this book, we usually represent vectors as lowercase letters in bold font. (Sometimes a vector is represented with a little arrow over it, but we do not use that notation here.) We represent scalars as lowercase letters in italics.

First, let's add two vectors. Suppose you have two vectors:

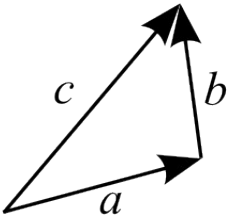
$$\mathbf{a} = \langle a_1, a_2, a_3, ..., a_n \rangle, \text{ and}$$

$$\mathbf{b} = \langle b_1, b_2, b_3, ..., b_n \rangle.$$

If $\mathbf{c} = \mathbf{a} + \mathbf{b}$, then

$$\mathbf{c} = \langle a_1 + b_1, a_2 + b_2, a_3 + b_3, ..., a_n + b_n \rangle.$$

Intuitively, if you think of a vector as an arrow, then adding two vectors is like placing the two arrows end-to-end.



The magnitude of a vector is its length.

$$||\mathbf{a}|| = \sqrt{(a_1)^2 + (a_2)^2 + (a_3)^2 + ... + (a_n)^2}.$$

Therefore, the distance between two vectors is the magnitude of their difference, $||\mathbf{b} - \mathbf{a}||$ or $||\mathbf{a} - \mathbf{b}||$.

A point and a vector are really the same thing. They are both just an ordered list of values. We usually draw a point with a dot, and a vector with an arrow, but this is just convention. We can use points as vectors and vice versa. They are just ordered lists of values. The meaning of those values really depends on how you use them.

A centroid is the point that represents the mean of a set of points. You can compute a centroid by computing the mean of each attribute in a dataset.

The inner product (a.k.a. dot product) is a product that can be computed from two vectors. If $c = \mathbf{a} \cdot \mathbf{b}$, then

$$c = a_1 b_1 + a_2 b_2 + a_3 b_3 + ... + a_n b_n.$$

(Note that inner product returns a scalar.)

Intuitively, the inner product measures the extent to which the two vectors are pointing in the same direction. If the two vectors point in exactly the same direction, their inner product is the product of their magnitudes. If they point in opposite directions, the inner product is the negative product of their magnitudes. If the point in orthogonal directions, the inner product is 0.

Okay, now let's talk about matrices. In this book, we usually represent matrices with capital letters in bold font. A matrix is just a table of values. It has rows and columns. Just like scalars and vectors, the meaning of a matrix depends on how it is used. Since matrices are used for many different things, the rows and columns can mean different things.

We refer to the elements in a matrix by row, and then column. For example, if $\mathbf{M}$ is a matrix, then $m_{i,j}$ refers to the element in row i, column j. If we only specify one index, then we are referring to a row of the matrix. So, $\mathbf{m}_i$ is row i of $\mathbf{M}$.

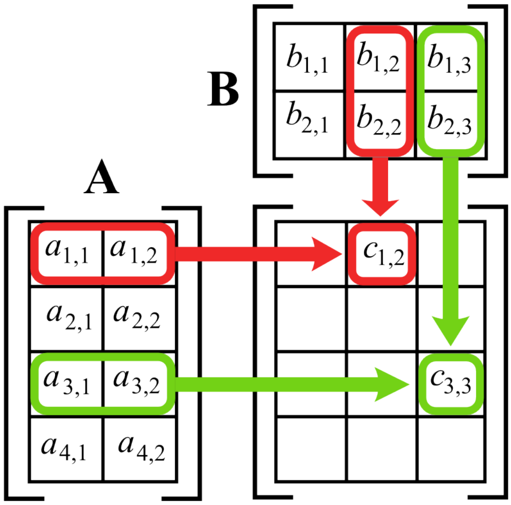
You can transpose a matrix, $\mathbf{M}^{\mathrm{T}}$, by switching all the rows with the columns. In other words, you rotate the matrix around the diagonal such that each $m_{i,j}$ is moved to $m_{j,i}$.

Matrices can also be inverted. This is denoted as $\mathbf{M}^{-1}$. Matrices must meet certain conditions in order to be invertible, but a generalization of matrix inversion, called the *Moore-Penrose pseudoinverse*, can be performed on any matrix. It is more correctly called a generalized inverse, because it is equivalent to inverse whenever the matrix is invertible, and it finds the closest (least-squares) approximation when the matrix is not invertible. So, robust implementations usually use the generalized inverse. Unfortunately, it is a fairly complex operation, so we will not describe it here.

Matrices can be multiplied. To multiply two matrices,

$$\mathbf{C} = \mathbf{A} \times \mathbf{B},$$

one computes the inner product of each row in $\mathbf{A}$ with each column in $\mathbf{B}$.



Note that $\mathbf{C}$ is not necessarily the same size as $\mathbf{A}$ or $\mathbf{B}$. It has the same number of rows as $\mathbf{A}$, and the same number of columns as $\mathbf{B}$. Each element in $\mathbf{C}$ is just the inner product of the corresponding row in $\mathbf{A}$ and column in $\mathbf{B}$.

If you want to multiply a matrix by a vector, you just treat the vector like a one-column matrix.

So, I like to visualize the equation for a linear model,

$$\hat{\mathbf{y}} = \mathbf{M}\mathbf{x} + \mathbf{b},$$

like this:

$$\begin{array}{cc} & \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \\ \begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,1} & m_{2,2} & m_{2,3} \end{bmatrix} & \begin{bmatrix} \ \\ \ \end{bmatrix} \end{array} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \end{bmatrix}$$

If we expand it all down to scalar arithmetic, we get,

$$\hat{y}_1 = m_{1,1}x_1 + m_{1,2}x_2 + m_{1,3}x_3 + b_1,$$

$$\hat{y}_2 = m_{2,1}x_1 + m_{2,2}x_2 + m_{2,3}x_3 + b_2.$$

You might notice that a vector is really just a one-row matrix. Or is it a one-column matrix? Well, the truth is, people really use vectors when they don't want to worry about specifying whether it is a column or a row. You are just supposed to figure out which one is appropriate for the situation. Sometimes either one could work just fine. Sometimes it is important to treat them in a specific manner. For example, when you multiply two vectors, the result depends on whether they are row or column vectors. If you multiply a row vector by a column vector, you get a single value. We denote this with a single dot. If you multiply a column vector by a row vector, you get a whole matrix of values. We denote this with a "$\otimes$". So, if $\mathbf{a}$ and $\mathbf{b}$ are both m-dimensional vectors, then

$$\mathbf{a} \cdot \mathbf{b} = \langle a_1b_1 + a_2b_2 + \cdots + a_mb_m \rangle$$

and

$$\mathbf{a} \otimes \mathbf{b} = \begin{bmatrix} a_1b_1 & a_1b_2 & \cdots & a_1b_m \\ a_2b_1 & a_2b_2 & & a_2b_m \\ \vdots & & \ddots & \\ a_mb_1 & a_mb_2 & & a_mb_m \end{bmatrix}$$

The inner product (a.k.a. dot product) requires that $\mathbf{a}$ and $\mathbf{b}$ be the same size, whereas the outer product (a.k.a. tensor product) can be done with vectors of mismatching sizes, resulting in a non-square matrix. In essence, inner product and outer product are really the same operation, differing only in which vector you implicitly “transpose” before you multiply them.

(Be careful not to confuse either of these operations with vector cross-product, denoted with a “$\times$” symbol. That is yet another operation, but we will not worry about it at this point.)

Even scalars can be said to be 1-dimensional vectors, or 1×1 matrices, but people usually do not want to bother worrying about whether they have a row-scalar or a column scalar, since transposing them has no real effect and multiplying them results in the same value either way. With vectors, the dimensions are more relevant. With matrices, even more so. Unfortunately, the standard notation for matrices and vectors does a poor job of specifying dimensions, so the reader must simply be careful. Often, if the writer is diligent, informative clues can be found in the text near the equations to help the reader understand them.

5.2 Review of the chain rule

The chain rule is a useful tool for differentiation. It lets you break down a difficult differentiation problem into two simpler ones. Often, it can be applied recursively to turn otherwise nasty differentiation problems into a whole bunch of easy differentiation problems.

Specifically, the chain rule tells us how to take the derivative of a “function of a function”. It says,

$$\frac{\partial}{\partial x} f(g(x)) = f'(g(x)) \frac{\partial}{\partial x} g(x).$$

Many students learn about the chain rule in Calculus class, but promptly forget it because they never use it for any practical purpose. Well, we use it in this book to derive the gradient for an MLP. Why? Because each layer of an MLP is a function. Values go in, and values come out. If you have an MLP with several layers, it may be described as a function of a function of a function of a function of a function. If we treat the whole MLP as one big function, it would be very difficult to differentiate. If we use the chain rule, however, it becomes quite easy because a single layer isn't very hard to differentiate.

The chain rule essentially lets us peel off the outer-most layer of the onion. Then, we can repeat the process until we arrive at the center.

Let's start with a simple example with polynomials to refresh our memories about the chain rule:

Suppose $f(x) = x^2 - x + 1$, and suppose $g(x) = 7x^2 + 3$. We can easily calculate the derivatives of these functions:

$f'(x) = 2x - 1$, and
$g'(x) = 14x$.

Now, suppose some larger function is defined as a combination of f and g:

$$h(x) = f(g(x)).$$

Suppose we want to calculate $h'(x)$. The chain rule tells us that

$$h'(x) = f'(g(x))g'(x).$$

To word this in English, you take the derivative of the outer function times the derivative of “the inside”. So, we can expand and simplify this to:

$$h'(x) = (2(g(x)) - 1)g'(x),$$
$$h'(x) = (2(7x^2 + 3) - 1)14x,$$
$$h'(x) = (14x^2 + 5)14x,$$
$$h'(x) = 196x^3 + 70x.$$

Tada! We used the chain rule to calculate the derivative of $h(x)$. Now, let's check our work. If we expand $h(x)$ from the very start, we get:

$$h(x) = f(g(x)),$$
$$h(x) = g(x)^2 - g(x) + 1,$$
$$h(x) = (7x^2 + 3)^2 - (7x^2 + 3) + 1,$$
$$h(x) = 49x^4 + 35x^2 + 13.$$

Then, if we take the derivative of $h(x)$, we get

$$h'(x) = 196x^3 + 70x.$$

Yay, it's the same answer. That means the chain rule really works. In this case, since we used simple polynomials, the chain rule may not have really saved us very much work. In a lot of other cases, however, it makes an otherwise insanely difficult problem quite simple. Such is the case with MLPs.