O'REILLY®

Thoughtful Machine Learning

A TEST-DRIVEN APPROACH

Matthew Kirk

Thoughtful Machine Learning

Matthew Kirk

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Thoughtful Machine Learning
by Matthew Kirk

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Ann Spencer
Production Editor: Melanie Yarbrough
Copyeditor: FIX ME!
Proofreader: FIX ME!
Indexer: FIX ME!
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

October 2014: First Edition

Revision History for the First Edition:

2014-08-05: Early Release Revision 1

See *http://oreilly.com/catalog/errata.csp?isbn=9781449374068* for release details.

ISBN: 978-1-449-37406-8

[?]

Table of Contents

CHAPTER 1
Test Driven Machine Learning

A great scientist is a dreamer and a skeptic. In our modern history scientists have made exceptional breakthroughs like discovering gravity, going to the moon and the theory of relativity. All those scientists had something in common: they dreamt big. However, this wasn't without testing and validating their work first.

While we aren't in the company of Einstein and Newton these days, we are in the age of big data. With the rise of information age, it has become increasingly important to find ways to manipulate that data into something meaningful - otherwise known as the subject of data science and machine learning.

Machine learning has been a subject of interest because of its ability to use information to solve complex problems like facial recognition or handwriting detection. Many times, machine learning algorithms do this by having tests baked in. Examples of these tests are statistical hypotheses, thresholds and minimizing mean squared errors over time. Theoretically, machine learning algorithms have build a solid foundation. These algorithms have the ability to learn from past mistakes and minimize errors over time.

However, we as humans don't have the same rate of effectiveness. While internally the algorithms are minimizing error, sometimes we're not minimizing the right error, or we might be making errors in our own code. Therefore, we need tests for human error, as well as a way to document our progress. The most popular way of writing these tests is called Test Driven Development or TDD for short. This method of writing tests first has become popularized as a best practice for programmers. However, it is a best practice that is sometimes not exercised in a development environment.

There are two good reasons to use test driven development. One reason is that while test driven development takes 15-35% more time in active development mode, it also has the ability to reduce bugs up to 90% [[*http://research.microsoft.com/en-us/news/features/nagappan-100609.aspx*]]. Another reason to use test driven development is for the benefit of documenting how the code is intended to work. As code becomes more

complex, the need for a specification becomes greater - especially as people are making bigger decisions based on what comes out of analysis.

Harvard scholars Carmen Reinhart and Kenneth Rogoff wrote a economics paper that was cited multiple times stating that countries that took on over 90% of their gross domestic product suffered sharp drops in economic growth. Paul Ryan used this conclusion heavily in his presidential race. In 2013, three researchers from University of Massachusetts found that the calculation was incorrect because it was missing a substantial amount of countries from its analysis.

While some examples aren't as drastic as this one, it is a blow to one's academic reputation due to one error doing the statistical analysis. One mistake can cascade into many more - and this is the work of Harvard researchers who have been through a rigorous process of peer review and have years of experience in research. It can happen to anybody. Using test driven development would have helped to mitigate the risk of making such an error - and would have saved them from the embarrassment.

History of Test Driven Development

In 1999, Kent Beck popularized Test Driven Development (or TDD) through his work with extreme programming. This method's power comes from the ability to first define our intentions and then satisfy those intentions. The practice of TDD involves the writing of a failing test, the writing of the code that makes it pass, and then refactoring. Some people call it "red green refactor" after the colors of many testing libraries. Red is writing a test that doesn't work originally but documents what your goal is, green is about making the code work so the test passes. Finally you refactor the original code that works so that you are happy with it's design.

While testing has always been a mainstay in the traditional development practice, TDD emphasizes testing first instead of testing near the end of a development cycle. In a waterfall model, acceptance tests were used and involved many people after the code was actually written - usually handled by someone who was an end-user and not a programmer. This approach seems good until coverage becomes a factor. Many times quality assurance professionals can only test what they want to test and not get to everything underneath the surface.

TDD & The Scientific Method

Part of the reason why TDD is so appealing is that it syncs well with people and their working style. The process of hypothesizing, testing, and theorizing makes it very similar to the scientific method.

Science involves trial and error. Scientists come up with a hypothesis, test that hypothesis and then combine their hypotheses into a theory.

Hypothesize, test, and theorize could be called Red Green Refactor instead.

Just like the scientific method, writing tests first works well with machine learning code. Most machine learning practitioners apply some form of the scientific method and TDD forces one to write cleaner and more stable code. Beyond the similarity of approaches of TDD to the scientific method, there are three other reasons why TDD is just a subset of the scientific method: logical proposition of validity, sharing results through documentation, and working in feedback loops.

The beauty of test driven development is that you can utilize it to experiment as well. Many times we write tests first with the idea that we will eventually fix the error that is created by the initial test. But it doesn't have to be that way, you can use tests to experiment with things that might not ever work. Using tests in this way is very useful for many problems that aren't easily solveable.

TDD is making a logical proposition of validity

When a scientist is using the scientific method, they are trying to solve a problem and prove that it is valid. Solving a problem requires creative guessing, but without justification it is just a belief.

Knowledge, according to Plato is a justified true belief and we need both a true belief and justification for that. To justify our beliefs, we need to construct a stable logical proposition. In logic, there are two types of conditions to use for proposing whether something is true: necessary and sufficient conditions.

Necessary conditions are those without which our hypothesis fails. For example this could be a unanimous vote, or a pre-flight checklist. The emphasis is that all conditions must be satisfied to be convinced that whatever we are testing is correct.

Sufficient conditions, unlike necessary conditions, mean that there is enough evidence for an argument. For instance thunder is sufficient evidence that lightning has happened since they go together, but thunder isn't necessary for lighting to happen. Many times sufficient conditions take the form of a statistical hypothesis. It might not be perfect but it is sufficient enough to prove what we are testing.

Together necessary and sufficient conditions are what a scientist uses to make an argument for the validity of their solutions. Both the scientific method and TDD use this religiously to make a set of arguments come together in a cohesive way. While the scientific method uses hypothesis testing and axioms, TDD uses integration and unit tests.

Table 1-1. A comparison of TDD to the Scientific Method

	Scientific Method	TDD
Necessary Conditions	Axioms	Pure Functional Testing
Sufficient Conditions	Statistical Hypothesis Testing	Unit and Integration Testing

Example: Proof through Axioms and Functional Tests

Fermat famously conjectured in 1637 that: "there are no positive integers a, b, and c that can satisfy the equation $a^n + b^n = c^n$ for any integer value of n greater than two." On the surface this appears like a simple problem, and supposidly Fermat himself said he had a proof. Except the proof was too big for the margin of the book he was working out of.

For 358 years this problem was toiled over. In 1995, Andrew Wiles solved this problem using Galois transformations and elliptic curves. His 100 page proof was not elegant but was sound. Each section took a previous result and applied it to the next step.

The 100 pages of proof was based on axioms or presumptions that have been proved before much like a functional testing suite would have been done as well. In programming terms all of those axioms and assertions that Andrew Wiles put into his proof could have been written as functional tests. These functional tests are just coded axioms and assertions. Each step feeding into the next section.

This vacuum of testing in most cases doesn't exist in production. Many times the tests we are writing are a scattershot of assertions about the code. In many cases we are testing the thunder not the lightning as I explained above. For that in most cases testing focuses on sufficient conditions not necessary conditions.

Example: Proof through sufficient conditions, unit tests and integration tests

Unlike pure mathematics sufficient conditions are focused on enough evidence to support a causality. An example is inflation. This mysterous force in economics has been studied since the nineteenth century. The problem with proving that inflation exists is that we cannot use axioms.

Instead we rely on the sufficient evidence from our observations that inflation exists. Based on our experience looking at economic data and separating out factors we know to be true we have found that economies tend to grow over time. Sometimes they deflate as well. The existence of inflation can be proved purely on our previous observations which are consistent.

Sufficient conditions like this have an analog to integration tests. Integration tests aim to test the overarching behavior of a piece of code. Instead of monitoring little changes integration tests will watch the entire program and see whether the intended behavior is still there. Likewise if the economy was a program we could assert that inflation or deflation exists.

TDD involves writing your assumptions down on paper or code

Academic institutions require professors to publish their research. While many complain that universities focus too much on publications there's a reason why: publications are the way research becomes timeless. If professors decided to do research in solitude

and make exceptional breakthroughs but didn't publish, that research would be worthless.

Test Driven Development is the same way: tests can be great in peer reviews as well as a version of documentation. Many times documentation isn't necessary due to the usage of tests. Software is abstract and always changing, so if someone doesn't document or test their code it will most likely be changed in the future. If there isn't a test ensuring that the code operates a certain way then when a new programmer comes to work on the software they will probably change it.

TDD and scientific method work in feedback loops

Both the scientific method and TDD work in feedback loops. When someone makes a hypothesis and tests that hypothesis, they find out more information about the problem they're investigating. The same is true with TDD; someone makes a test for what they want and then as they go through writing code they have more information as to how to proceed.

Overall, TDD is a type of scientific method. We make hypotheses, test them and then revisit them. This is the same approach that Test driven development takes with writing a test that fails first, finding the solution to it, and then refactoring that solution.

Example: Peer Review

Peer review is common across many fields. Whether it be academic journals, books, or programming. The reason editors are so valuable is because they are a third party to a piece of writing and can give objective feedback. In the scientific community this is done by peer reviewing journal articles.

Test driven development is different in that the third party is a program. When someone writes tests it codes the assumptions and requirements and is entirely objective. This feedback can be valuable to test assumptions before someone else looks at the code. It also helps with reducing bugs and feature misses.

This doesn't mitigate the inherent issues with machine learning or math models though; it just defines the process as to how to tackle problems and find a good enough solution.

Risks with machine learning

While the scientific method and TDD are a good start to a development process, we still have issues that we might come across. Someone can follow the scientific method and still have wrong results. TDD just helps us create better code and be more objective.

Specific to machine learning there are four main categories that are challenges:

1. Unstable Data

2. Underfitting
3. Overfitting
4. Unpredictable Future

Unstable data

Machine learning algorithms do their best to avoid unstable data by minimizing outliers, but what if the errors were our own fault? If we are misrepresenting what is correct data, then we will end up skewing our results.

This is a real problem considering the amount of incorrect information we may have. For example, if an API you are using changes from giving you 0 to 1 binary information to -1 to 1, then that could be detrimental to the output of the model. We might also have holes in a time series of data. With this instability, we need a way of testing for data issues to mitigate human error.

Underfitting

Underfitting is when a model doesn't take into account enough information to accurately model real life. For example, we observed only two points on an exponential curve, we would probably assert that there is a linear relationship there. But there may not be a pattern, because there are only two points to reference.

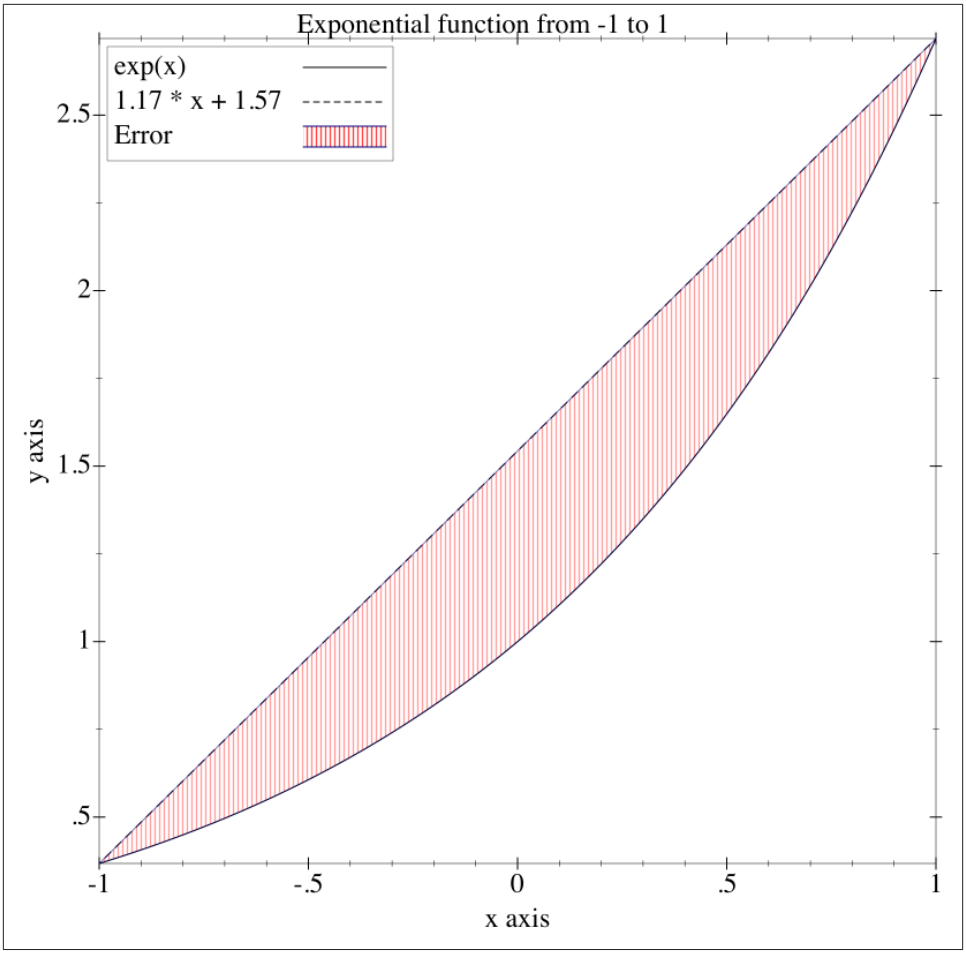


Figure 1-1. In the range of -1 to 1 a linear line will fit an exponential curve well

Unfortunately though when you increase the range you won't see nearly as well of results and instead the error will drastically increase.

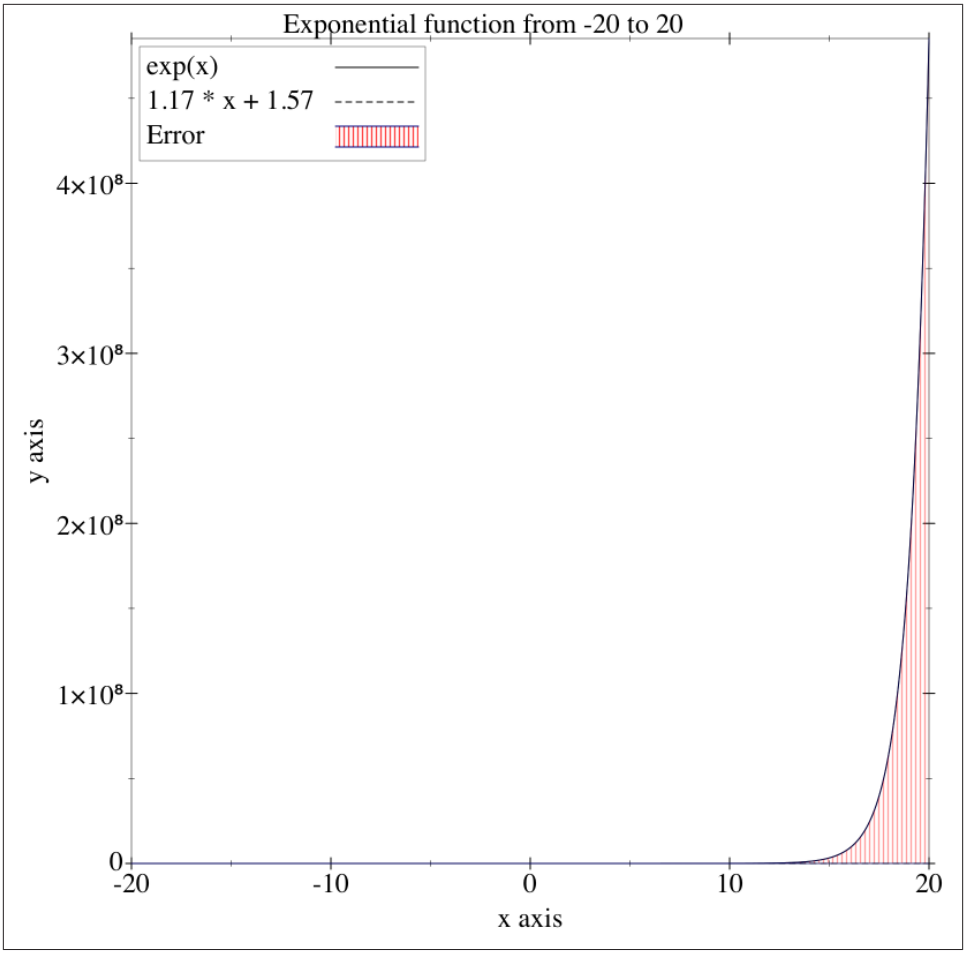


Figure 1-2. In the range of -20 to 20 a linear line will not fit an exponential curve at all

In statistics, there is a measure called power which denotes the probability of not finding a false negative. As power goes up, false negatives go down. However, what influences this measure is the sample size. If our sample size is too small we just don't have enough information to come up with a good solution.

Overfitting

While too little of sample isn't ideal, we also have the risk of overfitting data as well. Using the same exponential curve example, let's say we have three hundred thousand data points. Overfitting the model would be building a function that has three hundred thousand operators in it, effectively memorizing the data. This is possible, but wouldn't perform very well if there was a new data point that was out of that sample.

It seems that the best way to mitigate underfitting a model is to give it more information, but this actually can be a problem as well. More data can mean more noise and more problems. Using too much data and too complex of a model will yield something that works for that particular data set and nothing else.

Unpredictable future

Machine learning is well suited for the unpredictable future since most algorithms learn from new information. But as new information is found, it can also come in unstable ways, and new issues can arise that weren't thought of before. We don't know what we don't know. Sometimes it's hard to tell whether our model is working or not given new information.

What things to test for to reduce risks

Given the fact that we have problems of unstable data, underfitted models, overfitted models, and future resiliency, what should we do? There are some general guidelines that can be written into tests and mitigate the risk.

Heuristics for testing Machine Learning code

Heuristics or rules of thumb can be useful for overcoming common issues. In our case we want to mitigate the risks of unstable data, overfitting, underfitting, and performance in the future. This section introduces some heuristics for ensuring that those issues won't arrise.

Mitigate unstable data with seam testing

Introduced in Working Effectively with Legacy Code, Michael Feathers introduces testing seams when interacting with legacy code. Seam testing simply is the parts of integration between parts of a code base. So in legacy code many times we are given a piece of code we don't know what it does internally but have expectations on what happens when we feed it something. Machine learning algorithms aren't legacy code but they are similar. Like legacy code, machine learning algorithms should be treated like a black box.

Data will flow into a machine learning algorithm and flow out of the algorithm. Those two seams we can test by unit testing our data inputs and outputs to make sure they are valid within our given tolerances.

Example: Seam Testing a Neural Network

Let's say that you would like to test a neural network. You know that the data that is yielded to a neural network needs to be between 0 and 1 and that in your case you want the data to sum up to 1. When data sums up to 1 that means the data is modeling a

percentage of. For instance if you have 2 Widgets and 3 Whirligigs the array of data would be: 2/5 Widgets and 3/5 Whirligigs. Since we want to make sure that we are only feeding information that is positive and adds up to 1 we'd write the following test in our test suite:

```
it 'needs to be between 0 and 1' do
  @weights = NeuralNetwork.weights
  @weights.each do |point|
    (0..1).must_include(point)
  end
end

it 'has data that sums up to 1' do
  @weights = NeuralNetwork.weights
  @weights.reduce(&:+).must_equal 1
end
```

Seam testing serves as a good way to define interfaces between pieces of code. While this is a trivial example, note that the more complex the data gets, the more important these seam tests are. As new programmers touch the code they might not know all the intricacies that you do.

Check fit by cross validating

Cross validation is a method of splitting all of your data into two parts: training and validation. The training data is used to build the machine learning model and the validation data is used to validate that the model is doing what is expected. This increases the ability to find and determine the underlining errors in a model.

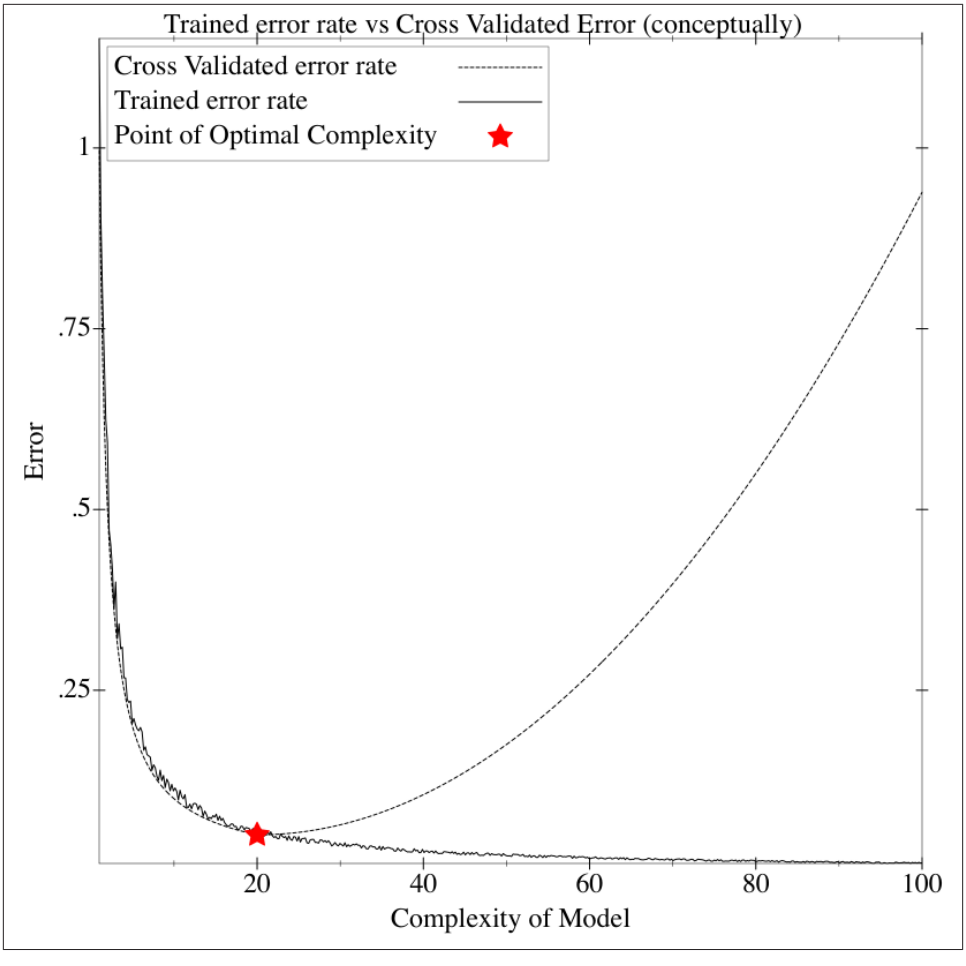


Figure 1-3. Our real goal is to minimize the cross validated error or real error rate

What is Training?

Training is special to the machine learning world. Since machine learning algorithms aim to map previous observations to outcomes training is essential. These algorithms learn from data that has been collected and without an initial set to train on the algorithm would be useless.

Swapping training with validation helps increase the tests. you would do this by splitting the data into two and the first time use set 1 to train and set 2 to validate, then the second test would be swapped. Depending on how much data you have, you could split the data

into smaller sets and cross validate that way. If you have enough data you could split cross validation into an indefinite amount of sets.

In most cases, people decide to split validation and training data in half - one part to train the model and the other to validate that it works with real data. If for instance you are training a language model that takes many parts of speech tagging and trains it using a hidden markov model, we want to minimize the error of the model.

Example: Cross Validation of a Model

From our trained model we might have a 5% error but when we introduce data outside of the model that error might rocket to something like 15%. Which is why it's important to use a data set that is separate. This is as essential to machine learning as double entry accounting is to accounting.

An example in code would be like this:

```
def compare(network, text_file)
  misses = 0
  hits = 0

  sentences.each do |sentence|
    if model.run(sentence).classification == sentence.classification
      hits += 1
    else
      misses += 1
    end
  end

  assert misses < (0.05 * (misses + hits))
end
```

Using the compare function I can have two tests with my two sets of data:

```
def test_first_half
  compare(first_data_set, second_data_set)
end

def test_second_half
  compare(second_data_set, first_data_set)
end
```

This method of first splitting data into two sets eliminates common issues that might happen as a result of the improper parameters on your machine learning model. It's a great way of finding issues before it becomes a part of any codebase.

Reduce overfitting risk by testing the speed of training

Occam's razor is about simplicity when modeling something, stating that the simpler solution is the better one. This directly implies to not overfit your data. The idea that the simpler solution is the better one has to do with how overfitted models generally

just memorize the data that is given to it. If a simpler solution can be found, it will notice the patterns versus parsing out the previous data.

A good proxy for complexity in a machine learning model is how fast it takes to train it. If you are testing different approaches to solving a problem and one takes 3 hours to train while the other takes 30 minutes, generally speaking the one that takes less time to train is probably better. The approach to take would be to wrap a benchmark around the code to find out if it's getting faster or slower over time.

Many machine learning algorithms have max iterations built into them. In the case of Neural Nets, you might set a max epoch of 1000 so that if the model isn't trained within 1000 iterations, the model isn't good enough. An epoch is just a measure of one iteration through all inputs gone through the network.

Example: Benchmark testing

To take it a step further, you can also use unit testing frameworks like MiniTest. This adds computational complexity and an ips benchmark test to your test suite so that the performance doesn't degrade over time.

For a gut check, one can also put a benchmark test around their code.

```
it 'should not run too much slower than last time' do
  bm = Benchmark.measure do
    model.run('sentence')
  end
  bm.real.must_be < (time_to_run_last_time * 1.2)
end
```

Where in this case we don't want the test to run more than 20% over what it did last time.

Monitor for future shifts with Precision and Recall

Precision and recall are a way of monitoring the power of the machine learning implementation. Precision is a metric that monitors the percentage of true positives. For example, a precision of 4/7 would mean that 4 were correct out of 7 yielded to the user. Recall is the ratio of true positives to true positive plus false negatives. In the example, let's say that we have 4 true positives and 9. So recall would be 4/9.

User input is needed to calculate precision and recall. This closes the learning loop and improves data over time due to information feeding back for misclassified information. In the case of Netflix, they do this by displaying a star rating that they predict is what you want. Then when you don't agree with it, they feed that back into their model for future predictions.

Conclusion

Machine learning is a science and requires an objective approach to problems. Just like the scientific method test driven development can aid in solving a problem. The reason that TDD and the scientific method are so similar is because of these three reasons:

TDD and Scientific Method Similarity

1. Both propose that the solution is logical and valid
2. Both share results through documentation and work over time
3. Both work in feedback loops

But while the scientific method and test driven development are similar there are specific issues to machine learning.

Specific Issues to Machine Learning

1. Unstable Data
2. Underfitting
3. Overfitting
4. Future Resiliency

Table 1-2. These can be mitigated through the following heuristics

Problem / Risk	Heuristic
Unstable Data	Test the Seams
Underfitting	Cross Validation
Overfitting	Benchmark Testing (Occam's Razor)
Future Resiliency	Precision / Recall tracking over time

The best part is that all of these can be written and thought of before writing actual code. Test driven development like the scientific method is valuable as a way to approach machine learning problems.

CHAPTER 2

A Quick Introduction to Machine Learning

You've picked up this book interested in Machine Learning. While you probably have a concept as to what Machine Learning is, many times the subject can be defined in somewhat of a vague way. In this quick introduction we'll go over what Machine Learning is, as well as a general framework for thinking about machine learning algorithms. We will talk about the algorithms in this book on a very shallow level but by the end of this you will at least know what to expect from this book in terms of Machine Learning itself.

What is Machine Learning?

Machine learning is the intersection between theoretically sound computer science and practically noisy data. In its essence it's about making sense out of data in much the same way that humans do.

Machine learning is a type of artificial intelligence where by an algorithm or method will determine patterns out of data. Generally speaking there are a few problems Machine Learning tackles.

Table 2-1. The Problems of Machine Learning

The Problem	Machine Learning category
Fitting some data to a function or function approximation	Supervised Learning
Figuring out what the data is without any feedback	Unsupervised Learning
Playing a game with rewards and payoffs	Reinforcement Learning

Supervised Learning

Supervised learning or function approximation is simply fitting data to a function of any variety. For instance given the noisy data below you can fit a line that generally approximates it.

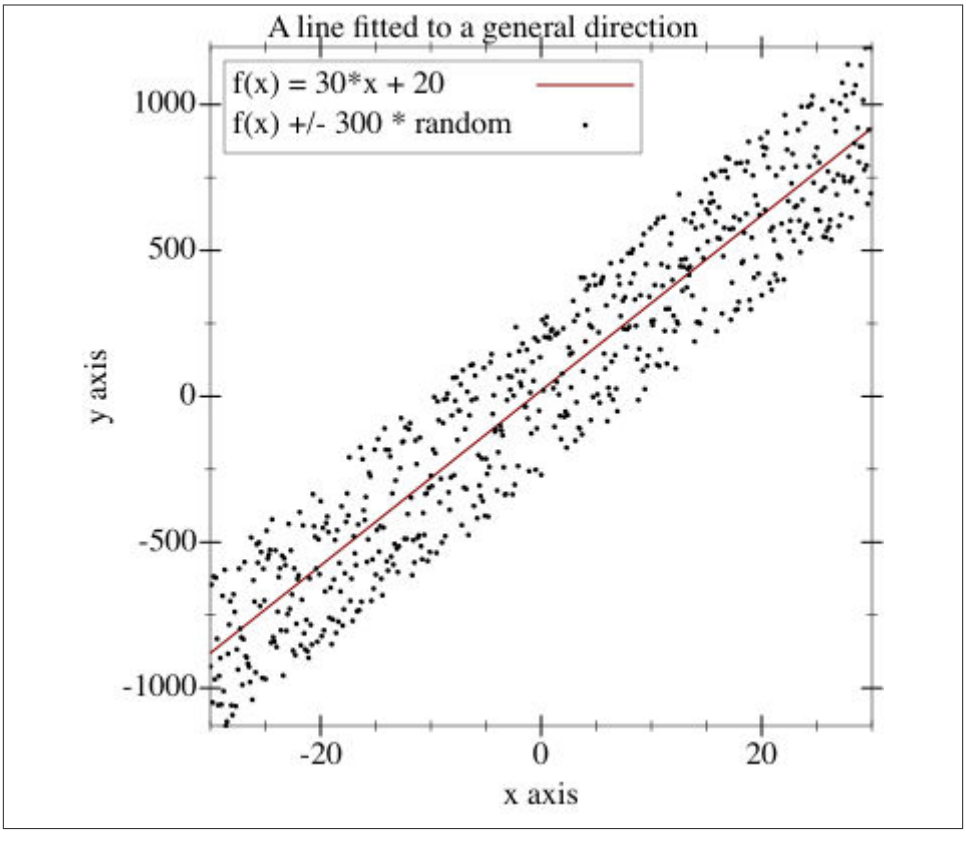


Figure 2-1. This shows a line fitted to some random data

Unsupervised Learning

Unsupervised learning is about figuring out what makes the data special. For instance if we were given many data points we could group them due to similarity, or perhaps determine which variables are better than others.

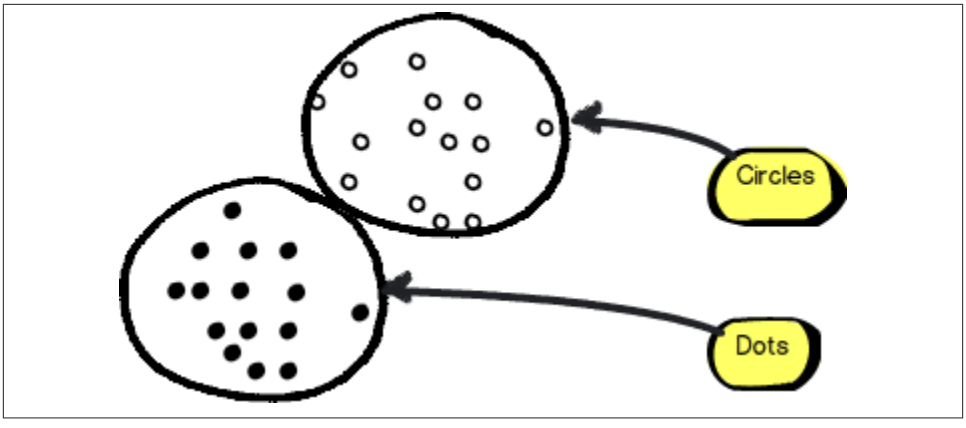


Figure 2-2. Clustering which is unsupervised learning is a common example and looks like this:

Reinforcement Learning

Reinforcement learning is the subtopic about figuring out how to play a multistage game with rewards and payoffs. Think of it like the algorithms that optimize life of something. A common example of a reinforcement learning algorithm is a mouse trying to find cheese in a maze. For the most part the mouse gets zero reward until it finally finds the cheese.

We will discuss supervised and unsupervised learning in this book but skip reinforcement learning. In the last chapter I include some resources that you can check out if you'd like to learn more about reinforcement learning.

What can Machine Learning Accomplish?

Really what makes Machine Learning unique is it's ability to optimally figure things out. But each machine learning algorithm has it's quirks and tradeoffs. Some do better than others. In this book we will be covering quite a bit of algorithms and each has their benefits and consequences. To help you in navigating the book the following is a matrix to help you understand how each chapter will help or not.

Table 2-2. Machine Learning Algorithm Matrix

Algorithm	Type	Class	Restriction Bias	Preference Bias
KNN	Supervised Learning	Instance Based	Generally speaking KNN is good for measuring distance based approximations, it suffers from the curse of dimensionality	Prefers problems that are distance based

Algorithm	Type	Class	Restriction Bias	Preference Bias
Naïve Bayes	Supervised Learning	Probabilistic	Works on problems where the inputs are independent from each other	Prefers problems where the probability will always be greater than zero for each class
SVM	Supervised Learning	Decision Boundary	Works where there is a definite distinction between two classifications	Prefers binary classification problems
Neural Networks	Supervised Learning	Non linear functional approximation	Little restriction bias	Prefers binary inputs
(Kernel) Ridge Regression	Supervised	Regression	Low restriction on problems it can solve	prefers continuous variables
Hidden Markov Models	Supervised / Unsupervised	Markovian	Generally works well for system information where the markov assumption holds	Prefers timeseries data and memoryless information
Clustering	Unsupervised	Clustering	No restriction	Prefers data that is in groupings given some form of distance (Euclidean, Manhattan, or others).
Filtering	Unsupervised	Feature Transformation	No restriction	Prefer data to have lots of variables to filter on

Refer to this matrix throughout the book to understand how things relate against each other.

Machine Learning is only as good as what it applies to so let's get to implementing some of these algorithms!

Before we start make sure you have Ruby installed and ready to go

Before we get started you will need to get Ruby installed. Go to *https://www.ruby-lang.org/en/* and install Ruby. This book was tested using Ruby 2.1.2 but things do change rapidly in the Ruby community. All of those changes will be annotated in the coding resources which will are available on GitHub.

Mathematical Notation used throughout the book

Table 2-3. This book uses mathematics to solve problems but all of the examples attached are programmer centric. Throughout the book we use the following notations:

Symbol	How do you say it?	What does it do?	
$\sum_{i=0}^{2} 0x_i$	The sum of all x's from x_0 to $x_2 0$	This is the same thing as $x_0 + x_1 + \cdots + x_2 0$	
latexmath:[$	x	$]	The absolute value of X
This takes any value of x and makes it positive. So $x = -1$ would equal 1 and $x = 1$ would equal 1 as well		$\sqrt{4}$	The square root of 4
This is the opposite of 2^2		$z_k = < 0.5, 0.5 >$	Vector z_k equals 0.5 and 0.5
This is a point on the xy coordinates and is denoted as a vector which is a group of numerical points		$log_2(2)$	Log 2
This solves for i in $2^i = 2$		$P(A)$	Probability of A
In many cases this is the count of A divided by the total occurances		latexmath:[$P(A	B)$]
Probability of A given B	This is the probability of A and B divided by the probability of B		$1, 2, 3 \cap 1$
The intersection of set one and two	This turns into a set called ${1}$		$1, 2, 3 \cup 4, 1$
The union of set one and two	This equates to 1, 2, 3, 4		$det(C)$
The determinant of the matrix C	This will help determine whether a matrix is invertible or not		$a \propto b$
A is proportional to b	This means that $ma = b$		$\min f(x)$
Minimize $f(x)$	This is an objective function to minimize the function $f(x)$		X^T

CHAPTER 3
K-Nearest Neighbor Classification

I'm sure you know of someone who really likes a certain brand. Whether it be a certain technology company, a certain clothing brand we all have a certain affinity for certain brands. Usually you can detect this by what they wear, talk about and interact with. But what about a different way of approaching this problem. How would we determine a certain brand affinity?

Intuitively in an e-commerce site we would do this through looking at previous orders of similar users and see what they buy. So for instance let's assume that a user has a history of orders, each including two items.

Image showing user with a history of orders of multiple brands.

Based on their previous orders this user buys a lot of *Milan Clothing Supplies* (not real but you get the picture). Matter of fact out of the last 5 orders, they have bought 5 *Milan Clothing Supplies* shirts. You could say they have a certain affinity towards this company. Knowing this if we pose the question of what brand this user is particularly interested in they would be at the top.

This general idea is known as the K-Nearest Neighbor Classification algorithm. In our case K equals 5 and for each order we are voting on brands. Whatever gets the highest vote is our classification. This chapter will introduce and work through what the K-Nearest Neighbor (KNN) classification is as well as work through a code example that detects whether the face has glasses on and whether they have facial hair.

K-Nearest Neighbor Classification in Brief

K-Nearest Neighbor Classification is an instance based supervised learning method that works well with distance sensitive data. It suffers from the curse of dimensionality and other problems with distance based algorithms.

History of K-Nearest Neighbor Classification

The K-Nearest Neighbor (KNN) algorithm was originally introduced by Evelyn Fix, PhD. and JL Hodges Jr., PhD. in an unpublished technical report written for the United States Air Force medicine school. Their original research focused on splitting up classification problems into a few subproblems:

1. Distributions F and G are completely known.
2. Distributions F and G are completely known except for a few parameters.
3. F and G are unknown, except possibly for existence of densities, etc.

They pointed out that if you know the distributions of two classifications or you know what the distribution is minus some parameters you can easily back out useful solutions. Instead they focused their work on the more difficult case of finding classifications amongst distributions that are unknown. What they came up with set forth the original path for the KNN algorithm.

This algorithm has been shown to have no worse than twice the Bayes error rate as data approaches infinity. This means that as entities are added to your data set the error will be no worse than a Bayes error rate. Also being such a simple algorithm this is easier to implement as a first stab at a classification problem. In many cases it works good enough.

[Cover TM, Hart PE (1967). "Nearest neighbor pattern classification". IEEE Transactions on Information Theory 13 (1): 21–27. doi: 10.1109/TIT.1967.1053964.] *http://www.dtic.mil/dtic/tr/fulltext/u2/a800276.pdf*

The one thing that is tough though is how arbitrary it can seem. How do you pick K? How do you determine what is a neighbor and what isn't? These are questions we'll aim to answer in the next couple sections.

House happiness based on a neighborhood

Imagine you are looking to buy a new house. You are looking at two different houses and want to figure out whether the neighbors are happy or not. Of course you don't want to move into an unhappy neighborhood. You go around asking houses whether they are happy where they are at and collect the following information.

We're going to use coordinate minutes since we want to make this specific to a small enough neighborhood.

Table 3-1. House happiness

Latitude Minutes	Longitude Minutes	Happy?
56	2	Yes
3	20	No
18	1	Yes
20	14	No
30	30	Yes
35	35	Yes

The two houses we are interested in are at (10, 10) and (40,40). So which house is happy and which one is not? One method of finding this out would be to use the nearest neighbor and determine whether they are happy or not. Nearest in this sense is absolute distance which is also known as the Euclidean distance.

The Euclidean distance for a two dimensional point like we have above would be $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

In Ruby what would this look like?

```
require 'matrix'

# Euclidean distance between two vectors v1 and v2
# Note that Vector#magnitude is the same thing as the Euclidean distance
# from (0,0,....) to the vector point.
distance = ->(v1, v2) {
  (v1 - v2).magnitude
}

house_happiness = {
  Vector[56, 2] => 'Happy',
  Vector[3, 20] => 'Not Happy',
  Vector[18, 1] => 'Happy',
  Vector[20, 14] => 'Not Happy',
```

```
  Vector[30, 30] => 'Happy',
  Vector[35, 35] => 'Happy'
}

house_1 = Vector[10, 10]
house_2 = Vector[40, 40]

find_nearest = ->(house) {
  house_happiness.sort_by {|point,v|
    distance.(point, house)
  }.first
}

find_nearest.(house_1) #=> [Vector[20, 14], "Not Happy"]
find_nearest.(house_2) #=> [Vector[35, 35], "Happy"]
```

Based on this reasoning you can see that the nearest neighbor for the first house is not happy while the second house neighbor is. But what if we increased the number of neighbors we looked at?

```
# Using same code from above in this as well

find_nearest_with_k = ->(house, k) {
  house_happiness.sort_by {|point, v|
    distance.(point, house)
  }.first(k)
}

find_nearest_with_k.(house_1, 3)
#=> [[Vector[20, 14], "Not Happy"], [Vector[18, 1], "Happy"], [Vector[3, 20], "Not Happy"]]
find_nearest_with_k.(house_2, 3)
#=> [[Vector[35, 35], "Happy"], [Vector[30, 30], "Happy"], [Vector[20, 14], "Not Happy"]]
```

Using more neighbors shows that the classification hasn't changed! This is a good thing and increases the confidence in the classification. This method is what the K Nearest Neighbors classification is. More or less we take the *nearest* neighbors and use their attributes to come up with a voting. In this case we wanted to see whether one house would be happy over the other. It can really be anything.

K Nearest Neighbors is an excellent algorithm because it is so simple as you see above. It is also extremely powerful as well. It can be used to classify or regress data.

Classification and Regression

Note that we are mainly looking for whether the house is happy or not. Instead of trying to value the happiness we are mainly looking for whether it meets our criteria or not. This is called a classification problem and can take many forms.

Many times classification problems are binary meaning that they only have two answers. Good or bad, true or false, right or wrong are all binary. A lot of problems distill into this question.

On the other hand we could look for a numerical value of happiness for the house but that would be called a regression problem which we won't do in this chapter. We will return to this later when we talk about Kernel Ridge Regressions. The idea there is fitting a function to calculate a continuous answer.

This chapter will cover quite a bit and is broken out into a few different concerns when using K Nearest Neighbors. We will first discuss picking K, or the constant that determines your neighborhood, then delve into what nearest means and then finally follow up with an example of facial classification using OpenCV.

How do you pick K?

In the case of figuring out house value we implicitly picked a K of 5. This works well for the case where we are making a quick judgement based on their most recent purchases. But for bigger problems we might not have the ability to guess.

K in the K-Nearest Neighbor algorithm is just an arbitrary number generally ranging from one to the number of data points. With that much of a range you might think that it's difficult to pick out the optimal K, but in practice it's not quite a vast decision.

There are three methods for picking out K

1. Guessing
2. Using a heuristic
3. Optimizing using an algorithm

Guessing a K

Guessing is the easiest solution. In our case of classifying brand ambassadors into groups we just pick 11 as a good guess. We have knowledge that 11 orders is probably good enough to get a good idea of how a person shops.

Many times when we are approaching a problem we can qualitatively come up with a good enough K to solve the problem so guessing will work. Though if you want to be more scientific about it there are a few heuristics that can help.

Heuristics for picking K

There are three heuristics that can help you in determining an optimal K for a K-Nearest Algorithm.

. . Avoid K that is even when there are only two classes to classify . Choose a K that is greater or equal to the number of classes + 1 . Choose a K that is low enough to avoid noise

Use coprime class and K combinations

Picking coprime numbers of class and K will ensure less ties. Coprime nubmers are simply two numbers not sharing any common divisor except for 1. So for instance 4 and 9 are coprime while 3 and 9 are not.

Imagine the case where you have two classes *good* and *bad*. If we were to pick a K of 6, which is even, then we might run into the situation of having ties. Graphically it looks like this:

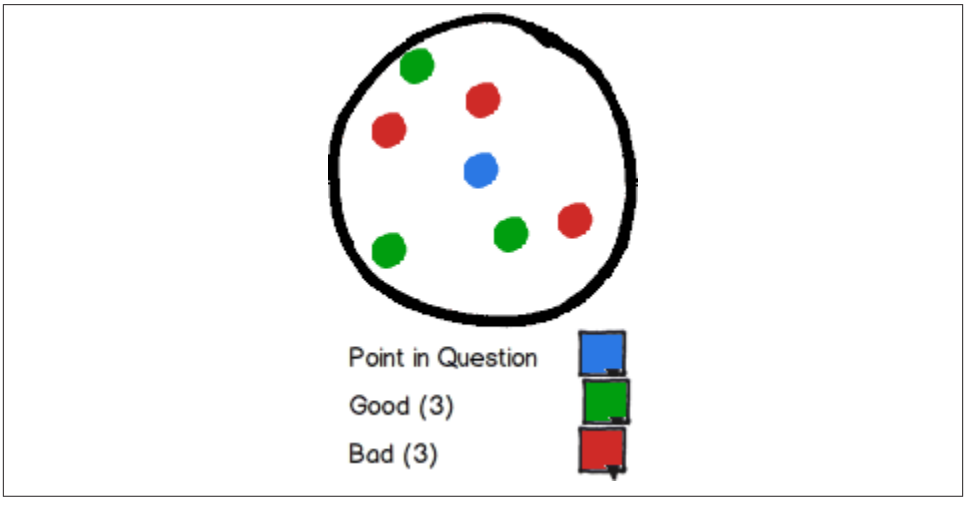


Figure 3-1. Tie with K=6 and two classes

Instead if you were to pick a K of 5 there wouldn't be a tie.

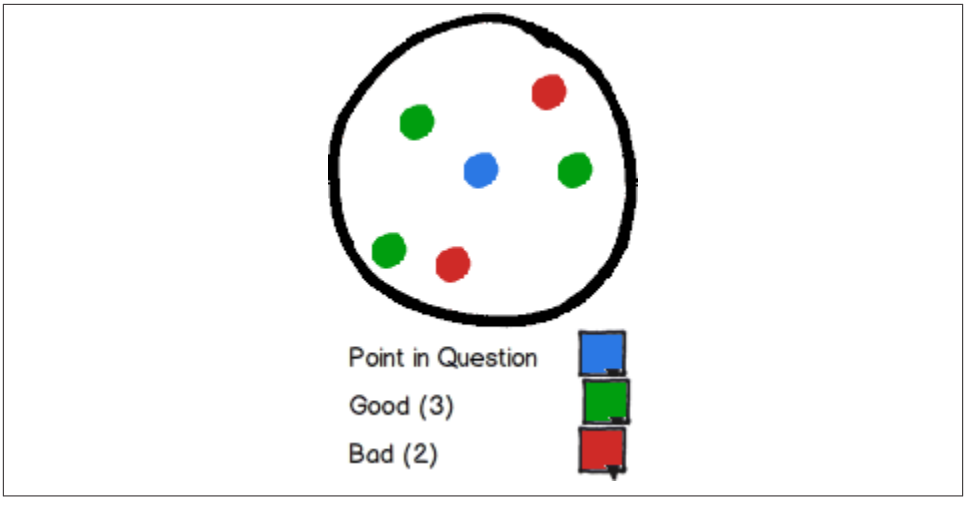


Figure 3-2. K=5 with two classes and no tie

Choose a K that is greater or equal to the number of classes + 1

Imagine the case where there are 3 classes: lawful, chaotic, and neutral. A good heuristic is to pick a K that is at least 3 because anything less will mean that there is no chance that each class gets a chance to show up.

For instance imagine the case of K=2:

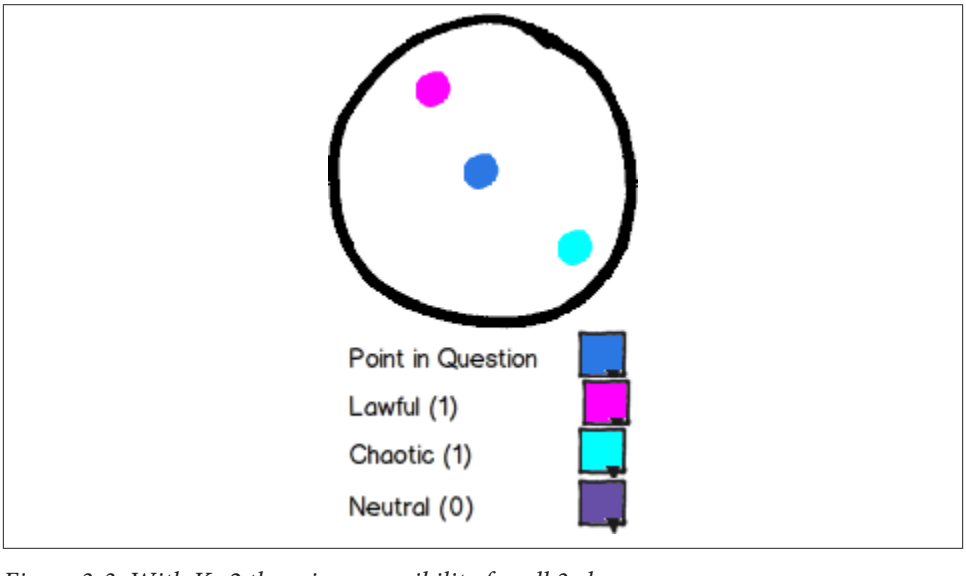


Figure 3-3. With K=2 there is no possibility for all 3 classes

Note how there is only two classes that get the chance to be used. Instead we will need to use at least K=3. But based off of what we found in the first heuristic ties are not a good thing. Instead of K=3 we should use K=4.

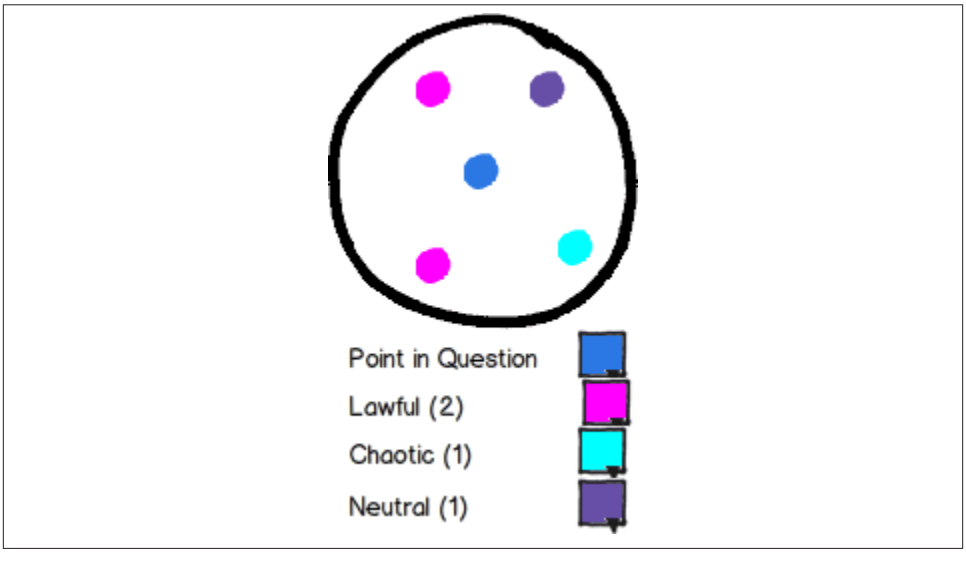


Figure 3-4. With K set greater than the amount of classes there is a chance for all classes

Choose a K that is low enough to avoid noise

As K goes up you eventually approach the size of entire data set. If you were to pick the entire dataset what would happen is you would vote whatever the mode of the dataset is. If for instance in our brand affinity example example you have 100 orders and they are as follows.

Table 3-2. Brands ordered

Brand	Count
Widget Inc	30
Zoomba	23
Robots and Rockets	12
Ion 5	35
Total	100

If we were to set K=100 then our answer will always be *Ion 5* because that is the distribution of the order history. That is not really what we want, we want to determine instead what the most recent order affinity is like. More specifically we want to minimize the amount of noise that comes into our classification. Without coming up with a specific

algorithm for this we can justify K being set to a much lower rate of something like K=3 or K=11 even.

Algorithms for Picking K

Picking K can be somewhat qualitative and non-scientific and that's why there are many algorithms out there showing how to optimize K over a given training set. There are many approaches ranging from genetic algorithms, brute force, to grid searches.

In many cases people assert that K should be determined due to *domain knowledge* that you have as the implementor. For instance if you know that 5 is good enough you can pick that.

This problem is actually called a *hill climbing* problem where you are trying to minimize error based on an arbitrary K. The idea is to iterate through a couple of possible K's until you find a suitable error. The difficult part about finding a K using an algorithm like genetic algorithms or brute force is that as K increases the complexity of the classification also increases and slows down. So as you increase K the program actually gets slower.

If you want to learn more about genetic algorithms applied to finding an optimal K you can read more about it here: *http://pubs.acs.org/doi/abs/10.1021/ci060149f*

Personally I think iterating through 2 through 0.01 * sample size is good enough. You should have a decent idea of what works and what doesn't just by experimenting with different K's.

What makes a neighbor near?

Imagine for one second that you are sitting on the corner of a city block. How far is it to go from one corner to the opposite corner?

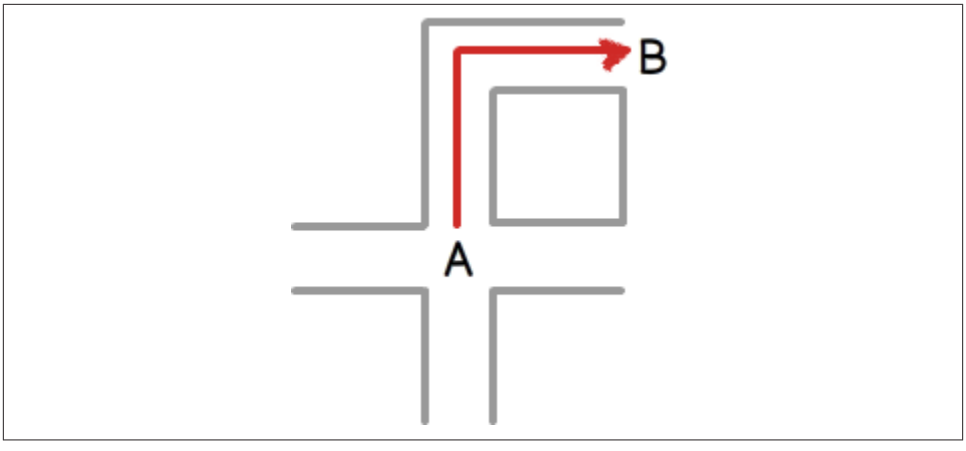


Figure 3-5. Driving from Point A to Point B on a City Block

The answer depends on your constraints: are you able to jump over fences on foot, or do you have to drive. If you were to drive you'd be traveling two times the length of the city block whereas if you walked straight it'd be latextmath:[$\sqrt{2x^2}$] where x is the length of a city block. Assuming that a city block is 250 feet (76.2 meters) long we could say that driving would take 500 feet (152.4 meters). Whereas if we were able to travel straight it'd be roughly 353.5 feet (107.75 meters).

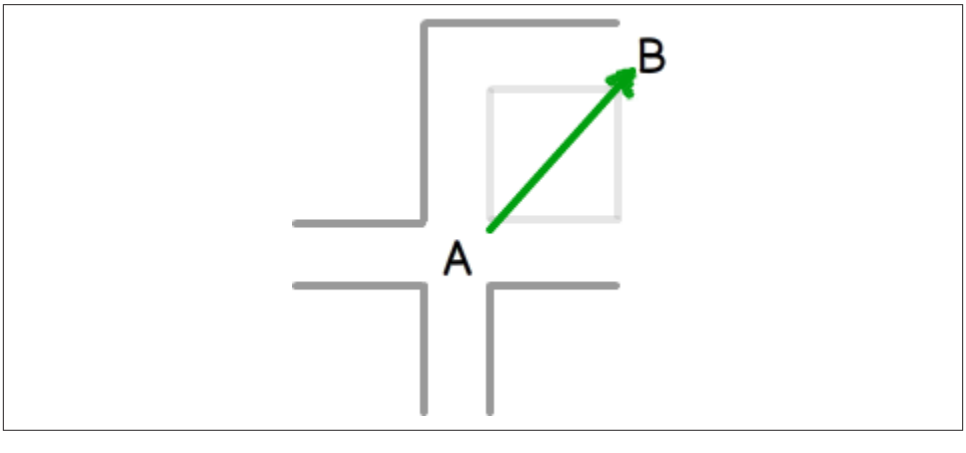


Figure 3-6. Straight line between A and B

What we've defined above you probably remember from geometry class. The Pythagorean theorem states that the length of a hypotenuse is $\sqrt{a^2 + b^2}$.

In modern mathematics terms these are both called metrics. They are a measure of how far points are from each other. These metrics are calculated using a distance function

which in our case above was the Taxicab distance and Euclidean distance. There are many ways of measuring distance ranging from the Taxicab to Chebyshev distances. The way you measure distance is essential in understanding how the KNN algorithm works because it is based on proximity of data. For the most part Euclidean distances are commonly used and is the shortest path between two points.

Minkowski Distance

A generalization of Euclidean and Taxicab distances is called the Minkowski distance. To understand the Minkowski distance let's first look at what the Taxicab distance function looks like:

$$d_{taxicab}(x, y) = \sum_{i=1}^{n} \left| x_i - y_i \right|$$

This function takes the absolute differences between all dimensions of the points x and y. Now let's look at the Euclidean distance function.

$$d_{euclid}(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Note that squaring something will always yield a positive number and that $\sqrt{x} = x^{\frac{1}{2}}$. So we could rewrite this to be:

$$d_{euclid}(x, y) = \left(\sum_{i=1}^{n} \left| x_i - y_i \right|^2 \right)^{\frac{1}{2}}$$

This is very similar to the Taxicab distance function above and in fact Minkowski generalizes this to the following:

$$d(x, y) = \left(\sum_{i=1}^{n} \left| x_i - y_i \right|^p \right)^{\frac{1}{p}}$$

Introducing a new parameter p we can build the Taxicab distance function using $p = 1$ and Euclidean distance using $p = 2$. This is intriguing because if we needed we could up our p as much as we need to. While we won't get into the application of all versions of a Minkowski distance this gives you the option to study this more.

Mahalanobis Distance

One problem with the Minkowski type distance functions is that they assume that data should be symmetric in nature. That distance is the same on all sides.

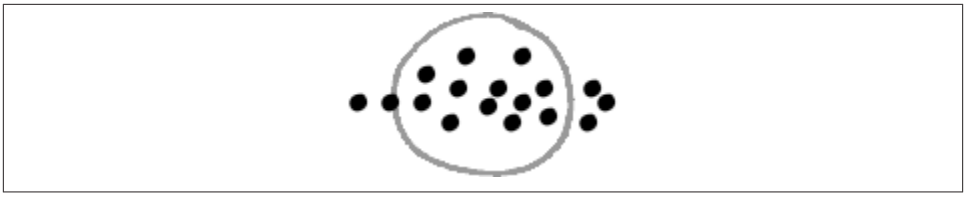


Figure 3-7. With Squashed Data Minkowski distances don't work as well

A lot of times data is not spherical in nature or well suited for symmetric distances like the Minkowski distances.

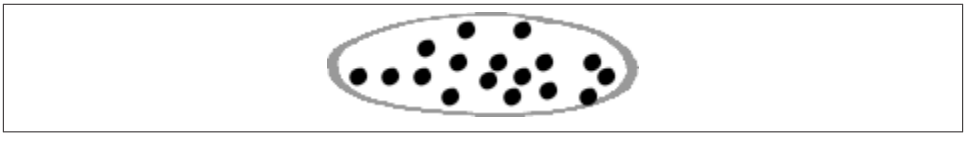


Figure 3-8. Using the Mahalanobis distance

Instead we should take into consideration the ellipsoidal nature of the data above. Instead of drawing a perfect circle around the data we would like to figure something out that is better suited for the variability in the data. The Mahalanobis distance function takes into consideration a volatility about each dimension. So for each dimension of the data there is a certain variable s_i which is the standard deviation of that set.

$$d(x,\ y) = \sqrt{\sum_{i=1}^{n} \frac{(x_i\ -\ y_i)^2}{s_i^{\ 2}}}$$

Determining classes

Classes can be quite arbitrary. Sometimes things aren't as mutually exclusive as we first think. So a caveat with building K-Nearest Neighbor classification tools is that as attributes that you are modeling goes up the classes go up exponentially.

For example: if we have two attribute colors: *red*, *yellow* we end up with the classes *red*, *yellow*, *orange*.

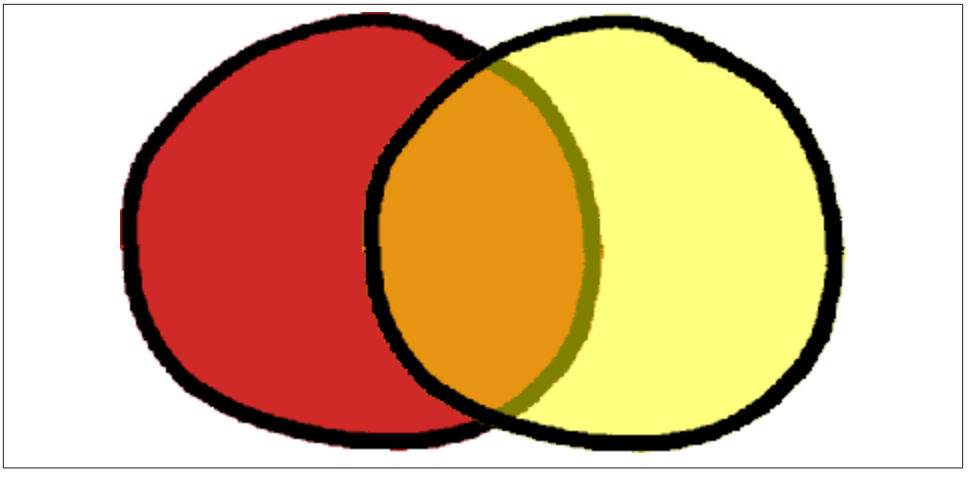


Figure 3-9. Mix of two classes

Now if we add *blue* then we'll have *red*, *yellow*, *blue*, *green*, *burnt sienna*, *orange*, *purple*. In the first case our two attributes yielded 3 classes, while in the second case our 3 attributes yielded 7 classes.

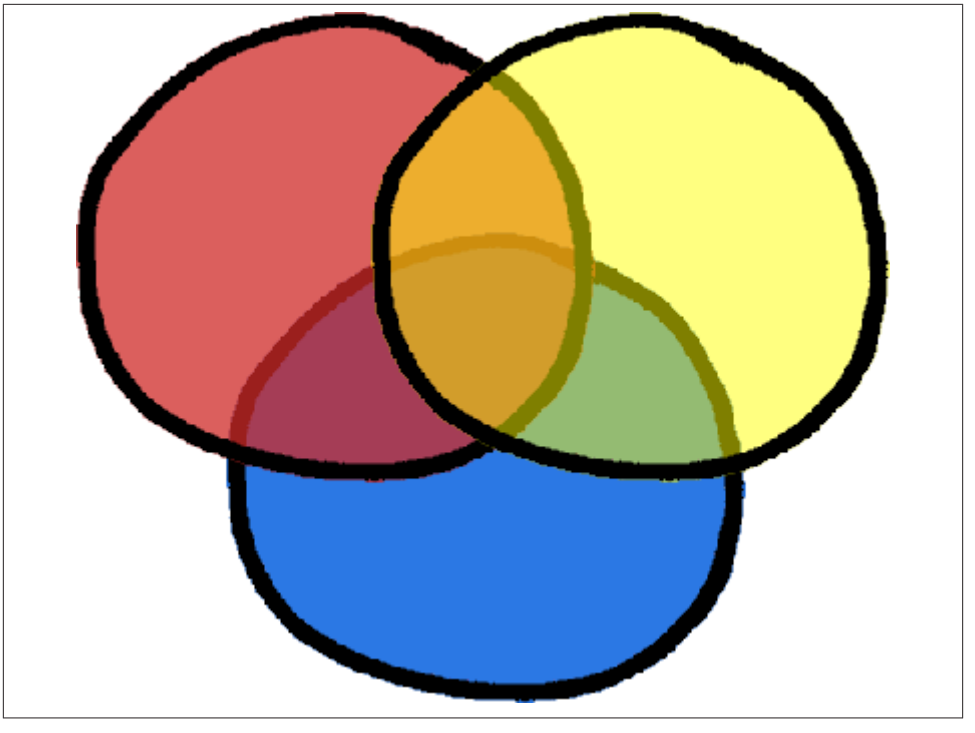


Figure 3-10. Mix of three classes

The general case of combining attributes into mutually exclusive classes is $n2^n$ - 1 where n is the amount of attributes you are adding mixing.

Unless there's a strong argument for taking out mixed classes this is an important distinction to make. If you have 4 attributes you can assume that there are 15 classes you need to take into consideration and set your K to at least 16.

The Curse of Dimensionality

K-Nearest Neighbors has one downside which is called the *Curse of Dimensionality*. This curse is the effect that high dimensional data tends to be sparse and far apart. Imagine a shotgun blast where over time the pellets expand through the air. This problem is common in algorithms which are based on locality and the ability to determine how close something is.

This figure shows how a unit sphere (the distance from 0,0,0 to the edge is exactly 1) when shrunken onto a 2 dimensional plane actually gets a slightly less average distance to points. The opposite happens when expanding into more dimensions:

Figure 3-11. Curse of Dimensionality on a Sphere

There is no way of avoiding this with regards to the KNN algorithm itself and has to be mitigated through other means. The other means we'll be covering in a subsequent chapter.

We will also discuss the curse of dimensionality in Chapter 9 and 10.

Example: Beard and Glasses Detection using KNN and OpenCV

Suppose that we wanted to determine with a general accuracy whether someone had facial hair and whether they were wearing glasses. How would we do such a thing? We really don't have a lot of information about the distribution of any of this data so KNN will be a good algorithm to use.

In this section we will attempt to do just that with the K-Nearest Neighbor algorithm along with help from OpenCV. First we'll explain what the program's class diagram looks like. From there we'll delve deeper into how to convert a raw image into an avatar. Then we'll extract features out of those avatar images. When we have enough features we'll then use KNN to build a neighborhood of faces which will help us determine attributes of input images.

Setup Notes

All of the code that we are using for this example can be found on Github at *https://github.com/thoughtfulml/examples/tree/master/2-k-nearest-neighbors*.

Since Ruby is constantly changing the README should be the most up to date instructions on getting these code examples working.

You will have to install imagemagick, OpenCV and a recent Ruby version to get started.

The Class Diagram

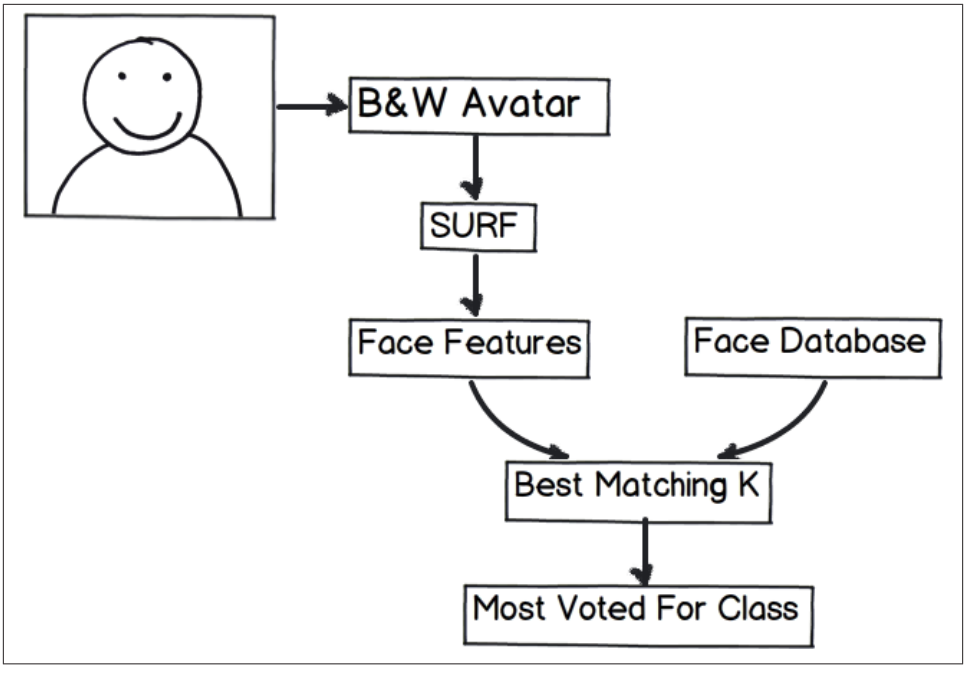


Figure 3-12. Class Diagram for our Facial Hair and Glasses Detector

The general idea is to take a raw image (Image), extract out a smaller avatar image (Face) and then put all the Face classes inside of a Neighborhood which is faces that have been annotated with information.

Raw Image to Avatar

The Image class which takes a picture of a human does one thing which is take a raw image and try to find a face inside of it. To accomplish this we rely on OpenCV. What we want is something like this.

Figure 3-13. Raw image that gets extracted

Figure 3-14. Extracted avatar from Haar Classifier

Knowing a bit about OpenCV we know that we can achieve this by using a Haar-like feature to extract out what looks like a face. We use data that is given by the OpenCV library and rely on their implementation to accomplish this.

This data is freely available and not actually made by me. Instead someone trained this data set with facial images to figure out what looks like a face and what doesn't. It extracts out some features on that face that was built by others.

OpenCV and Haar-like features

OpenCV (Open Computer Vision) is a tool that can achieve extracting faces out of a bigger image using something called a Haar-like features which is just a rectangle around something we want. For more information on OpenCV check out these resources:

OpenCV resources

1. Mastering OpenCV Practical Computer Projects *http://www.amazon.com/Mastering-OpenCV-Practical-Computer-Projects/dp/1849517827/ref=sr_1_1?ie=UTF8&qid=1386900437&sr=8-1&keywords=OpenCV*
2. The documentation is great as well: *http://docs.opencv.org*

Haar-like features is a way of finding a rectangle of a certain kind of feature in an image. Since based on previous information we know that faces generally are of a certain look rather than background information Haar-like training information can be used to determine the rectangle where the face exists.

To determine whether our avatars are correct we use pHash. Unlike MD5 or SHA1 which are cryptographic hashes, This *perceptual* hash uses hamming distances to find close matches. So even though the photo is slightly off it'll still be a duplicate.

```
# test/lib/image_spec.rb
require 'spec_helper'

describe Image do
  it 'tries to convert to a face avatar using haar classifier' do
    @image = Image.new('./test/fixtures/raw.jpg')
    @face = @image.to_face

    avatar1 = Phashion::Image.new("./test/fixtures/avatar.jpg")
    avatar2 = Phashion::Image.new(@face.filepath)

    assert avatar1.duplicate?(avatar2)
  end
end
```

This will fail as you could imagine because *@image.to_face* does nothing and *@face* doesn't have a filepath associated with it.

To fill in the gaps let's try the following:

```
# lib/image.rb
# Stub out Face for now
Face = Struct.new(:filepath)

class Image
  HAAR_FILEPATH = './data/haarcascade_frontalface_alt.xml'
  FACE_DETECTOR = OpenCV::CvHaarClassifierCascade::load(HAAR_FILEPATH)
```

```
  attr_reader :filepath

  def initialize(filepath)
    @filepath = filepath
  end

  def self.write(filepath)
    yield
    filepath
  end

  def face_region
    @image = OpenCV::CvMat.load(@filepath, OpenCV::CV_LOAD_IMAGE_GRAYSCALE)
    FACE_DETECTOR.detect_objects(@image).first
  end

  def to_face
    name = File.basename(@filepath)
    outfile = File.expand_path("../../public/faces/avatar_#{name}", __FILE__)

    self.class.write(outfile) do
      image = MiniMagick::Image.open(@filepath)
      image.crop(crop_params)
      image.write(outfile)
    end

    Face.new(outfile)
  end

  def x_size
    face_region.bottom_right.x - face_region.top_left.x
  end

  def y_size
    face_region.bottom_right.y - face_region.top_left.y
  end

  def crop_params
    crop_params = <<-EOL
      #{x_size - 1}x#{y_size-1}+#{top_left.x + 1}+#{top_left.y + 1}
    EOL
  end
end
```

At this point you can see we're using two libraries MiniMagick and OpenCV, as well as haarcascade_frontalface_alt.xml which is a training set from the OpenCV library that detects faces. Our tests should pass and we can assume that Image works as expected. But we need to now focus on making a Face class which is not just a Struct.

The Face Class

The Face class has one responsibility which is to load a avatar image and extract features out of it. These features will then interface with the Neighborhood class which we will get to in the next section. This is where things become more difficult because features can be extracted in many different ways.

Features vs Dimensions vs Instances

Machine Learning uses the terms features, dimensions, and instances often.

Features are a property of a given data set. Generally they are important combinations of dimensions which are just the different sections of the data. Lastly we have instances which are a specific instance of data.

A good way of thinking about features vs dimensions would be lighting in a room. Let's say you have 3 lights. There are 8 possible combinations of how the lightsare in the room. But you really just want to know if the room is light enough or not (which means there are at least 2 lights on).

Table 3-3. Lighting for the room

Light 1 On?	Light 2 On?	Light 3 On?	Light Enough?
No	No	No	No
Yes	No	No	No
No	Yes	No	No
No	No	Yes	No
Yes	Yes	No	Yes
No	Yes	Yes	Yes
Yes	No	Yes	Yes
Yes	Yes	Yes	Yes

In this case we have one feature which is whether the room is light enough or not which are based on the three dimensions of the lights being on or off. The instances are just the combinations of lights.

Some ways include:

1. Extracting out the shades of each pixel (this is a grayscale image so no colors)
2. Using SIFT
3. Using SURF

Extracting out the shades of the image into a pixel by pixel matrix would be the naive approach and most likely would have us run into the curse of dimensionality. The na-

ivety would come from not reducing the amount of noisy pixels that mean nothing to us. Some algorithms would be able to handle grayscale inputs like for instance a Neural Network which we will get to in an upcoming chapter. Deep learning for instance takes this benefit of neural networks and is able to find features out of grayscale pixels.

Another method would be using an algorithm called SIFT (Scale Invariant Feature Transform). This algorithm which was built by David Lowe in 1999 is a computer vision algorithm for detecting features that are of importance. This algorithm also happens to be patented by the University of British Columbia. This is a big improvement over pixel by pixel matrices.

There is a similar algorithm called SURF (Speeded Up Robust Features). This was proposed by Herbert Bay in 2006 and is an improvement over SIFT because it is quite fast. This algorithm has been shown to be quite successful at recognizing many features of images and therefore will be our choice.

Either SIFT or SURF would be good choices to go with. OpenCV has a good implementation of SURF so that is what we will use to extract our features out of avatars.

Testing the Face Class

SURF gives us two pieces of information about the face: key points, and descriptors. Key points are points in 2-d space (x,y pairs). Descriptors are more interesting because they contain either a 64 or 128 wide vector of descriptors about the feature. Instead of testing how well the SURF algorithm detects features we need to ensure that the data is the same always. Meaning that two Face classes should have the same features extracted.

A test for this would look like

```
# test/lib/face_spec.rb
require 'spec_helper'
require 'matrix'

describe Face do
  let(:avatar_path) { './test/fixtures/avatar.jpg' }

  it 'has the same descriptors for the exact same face' do
    @face_descriptors = Face.new(avatar_path).descriptors
    @face2_descriptors = Face.new(avatar_path).descriptors

    @face_descriptors.sort_by! { |row| Vector[*row].magnitude }
    @face2_descriptors.sort_by! { |row| Vector[*row].magnitude }

    @face_descriptors.zip(@face2_descriptors).each do |f1, f2|
      assert (0.99..1.01).include?(cosine_similarity(f1, f2)),
        "Face descriptors don't match"
    end
  end

  it 'has the same keypoints for the exact same face' do
```

```
    @face = Face.new(avatar_path)
    @face2 = Face.new(avatar_path)

    # This is purely because Ruby's implementation of OpenCV doesn't
    # Have a representation of == for SurfPoints :(
    @face.keypoints.each_with_index do |kp, i|
      f1 = Vector[kp.pt.x, kp.pt.y]
      f2 = Vector[@face2.keypoints[i].pt.x, @face2.keypoints[i].pt.y]

      assert (0.99..1.01).include?(cosine_similarity(f1,f2)),
        "Face keypoints do not match"
    end
  end
end
```

Cosine Similarity

You'll notice that we're using a function called cosine_similarity here. This is a good way of determining how close something is. Instead of relying on the data being 100% equal we can compare two vectors to be close enough.

Inside of the spec_helper file to implement this method we would write:

```
# test/spec_helper.rb
def cosine_similarity(array_1, array_2)
  v1 = Vector[*array_1]
  v2 = Vector[*array_2]
  v1.inner_product(v2) / (v1.magnitude * v2.magnitude)
end
```

Cosine similarity is a useful measure of comparing two different vectors that need to be similar. Effectively what it is measuring is the angle between the two vectors and not the magnitude. So for instance in this case if we had [2,2] vs [1,1] we would see a cosine similarity of 1.

```
cosine_similarity([2,2], [1,1]) #=> 0.9999 ~ 1
```

Using this is very useful for comparing two descriptors such as our face to see whether they are at least in the same direction.

You remember that our Face before was a struct so we need to actually fill in those pieces.

```
# lib/face.rb

class Face
  include OpenCV
  MIN_HESSIAN = 600

  attr_reader :filepath
```

```
    def initialize(filepath)
      @filepath = filepath
    end

    def descriptors
      @descriptors ||= features.last
    end

    def keypoints
      @keypoints ||= features.first
    end

    private
    def features
      image = CvMat.load(@filepath, CV_LOAD_IMAGE_GRAYSCALE)
      param = CvSURFParams.new(MIN_HESSIAN)
      @keypoints, @descriptors = image.extract_surf(param)
    end
  end
```

As you can see there's not a lot going on here except extracting out key points and descriptors from what OpenCV says with the SURF algorithm. MIN_HESSIAN is an arbitrary number that is recommended to be between 400 and 800. As the MIN_HESSIAN goes up the less amount of features SURF will detect but those features will be more important. We are also extracting 64 dimensions out of each face.

We have taken our raw images and converted them into smaller avatars. And now we have a way of extracting a feature set out of them, now what?

The Neighborhood Class

At this point we have enough information to build a Neighborhood class which will find us the closest features and *vote* on attributes associated with the pictures those features are contained in.

Visually what we want to do is this:

Create features to nearest neighbor features to nearest neighbor

As you can see for each feature of the image we want to match that to a big set of features. As soon as we find a closest *K* of those features, each one of those close features then has an image associated with it. And for all of those faces there are attributes associated with that which we'll use to calculate the eventual classification.

Based on our previous knowledge of finding things that are close we could approach this in many different ways. We could choose to use a Mahalanobis distance, a Euclidean distance or even a Manhattan distance. Since we don't really know enough about the data we will go with the simplest most common distance metric which is Euclidean. This is a good generic distance function to use and could probably be revisited at a different time.

K-D Tree

A K-D Tree or K Dimensional Tree is a data structure used to match vectors to a tree of vectors. This can be utilized to do a very quick nearest neighbor search using k-dimensions.

To test this we would first define our test to be:

```
# test/lib/neighborhood_spec.rb

describe Neighborhood do
  it 'finds the nearest id for a given face' do
    files = ['./test/fixtures/avatar.jpg']
    n = Neighborhood.new(files)

    n.nearest_feature_ids(files.first, 1).each do |id|
      n.file_from_id(id).must_equal files.first
    end
  end
end
```

Filling in the blanks we have a neighborhood that now looks like

```
# lib/neighborhood.rb

class Neighborhood
  def initialize(files)
    @ids = {}
    @files = files
    setup!
  end

  def file_from_id(id)
    @ids.fetch(id)
  end

  def nearest_feature_ids(file, k)
    desc = Face.new(file).descriptors

    ids = []

    desc.each do |d|
```

```
        ids.concat(@kd_tree.find_nearest(d, k).map(&:last))
      end

      ids.uniq
    end
  end
```

You'll notice that the ids are unique. That is because we need a unique set of features to match against and don't care about replacement in our feature set. Now that we have a neighborhood that works with a simple file we need to annotate and actually use some real data for that we'll need a face database.

Bootstrapping the neighborhood with faces

To be able to achieve what we want with our facial hair and glasses detection program we need a set of faces with facial hair, glasses and without. For that we have access to the AT&T facial database [*http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html*]. These data contains 40 different people photographed 10 times each. There was no annotation on any of the pictures so instead we can use manually build JSON files called *attributes.json*. These sit at the root of each individuals folder of photos and takes the form of:

```
{
  "facial_hair": false,
  "glasses": false,
}
```

If in one case there is a picture of a person with glasses vs the same person without glasses we use the following type of *attributes.json*.

```
[{
  "ids": [1,2,5,6,7,8,9,10],
  "facial_hair": true,
  "glasses": true,
},
{
  "ids": [3,4],
  "facial_hair": true,
  "glasses": false,
}]
```

Note that we have an array with ids which split up the attributes. We need to write a test for attaching attributes to images.

```
# test/lib/neighborhood_spec.rb

describe Neighborhood do
  it 'returns attributes from given files' do
    files = ['./test/fixtures/avatar.jpg']

    n = Neighborhood.new(files)
```

```
    expected = {
      'fixtures' => JSON.parse(File.read('./test/fixtures/attributes.json'))
    }

    n.attributes.must_equal expected
  end
end
```

What this does is for each folder the attributes should be parsed and brought into a hash.

```
# lib/neighborhood.rb

class Neighborhood
  # initialize
  # file_from_id
  # nearest_feature_ids

  def attributes
    attributes = {}
    @files.each do |file|
      att_name = File.join(File.dirname(file), 'attributes.json')

      attributes[att_name.split("/")[-2]] = JSON.parse(File.read(att_name))
    end
    attributes
  end
end
```

But we're still missing one piece which is a hash containing the votes of the different classes ("Facial Hair No Glasses", "Facial Hair Glasses", "Glasses No Facial Hair", "Glasses Facial Hair"). In our case we'd like something like this

```
{
  'glasses' => {false => 1, true => 0},
  'facial_hair' => {false => 1, true => 1}
}
```

That way we can see the votes for each class broken out. Writing a test for this it would look like:

```
# test/lib/neighborhood.rb

describe Neighborhood do
  it 'finds the nearest face which is itself' do
    files = ['./test/fixtures/avatar.jpg']
    neighborhood = Neighborhood.new(files)

    descriptor_count = Face.new(files.first).descriptors.length
    attributes = JSON.parse(File.read('./test/fixtures/attributes.json'))

    expectation = {
      'glasses' => {
```

```
        attributes.fetch('glasses') => descriptor_count,
        !attributes.fetch('glasses') => 0
      },
      'facial_hair' => {
        attributes.fetch('facial_hair') => descriptor_count,
        !attributes.fetch('facial_hair') => 0
      }
    }

    neighborhood.attributes_guess(files.first).must_equal expectation
  end

  it 'returns the proper face class' do
    file = './public/att_faces/s1/1.png'
    attrs = JSON.parse(File.read('./public/att_faces/s1/attributes.json'))

    expectation = {'glasses' => false, 'facial_hair' => false}

    attributes = %w[glasses facial_hair]
    Neighborhood.face_class(file, attributes).must_equal expectation
  end
end
```

And filling in the pieces we'd have this in our Neighborhood class.

```
# lib/neighborhood.rb

class Neighborhood
  # initialize
  # file_from_id
  # nearest_feature_ids
  # attributes

  def self.face_class(filename, subkeys)
    dir = File.dirname(filename)
    base = File.basename(filename, '.png')

    attributes_path = File.expand_path('../attributes.json', filepath)
    json = JSON.parse(File.read(attributes_path)))

    h = nil

    if json.is_a?(Array)
      h = json.find do |hh|
        hh.fetch('ids').include?(base.to_i)
      end or
      raise "Cannot find #{base.to_i} inside of #{json} for file #{filename}"
    else
      h = json
    end

    h.select {|k,v| subkeys.include?(k) }
  end
```

```
  def attributes_guess(file, k = 4)
    ids = nearest_feature_ids(file, k)

    votes = {
      'glasses' => {false => 0, true => 0},
      'facial_hair' => {false => 0, true => 0}
    }

    ids.each do |id|
      resp = self.class.face_class(@ids[id], %w[glasses facial_hair])

      resp.each do |k,v|
        votes[k][v] += 1
      end
    end

    votes
  end
end
```

Now we have the issue of making this thing useful. You'll notice that the default value for attributes_guess is K. We need to find that still.

Cross Validation and Finding K

Now we need to actually train and build an optimal model and find our K. For that we're going to first fold our AT&T database into two pieces like so:

```
# test/lib/neighborhood_spec.rb

describe Neighborhood do
  let(:files) { Dir['./public/att_faces/**/*.png'] }

  let(:file_folds) do
    {
      'fold1' => files.each_with_index.select {|f, i| i.even? }.map(&:first),
      'fold2' => files.each_with_index.select {|f, i| i.odd? }.map(&:first)
    }
  end

  let(:neighborhoods) do
    {
      'fold1' => Neighborhood.new(file_folds.fetch('fold1')),
      'fold2' => Neighborhood.new(file_folds.fetch('fold2'))
    }
  end
end
```

Next we are going to want to build a test for each fold and cross validate to see what the errors look like. This isn't a unit test at this point because we are doing some experimentation instead.

```
# test/lib/neighborhood_spec.rb

describe Neighborhood do
  %w[fold1 fold2].each_with_index do |fold, i|
    other_fold = "fold#{(i + 1) % 2 + 1}"
    it "cross validates #{fold} against #{other_fold}" do
      (1..7).each do |k_exp|
        k = 2 ** k_exp - 1
        errors = 0
        successes = 0

        dist = measure_x_times(2) do
          file_folds.fetch(fold).each do |vf|
            face_class = Neighborhood.face_class(vf, %w[glasses facial_hair])
            actual = neighborhoods.fetch(other_fold).attributes_guess(vf, k)

            face_class.each do |k,v|
              if actual[k][v] > actual[k][!v]
                successes += 1
              else
                errors += 1
              end
            end
          end
        end

        error_rate = errors / (errors + successes).to_f

        avg_time = dist.reduce(Rational(0,1)) do |sum, bm|
          sum += bm.real * Rational(1,2)
        end
        print "#{k}, #{error_rate}, #{avg_time}\n"
      end
    end
  end
end
```

This prints out some useful information in finding our optimal K.

Error Rate and Time to run over different K's. image::images/error_rate_speed_knn.png

Table 3-4. Cross Validation Fold #1

K	Error Rate	Time to Run
1	0.0	1.4284405
2	0.0	0.8799995
4	0.0575	1.3032545
8	0.1775	2.121337
16	0.2525	3.7583905
32	0.255	8.555531

K	Error Rate	Time to Run
64	0.255	23.3080745

Table 3-5. Cross Validation Fold #2

K	Error Rate	Time to Run
1	0.0	1.4773145
2	0.0	0.9168755
4	0.05	1.3097035
8	0.21	2.1183575
16	0.2475	3.890095
32	0.2475	8.6245775
64	0.2475	23.480187

We now can make a better judgement as to which K to use. Instead of picking the obvious K = 1 or K = 2 instead we should use K = 4. The reason is because as new data comes in we want to give the ability to check all 4 classes and therefore make the model stronger to change.

At this time our code works!

Conclusion

K-Nearest Neighbors is one of the best algorithms for classifying data sets. It is lazy, and non-parametric. It also works fairly fast if using something like a KD Tree. As you saw in this chapter it is well suited for any type of problem where you want to model voting or things that can be determined to be close together or not.

You also learned about the issues related with KNN, like the curse of dimensionality. When we built our tool to determine whether someone was wearing glasses or had facial hair we learned quickly that if we looked at all pixels things would break down, and instead had to apply SURF to reduce the dimensions.

But when approaching many problems KNN can be a good suitable tool for determining whether you can solve anything at all since it follows the Bayes error rate.

CHAPTER 4

Naive Bayesian Classification

Remember e-mail before Gmail? You probably remember your inbox full of spam messages ranging from Nigerian princes wanting to pawn off money to pharmaceutical advertisements. It became such a major issue that we spent most of our time filtering spam.

The amount of time spent filtering spam is much less than it used to be due to GMail, and tools like SpamAssasin. Using a classifier called a Naive Bayesian Classifier tools these have been able to mitigate the influx of spam to our inboxes.

In this chapter we will discuss

1. The Bayes Theorem
2. What a Naive Bayesian Classifier is and why it's called Naive
3. Building a Spam Filter using a naive bayesian classifier

Naive Bayesian Classification in Brief

A Naive Bayes Classifier is a supervised and probabalistic learning method. It does well with data that you can assert there is an independence with. Also prefers problems where the probability of any attribute is greater than zero.

Bayes Theorem used to find fraudulent orders

Imagine you're running an online store and lately you've been fraught with fraudulent orders. At your estimation you are seeing about 10% of all orders coming in be fraudulent. Basically 10% of orders are people stealing from you. Now of course you want to mitigate this by reducing the fraudulent orders but you are up against a conundrum.

Every month you receive at least 1000 orders, and if you were to check every single one, you'd spend more money fighting fraud than the fraud was costing you in the first place. Assuming that it takes up to 60 seconds per order to determine whether it's fraud or not, and a customer service representative costs around 15 dollars an hour to hire that is 200 hours would be 3,000 in time per year.

One way of doing this would be to construct a probability that an order is over 50% fraudulent. In this case we'd expect the amount of orders we'd have to look at to be much less. But this is where things become difficult because the only thing we can determine is the probability it's fraudulent which is 10%. Given that piece of information we'd be back at square one looking at all orders because it's more probable that an order is not fraudulent!

Let's say that we notice something which is that fraudulent orders use gift cards and multiple promo codes often. Using this knowledge how would we determine what is fraud or not namely how would we calculate the probability of fraud given that they used a gift card.

For that we have to first talk about conditional probabilities

Conditional Probabilities

Generally speaking people understand the meaning of the probability of something happening. For instance the probability of an order being fraudulent is 10%. But what about the probability of an order being fraudulent given that it used a gift card. For that we need something called a conditional probability.

Equation 4-1. The mathematical definition is

$$P(A|B) = \frac{P(A\capB)}{P(B)}$$

Probability symbols

Generally speaking writing $P(E)$ means that you are looking at the probability of a given event. This event can be a lot of different things including the event that A and B happened, the probability that A or B happened or the probability of A given B happening in the past. The way you would notate each of these is as follows.

$A \cap B$ could be called the *and* function since it is the intersection of A and B. For instance in ruby it would be:

```
a = [1,2,3]
b = [1,4,5]
```

```
a & b #=> [1]
```

$A \cup B$ could be called the *or* function since it is both A and B. For instance in Ruby it would be:

```
a = [1,2,3]
b = [1,4,5]

a | b #=> [1,2,3,4,5]
```

The probability of A given B would be as follows in Ruby:

```
a = [1,2,3]
b = [1,4,5]

total = 6.0

p_a_cap_b = (a & b).length / total
p_b = b.length / total

p_a_given_b = p_a_cap_b / p_b #=> 0.33
```

This is basically that the probability of A happening given that B happened is the probability of A *and* B happening divided by the probability of B. Graphically it looks something like this:

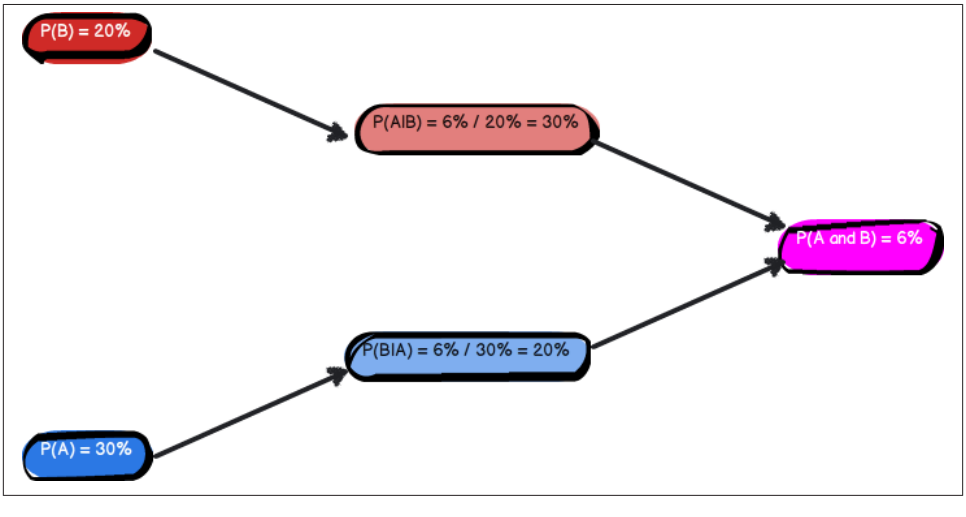


Figure 4-1. This shows how P(A|B) sits between P(A and B) and P(B)

In our fraud example let's say we want to measure the probability of fraud given it has a gift card. This would be $P(Fraud \mid Giftcard) = \frac{P(Fraud \cap Giftcard)}{P(Giftcard)}$. Now this works if you know the actual probability of Fraud and Giftcard happening.

At this point we are up against a problem which is we cannot calculate P(Fraud, Giftcard) because that is something that is hard to separate out. To solve this problem we need a trick introduced by Bayes.

Inverse Conditional Probability aka The Bayes Theorem

Reverend Thomas Bayes in the 1700's came up with the original research which would become the Bayes Theorem. Laplace extended this to become what we know today as a beautiful result. It is:

Equation 4-2. The Bayes Theorem

$$P(B \mid A) = \frac{P(A \mid B)P(B)}{P(A)}$$

This is because of the following:

\[P(B|A) = \frac{\frac{P(A \cap B)P(B)}{P(B)}}{P(A)} = \frac{P(A\capB)}{P(A)}\]

This is a beautiful result which is useful in our fraud example because we can effectively backout our result using other information. Using the Bayes theorem we would now calculate:

$$P(Fraud \mid Giftcard) = \frac{P(Giftcard \mid Fraud)P(Fraud)}{P(Giftcard)}$$

Remember that the probability of fraud was 10%. Let's say that the probability of giftcard use is 10% and based on our research the probability of giftcard use in a fraudulent order is 60%. So what is the probability that an order is fraudulent given that it uses a giftcard?

\[P(Fraud | Giftcard) = \frac{60% \dot 10%}{10%} = 60%\]

The beauty of this is that your work on measuring fraudulent orders goes down drastically because all you have to look for is the orders with giftcards. Since the total number of orders is 1000, and 100 of those are fraudulent we will look at 60 of those fraudulent orders. Out of the remaining 900 there are 90 of those which have giftcards in them which brings the total to look at to 150!

At this point you'll notice we reduced orders needing fraud review from 1000 to 40 or 4% of total. But can we do better? What about introducing something like people using multiple promo codes or other information?

Naive Bayesian Classifier

We've already solved the problem of finding fraudulent orders given that it has a giftcard but what about the problem of fraudulent orders given the fact that they have giftcards or they have multiple promo codes or other *features*. How would we go about that?

Namely we want to solve the problem of $P(A \mid B, C) = ??$. For this we need a bit more information and something called the chain rule.

The Chain Rule

If you remember back to probability class the probability of A and B happening is the probability of B given A times the probability of A. Mathematically this looks like $P(A \capB) = P(B|A)P(A)$. This is assuming they are *not* mutually exclusive. Using something called a joint probability this smaller result transforms intothe chain rule.

Joint probabilities are the probability that all the events will happen. We denote this by using $\cap$. The generic case of the chain rule is:

```
\[P(A_1, A_2,\dots,A_n)=P(A_1)P(A_2|A_1)P(A_3|A_1,A_2)\dotsP(A_n|
A_1,A_2,\dots,A_{n-1})\]
```

This expanded out version is useful in trying to solve our problem of giving lots of information into our bayesian probability estimates. But there is one problem which is that this can quickly evolve into a complex calculation using information we don't have, so we make one big assumption and act naive.

Naivety in Bayesian Reasoning

The chain rule above is useful for solving potentially inclusive problems but we don't have the ability to calculate all of those probabilities. For instance if we were to introduce multiple promos into our fraud example then we'd have the following to calculate.

$$P(Fraud \mid Giftcard, Promos) = \frac{P(Giftcard, Promos \mid Fraud)P(Fraud)}{P(Giftcard, Promos)}$$

Let's ignore the denominator for now since it doesn't depend on whether the order is fraudulent or not. AT this point we need to find the calculation for $P(Giftcard, Promos|Fraud)P(Fraud) If you remember from the chain rule above this is equivalent to $P(Fraud, Giftcard, Promos)$.

You can see this by the following:

$$P(Fraud, Gift, Promo) = P(Fraud)P(Gift, Promo \mid Fraud) = P(Fraud)P(Gift \mid Fraud)P(Promo \mid Frau$$

Now at this point we have a conundrum which is how do you measure the probability of promo given fraud and giftcards? While this is the correct probability it really can be difficult to measure especially with more features coming in. Instead what if we were to be a tad naive and just assume that we can get away with independence and just say that we don't care about the interaction between promo codes and giftcards just the interaction with those independently with fraud.

In that case our math would be much simpler and

$$P(Fraud, Gift, Promo) = P(Fraud)P(Gift \mid Fraud)P(Promo \mid Fraud)$$

This would be proportional to our numerator and to simplify things even more we can just assert that we'll normalize later with some magical Z which is just the sum of all the probabilities of classes. So now our model becomes:

$$P(Fraud \mid Gift, Promo) = \frac{1}{Z}P(Fraud)P(Gift \mid Fraud)P(Promo \mid Fraud)$$

To turn this into a classification problem we just determine which input whether Fraud or Not fraud yields the highest probability.

Table 4-1. Probability of Giftcards vs Promos

	Fraud	Not Fraud
Gift Card Present	60%	10%
Multiple Promos used	50%	30%
Probability of class	10%	90%

At this point you can use this information to determine whether an order is fraudulent purely based on whether it has a giftcard present and whether it used multiple promos. Calculating the probability that an order is fraudulent given the use of giftcards and multiple promos the probability would be 62.5%. While we can't exactly figure out how much savings this gives to you in terms of looking we know that we're using better information and making a better judgement.

There is one problem though which is what happens when the probability of using multiple promos given it's fraud is zero. This can happen for reasons including that there just isn't enough of a sample size. In that case it'd zero out the probability. The way we solve that is using something called a psuedocount.

Psuedocount

There is one big issue with a naive bayesian classifier and that is a problem of new information. For instance let's say we have a bunch of e-mail that classified as Spam or Ham. We build our probabilities using all of these data but then something bad happens a new spammy word called *fuzzbolt*. Nowhere in our data did we see the word *fuzzbolt* and so when we go to calculate the probability of Spam given the word *fuzzbolt* we get a probability of zero. This can have a zeroing out effect that will greatly skew results towards the data we have.

Since a naive bayesian classifier relies on multiplying all of the independent probabilities together to come up with a classification if any of those probabilities are zero then our probability will be zero.

Take for instance looking at the following email subject. "Fuzzbolt: Prince of Nigeria". Assuming we strip off "of" we have the following data collected.

Table 4-2. Probability of word given spam or ham

Word	Spam	Ham
Fuzzbolt	0	0
Prince	75%	15%
Nigeria	85%	10%

Now let's assume we want to calculate a score for ham or spam. In both cases the score would end up being zero cause *Fuzzbolt* doesn't happen. At that point since we have a tie we'd just go with the more common situation which is *ham*. This means that we have failed and classified something incorrectly due to one word not being recognized.

There is an easy fix for that and it's called *pseudocount*. When we go about calculating the probability we add a one to where the count of the word is. So everything will end up being word_count + 1. This helps mitigate the zeroing out effect for now. In the case of our fraud detector we would add 1 to each count to ensure that it is never zero.

So in our above example let's say we have 3000 words. We would give *fuzzbolt* a score of $\frac{1}{3000}$. The other scores would change slightly but this avoids the zeroing out problem.

Code Example: Spam Filter

The canonical machine learning example is building a spam filter. In this section we will work up a simple spam filter using a Naive Bayesian Classifier and improve it by utilizing a 3-gram tokenization model.

As we have learned before naive bayesian classifiers can be easily calculated, and operate well under strongly independent conditions.

Approaching how to solve this we need:

1. What the classes look like interacting with each other
2. A good data source
3. A tokenization model
4. An objective to minimize our error
5. A way to improve over time

Setup Notes

All of the code we're using for this example can be found on GitHub at *https://github.com/thoughtfulml/examples/tree/master/3-naive-bayesian-classification*.

Ruby is constantly changing so the README is the best place to come up to speed on running the examples.

You will have to make sure libxml is installed.

Class Diagram

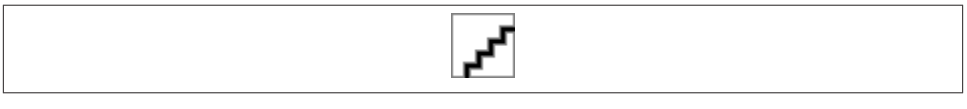


Figure 4-2. Class Diagram showing how e-mails get turned into a SpamTrainer

Each e-mail has an object that takes an *.eml* type text file that then tokenizes it into something the SpamTrainer can utilize for incoming e-mail messages.

Data Source

There are numerous sources of data that we can use but the best is raw email messages marked as either spam or ham. For our purposes we can use the CSDMC2010 SPAM corpus which is available on Sourceforge and has more information on *http://csmining.org/index.php/spam-email-datasets-.html*.

This dataset has 4327 total messages of which 2949 are Ham and 1378 are Spam. For our proof of concept this should work well enough for us.

Email Class

The Email class has one responsibility which is to parse an incoming email message according to the RFC for e-mails. To handle this we use the Mail gem since there's a lot of nuance in there. In our model all we're concerned with is subject and body.

The cases we need to handle are: HTML messages, Plaintext, and Multipart. Everything else we'll just ignore.

Building this class using Test Driven Development let's do through this one by one:

Starting with the simple plaintext case let's just copy one of the example training files from our data set under data/TRAINING/TRAIN_00001.eml to ./test/fixtures/plain.eml. This is a plaintext e-mail and will work for our purposes. IN testing e-mails it is good to know that the split between a message and header in an e-mail is usually denoted by "\r\n\r\n". Along with that header information is generally something like "Subject: A Subject goes here". Using that we can easily extract out our test case which is.

```
require 'spec_helper'

# test/lib/email_spec.rb

describe Email do
  describe 'plaintext' do
    let(:plain_file) { './test/fixtures/plain.eml' }
    let(:plaintext) { File.read(plain_file) }
    let(:plain_email) { Email.new(plain_file) }

    it 'parses and stores the plain body' do
      body = plaintext.split("\n\n")[1..-1].join("\n\n")
      plain_email.body.must_equal body
    end

    it 'parses the subject' do
      subject = plaintext.match(/^Subject: (.*)$/)[1]
      plain_email.subject.must_equal subject
    end
  end
end
```

Now instead of relying purely on regular expressions we want to utilize a gem to do this for us. For this we'll use the mail gem which will handle all of the nitty gritty details. Making e-mail work for this particular case we have

```
require 'forwardable'

# lib/email.rb

class Email
  extend Forwardable

  def_delegators :@mail, :subject

  def initialize(filepath)
    @filepath = filepath
    @mail = Mail.read(filepath)
```

```
  end

  def body
    @mail.body.decoded
  end
end
```

You'll notice that we're using def_delegators to delegate subject to the @mail object. This is just for simplicity sake. Now that we have captured the case of plaintext we need to solve the case of html. For that we want to only capture the inner_text and knowing that regular expressions are useless at this we need yet another gem: Nokogiri. Nokogiri will easily be able to do this for us. But first we need a test case which would look something like this:

```
# test/lib/email_spec.rb

describe Email do
  describe 'html' do
    let(:html_file) { './test/fixtures/html.eml' }
    let(:html) { File.read(html_file) }
    let(:html_email) { Email.new(html_file) }

    it "parses and stores the html body's inner_text" do
      body = html.split("\n\n")[1..-1].join("\n\n")
      html_email.body.must_equal Nokogiri::HTML.parse(body).inner_text
    end

    it "stores subject like plaintext does as well" do
      subject = html.match(/^Subject: (.*)$/)[1]
      html_email.subject.must_equal subject
    end
  end
end
```

At this point you'll notice that we're using Nokogiri to calculate the inner_text and will have to use this inside of the Email class as well. Now the problem is that we need to detect the content_type as well. So we'll add that in:

```
# lib/email.rb

require 'forwardable'

class Email
  extend Forwardable

  def_delegators :@mail, :subject, :content_type

  def initialize(filepath)
    @filepath = filepath
    @mail = Mail.read(filepath)
  end
```

```
    def body
      single_body(@mail.body.decoded, content_type)
    end

    private
    def single_body(body, content_type)
      case content_type
      when 'text/html'
        Nokogiri::HTML.parse(body).inner_text
      when 'text/plain'
        body.to_s
      else
        ''
      end
    end
  end
```

At this point we could add multipart processing as well but I will leave that as an exercise that you can try out yourself. In the coding repository that is called to earlier in the chapter you can see a multipart version.

Now we have a working email parser, but there's the next problem about tokenization or what to extract from the body and subject.

Tokenization and Context

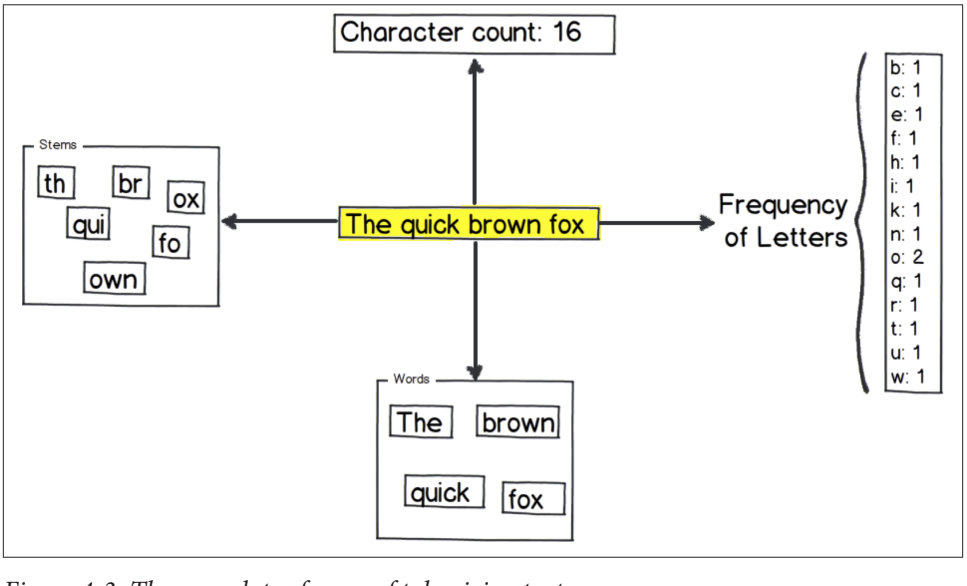


Figure 4-3. There are lots of ways of tokenizing text

There are numerous ways to tokenizing text: stems, word frequencies, words, and others. In the case of spam we are up against a tough problem because things are more contextual. The phrase *Buy now* sounds spammy where as *Buy*, *now* are not as spammy. Since we are building a naive bayesian classifier we are assuming that each individual token is contributing to the spamminess of the e-mail.

Building a tokenizer:

The goal of this tokenizer is to build something that will extract out words into a stream. Instead of returning an array we want to yield the token as it happens so that we are keeping a low memory profile. This should also downcase all strings to keep things similar.

```
# test/lib/tokenizer_spec.rb
require 'spec_helper'

describe Tokenizer do
  describe '1-gram tokenization' do
    it 'downcases all words' do
      expectation = %w[this is all caps]
      Tokenizer.tokenize("THIS IS ALL CAPS") do |token|
        token.must_equal expectation.shift
      end
    end

    it 'uses the block if given' do
      expectation = %w[feep foop]
      Tokenizer.tokenize("feep foop") do |token|
        token.must_equal expectation.shift
      end
    end
  end
end
```

You can see that this does two things including lowercasing all words. Also you'll notice that instead of returning an array we use a block. This is to mitigate memory constraints since there is no need to make an array and return it. This makes it more lazy. To make these following tests work though we will have to fill in the skeleton for our Tokenizer module like so:

```
# lib/tokenizer.rb

module Tokenizer
  extend self

  def tokenize(string, &block)
    current_word = ''
    return unless string.respond_to?(:scan)
    string.scan(/[a-zA-Z0-9_\u0000]+/).each do |token|
      yield token.downcase
```

```
    end
  end
end
```

Now that we have a way of parsing and tokenizing e-mails we can move onto the bayesian portion the SpamTrainer.

The Spam Trainer

Now that we have e-mail parsed correctly and words tokenized we now need to build the SpamTrainer.

The purpose of this class is to:

1. Store training data
2. Build a Bayesian Classifier
3. Minimize False Positive rate by testing.

Storing training data

The first step we need to tackle is the one about storing training data from a given set of e-mail messages. In a production environment you would pick something that has persistence. In our case we will go with storing everything in one big hash.

Remember that most machine learning algorithms have two steps: training, and then computation.

At this point we still need to:

1. Storing a set of all categories
2. Storing unique word counts for each category
3. Storing the totals for each category

The first step is to capture all of the category names for which a test would look something like this:

```
# test/lib/spam_trainer_spec.rb

describe SpamTrainer do
  describe 'initialization' do
    let(:hash_test) do
      {'spam' => './filepath', 'ham' => './another', 'scram' => './another2'}
    end

    it 'allows you to pass in multiple categories' do
      st = SpamTrainer.new(hash_test)
      st.categories.sort.must_equal hash_test.keys.uniq.sort
    end
```

```
  end
end
```

The solution is in the following code:

```
# lib/spam_trainer.rb

class SpamTrainer
  def initialize(training_files, n = 1)
    @categories = Set.new

    training_files.each do |tf|
      @categories << tf.first
    end
  end
end
```

You'll notice we're just using a set to capture this for now since it'll hold onto the unique version of what we need. Our next step is to capture the unique tokens for each email. We are using the special category called *_all* to capture the count for everything.

test/fixtures/small.eml.

```
Subject: One of a kind Money maker! Try it for free!

spam
```

```
# test/lib/spam_trainer_spec.rb

describe SpamTrainer do
  let(:training) do
    [['spam','./test/fixtures/plain.eml'], ['ham','./test/fixtures/small.eml']]
  end

  let(:trainer) { SpamTrainer.new(training)}

  it 'initializes counts all at 0 plus an _all category' do
    st = SpamTrainer.new(hash_test)
    %w[_all spam ham scram].each do |cat|
      st.total_for(cat).must_equal 0
    end
  end
end
```

To get this to work we have introduced a new step called *train!* which will take the training data iterate over it and save it into an internal hash. The following is a solution:

```
# lib/spam_trainer.rb

class SpamTrainer
  def initialize(training_files)
    setup!(training_files)
  end
```

```
    def setup!(training_files)
      @categories = Set.new

      training_files.each do |tf|
        @categories << tf.first
      end

      @totals = Hash[@categories.map {|c| [c, 0]}]
      @totals.default = 0
      @totals['_all'] = 0

      @training = Hash[@categories.map {|c| [c, Hash.new(0)]}]
    end

    def total_for(category)
      @totals.fetch(category)
    end

    def train!
      @to_train.each do |category, file|
        write(category, file)
      end
      @to_train = []
    end

    def write(category, file)
      email = Email.new(file)

      logger.debug("#{category} #{file}")

      @categories << category
      @training[category] ||= Hash.new(0)

      Tokenizer.unique_tokenizer(email.blob) do |token|
        @training[category][token] += 1
        @totals['_all'] += 1
        @totals[category] += 1
      end
    end
  end
```

At this point we have taken care of the training aspect of our program but really have no clue how well it performs. And it doesn't classify anything. For that we still need to build our classifier.

Building the Bayesian Classifier

Equation 4-3. Remember the Bayes theorem is:

$$P(A_i \mid B) = \frac{P(B \mid A_i)P(A_i)}{\sum_j P(B \mid A_j)P(A_j)}$$

But since we're being naive about this we've distilled that into something much simpler which is:

\[Score(Spam,W_1,W_2,\dots,W_n) = P(Spam)P(W_1|Spam)P(W_2|Spam) \dotsP(W_n|Spam)\]

Which is then divided by some normalizing constant Z.

Our goal now is to build the methods: *score*, *normalized_score*, and *classify*. Score will just be the raw score from the above calculation, normalized_score will fit the range from 0 to 1. This happens by dividing by the total sum Z.

The *score* method's test:

```
# test/lib/spam_trainer_spec.rb

describe SpamTrainer do
  describe 'scoring and classification' do
    let (:training) do
      [
        ['spam','./test/fixtures/plain.eml'],
        ['ham','./test/fixtures/plain.eml'],
        ['scram','./test/fixtures/plain.eml']
      ]
    end

    let(:trainer) do
      SpamTrainer.new(training)
    end

    let(:email) { Email.new('./test/fixtures/plain.eml') }

    it 'calculates the probability to be 1/n' do
      scores = trainer.score(email).values

      assert_in_delta scores.first, scores.last

      scores.each_slice(2) do |slice|
        assert_in_delta slice.first, slice.last
      end
    end
  end
end
```

Since the training data is uniform across the categories there is no reason for the score to differe across them. To make this work in our SpamTrainer object we will have to fill in the pieces like so:

```
# lib/spam_trainer.rb

class SpamTrainer
  #def initialize
  #def total_for
  #def train!
  #def write

  def score(email)
    train!

    unless email.respond_to?(:blob)
      raise 'Must implement #blob on given object'
    end

    cat_totals = totals

    aggregates = Hash[categories.map do |cat|
      [
        cat,
        Rational(cat_totals.fetch(cat).to_i, cat_totals.fetch("_all").to_i)
      ]
    end]

    Tokenizer.unique_tokenizer(email.blob) do |token|
      categories.each do |cat|
        r = Rational(get(cat, token) + 1, cat_totals.fetch(cat).to_i + 1)
        aggregates[cat] *= r
      end
    end

    aggregates
  end
end
```

Wow! There's a lot going on here. Let's take it step by step.

1. Train the model if not already trained (the *train!* method handles this)
2. For each token of the blob of an e-mail (this is just subject and body concatenated together)
3. . Iterate through all categories and calculate the naive bayesian probability of each without dividing by Z

Now that we have the score method figured out we need to build a normalized_score as well that adds up to 1. Testing for this we have:

```
# test/lib/spam_trainer_spec.rb

describe SpamTrainer do
  it 'calculates the probability to be exactly the same and add up to 1' do
    trainer.normalized_score(email).values.inject(&:+).must_equal 1
    trainer.normalized_score(email).values.first.must_equal Rational(1,3)
  end
end
```

and subsequently on the *SpamTrainer* class we have:

```
# lib/spam_trainer.rb

class SpamTrainer
  #def initialize
  #def total_for
  #def train!
  #def write
  #def score

  def normalized_score(email)
    score = score(email)
    sum = score.values.inject(&:+)

    Hash[score.map do |cat, aggregate|
      [cat, (aggregate / sum).to_f]
    end]
  end
end
```

After Scoring we need a Classification

Since we now have a score we need to calculate a classification for the end user to use. This classification should take the form of an object that returns *guess* and *score*. There is an issue of tie breaking here.

Let's say for instance we have a model that has *turkey* and *tofu*. What happens when the scores come back as 1/2 and 1/2? Probably the best course of action is to go with what is more popular whether it be *turkey* or *tofu*. What about the case where the probability are the same? I think in that case we can just go with alphabetical order.

Testing for this we need to introduce a preference order which is the occurance of each category. A test for this would be:

```
# test/lib/spam_trainer_spec.rb

describe SpamTrainer do
  describe 'scoring and classification' do
    it 'sets the preference based on how many times a category shows up' do
      expected = trainer.categories.sort_by {|cat| trainer.total_for(cat) }

      trainer.preference.must_equal expected
```

```
    end
  end
end
```

Getting this to work is trivial actually and would look like this

```
# lib/spam_trainer.rb

class SpamTrainer
  #def initialize
  #def total_for
  #def train!
  #def write
  #def score
  #def normalized_score

  def preference
    categories.sort_by {|cat| total_for(cat) }
  end
end
```

Now that we have preference setup we can test for our classification being correct. Which would look like:

```
# test/lib/spam_trainer.rb

describe SpamTrainer do
  describe 'scoring and classification' do
    it 'gives preference to whatever has the most in it' do
      score = trainer.score(email)
      preference = trainer.preference.last
      preference_score = score.fetch(preference)

      expected = SpamTrainer::Classification.new(preference, preference_score)

      trainer.classify(email).must_equal expected
    end
  end
end
```

Getting this to work in code again is simple:

```
# lib/spam_trainer.rb

class SpamTrainer
  Classification = Struct.new(:guess, :score)
  #def initialize
  #def total_for
  #def train!
  #def write
  #def score
  #def preference
  #def normalized_score
```

```
  def classify(email)
    score = score(email)
    max_score = 0.0
    max_key = preference.last
    score.each do |k,v|
      if v > max_score
        max_key = k
        max_score = v
      elsif v == max_score && preference.index(k) > preference.index(max_key)
        max_key = k
        max_score = v
      else
        # Do nothing
      end
    end
    throw 'error' if max_key.nil?
    Classification.new(max_key, max_score)
  end
end
```

Error Minimization through Cross Validation

At this point we need to measure how well our model works. To do this we need to take our data that we have downloaded above and do a cross validation test on it. From there we need to measure only false positives and then based on that determine whether we need to fine tune our model more.

Our objective: Minimize False Positives

Up until this point our goal with making models has been to minimize *error*. This error could be easily denoted as the count of misclassifications divided by the total classifications. In most cases this is exactly what we want but in a spam filter this isn't exactly what we're optimizing for. Instead we want to minimize false fositives. False positives, also known as Type I errors, are when the model incorrectly predicts a positive when it should have been negative.

In our case if our model predicts Spam when in fact it isn't then the user will lose their e-mails. We want our spam filter to have the least amount of false positives as possible. On the other hand if our model incorrectly predicts something as Ham when it isn't we don't care as much.

Instead of minimizing the total misclassifications divided by total classifications we want to minimize spam misclassifications divided by total classifications. We will also measure false negatives as well but they are less important since we are trying to reduce spam that enters your mailbox not eliminate it.

To take care of this we first need to take some information from our data set which is in it's keyfile.label

Building the two folds

Inside of the spam email training data there is a file called `keyfile.label`. This contains information about whether the file is *Spam* or *Ham*. Inside of our cross validation test we can easily parse it using the following code.

```
# test/cross_validation_spec.rb

describe 'Cross Validation' do
  def self.parse_emails(keyfile)
    emails = []
    File.open(keyfile, 'rb').each_line do |line|
      label, file = line.split(/\s+/)
      emails << Email.new(filepath, label)
    end
    emails
  end

  def self.label_to_training_data(fold_file)
    training_data = []
    st = SpamTrainer.new([])

    File.open(fold_file, 'rb').each_line do |line|
      label, file = line.split(/\s+/)
      st.write(label, file)
    end

    st
  end

  def self.validate(trainer, set_of_emails)
    correct = 0
    false_positives = 0.0
    false_negatives = 0.0
    confidence = 0.0

    set_of_emails.each do |email|
      classification = trainer.classify(email)
      confidence += classification.score
      if classification.guess == 'spam' && email.category == 'ham'
        false_positives += 1
      elsif classification.guess == 'ham' && email.category == 'spam'
        false_negatives += 1
      else
        correct += 1
      end
    end

    total = false_positives + false_negatives + correct

    message = <<-EOL
    False Positives: #{false_positives / total}
```

```
      False Negatives: #{false_negatives / total}
      Accuracy: #{(false_positives + false_negatives) / total}
      EOL
      message
    end
  end
```

Cross Validation and Error Measuring

From here we can actually build our cross validation test which will read fold1 and fold2 and then cross validate to determine the actual error rate. The test looks something like this:

```
# test/cross_validation_spec.rb
describe 'Cross Validation' do
  describe "Fold1 unigram model" do
    let(:trainer) {
      self.class.label_to_training_data('./test/fixtures/fold1.label')
    }

    let(:emails) {
      self.class.parse_emails('./test/fixtures/fold2.label')
    }

    it "validates fold1 against fold2 with a unigram model" do
      skip(self.class.validate(trainer, emails))
    end
  end

  describe "Fold2 unigram model" do
    let(:trainer) {
      self.class.label_to_training_data('./test/fixtures/fold2.label')
    }

    let(:emails) {
      self.class.parse_emails('./test/fixtures/fold1.label')
    }

    it "validates fold2 against fold1 with a unigram model" do
      skip(self.class.validate(trainer, emails))
    end
  end
end
```

Running the command `ruby test/cross_validation_spec.rb` we get the following results:

Results from cross validation test.

```
WARNING: Could not parse (and so ignoring) 'From spamassassin-devel-admin@lists.sourceforge.net Fr
Parsing emails for ./test/fixtures/fold2.label
WARNING: Could not parse (and so ignoring) 'From quinlan@pathname.com Thu Oct 10 12:29:12 2002'
Done parsing emails for ./test/fixtures/fold2.label
```

```
Cross Validation::Fold1 unigram model
  validates fold1 against fold2 with a unigram model

        False Positive Rate (Bad): 0.0036985668053629217
        False Negative Rate (not so bad): 0.16458622283865001
        Error Rate: 0.16828478964401294

WARNING: Could not parse (and so ignoring) 'From quinlan@pathname.com Thu Oct 10 12:29:12 2002'
Parsing emails for ./test/fixtures/fold1.label
WARNING: Could not parse (and so ignoring) 'From spamassassin-devel-admin@lists.sourceforge.net Fr
Done parsing emails for ./test/fixtures/fold1.label
Cross Validation::Fold2 unigram model
  validates fold2 against fold1 with a unigram model

        False Positive Rate (Bad): 0.005545286506469501
        False Negative Rate (not so bad): 0.17375231053604437
        Error Rate: 0.17929759704251386
```

You'll notice that the false negative rate (classifying it as ham when it's actually spam) is much higher than the false positive rate (classifying as spam when it's ham). This is because of the Bayes Theorem! Let's look at the actual probabilities for Ham vs Spam:

Table 4-3. Spam vs Ham

Category	Email Count	Word Count	Probability of Email	Probability of Word
Spam	1,378	231,472	31.8%	36.3%
Ham	2,949	406,984	68.2	63.7%
Total	4,327	638,456	100%	100%

As you can see Ham is more probable so we will just default to that and more often than not we will classify something as Ham when it might not be. The good thing here though is that we have reduced spam by 80% without sacrificing incoming messages.

Conclusion

During this chapter we have delved into building and understanding a naïve bayesian classifier. As you have learned it is well suited for data that can be asserted to be independent. Being a probablistic model it works well with classifying data into multiple directions given the underlying score. This supervised learning method is well suited for fraud detection, spam filtering and anything that has these types of features.

CHAPTER 5

Hidden Markov Model

Intuition tells us that certain words tend to be of a certain part of speech tag, it also tells us that if a user visits a signup page then they have a higher probability of becoming a customer. But how would someone build a model around intuition?

Hidden Markov models, which we will discuss in this chapter, are versed in finding the underlying state of a given system using observations and an assumption about how those states work. In this chapter we will first talk about how to track user states given their actions, then develop some more knowledge about what a **hidden Markov model** is and the finally build a part of speech tagger using the Brown Corpus.

Hidden Markov Model in Brief

Hidden Markov Models can be either supervised or unsupervised and are called Markovian due to their reliance on a Markov Model. They work well where there doesn't need to be a lot of historical information built into the model. They also work well with adding localized context to a classification.

Tracking user behavior using state machines

Have you ever heard of the sales funnel? This is the idea that there are different levels of customer interaction. People will start as a prospect and then transition into more engaged states.

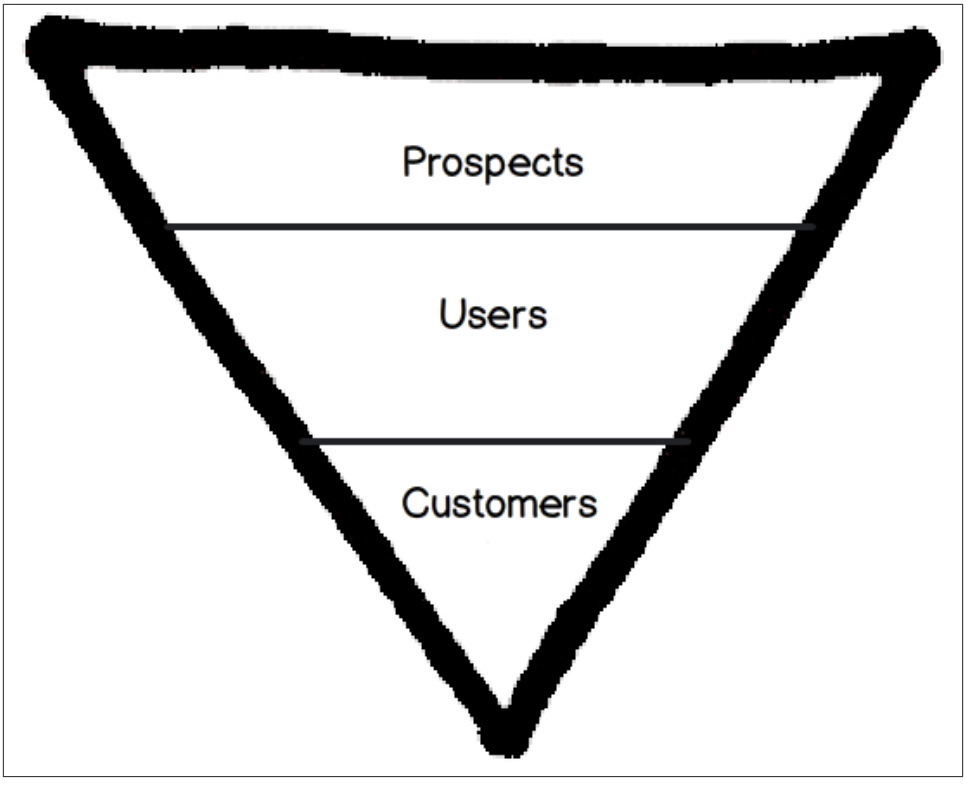


Figure 5-1. The generalized Sales Funnel, from Prospect to Customer

Let's say that we have an online store and determine that out of prospects that show up to the site 15% will signup for the site, and 5% will become customers right away. When the user is already a user they will cancel their membership 5% of the time and buy something 15% of the time. If they are a customer then they will cancel their account only 2% of the time and go back to being a regular user 95% of the time instead of continually buying things.

We could represent this information we have collected in something called a **transition matrix** which shows the probability of going from one state to another.

Table 5-1. Transition Probability

	Prospect	User	Customer
Prospect	0.80	0.15	0.05
User	0.05	0.80	0.15
Customer	0.02	0.95	0.03

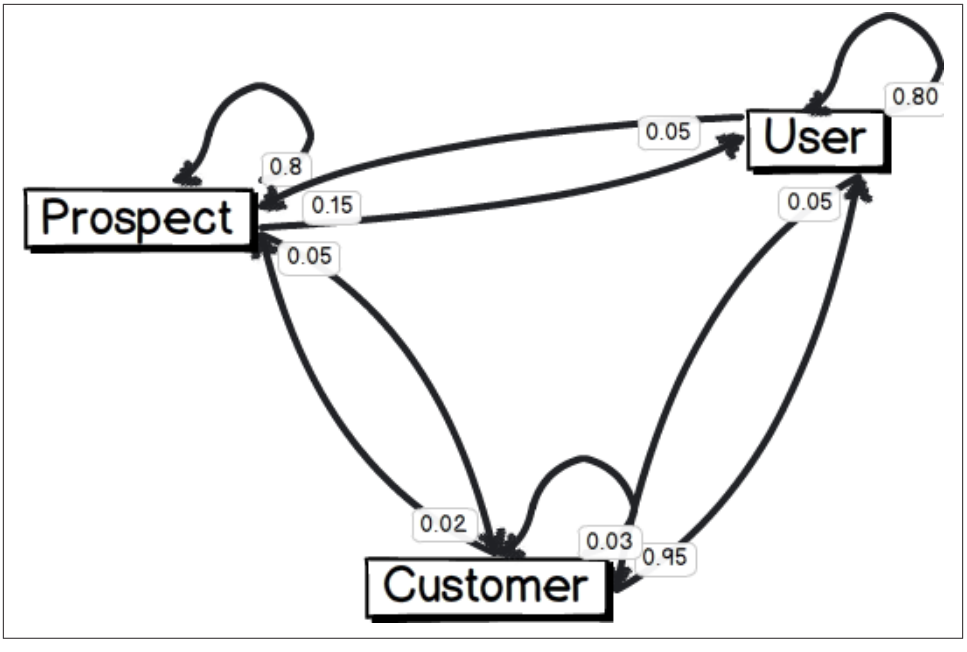


Figure 5-2. A Sales funnel state machine with probabilities

What this defines is a state machine. It also tells us a lot about how our current customers behave. We can determine what the conversion rate is, attrition rate, and other probabilities. Conversion rate would be the probability of signing up from being a prospect which would be 20%. This is simply the probability of going from prospect to user plus prospect to customer (15% + 5%). You could also determine attrition rate as being the average of 5% and 2% which is 3.5%.

This is an uncommon way of displaying user behavior in analytics and that is because it is too explanatory of user behavior. But there is one thing that it has over traditional conversion rate calculations and that is the ability to look at how a user operates over time. For instance we could determine the probability of being a *prospect* given the last 4 times they were in fact a *prospect*. This would be 80% multiplied five times which is 32%. The probability that someone keeps viewing the site and never sign up is low because eventually they might sign up.

But there is one major problem with this model. There is no way for us to reliably determine these states without asking each user individually. The state is hidden from our observation. A user can view the site anonymously.s

That is actually fine as you will soon see. If we are able to observe interaction with the site and make a judgement call about the underlying transitions from other sources (think google analytics) then we could solve this problem.

The way we do this is by introducing another level of complexity called **emissions**.

Emissions / Observations of underlying States

With our example above we don't know when someone goes from being a prospect to a user to a customer. But we are able to observe what a user is doing and what their behavior is. We know that for a given observation there is a probability that they are in a given state.

We can do this to determine what the underlying state of users is using emitted behaviors that we can easily observe. Let's say for instance that we have five pages on our website: *homepage*, *signup*, *product page*, *checkout*, and *contact us*. Now you could imagine that some of these pages matter while others do not. For instance signup would most likely mean the prospect becomes a user and checkout means the user becomes a customer.

This information gets more interesting because we know the probabilities of states. Let's say we know this:

Table 5-2. Emission and State probabilities

Page Name	Prospect	User	Customer
homepage	0.4	0.3	0.3
signup	0.1	0.8	0.1
product page	0.1	0.3	0.6
checkout	0	0.1	0.9
contact us	0.7	0.1	0.2

We know the probability of users switching states as well as the probability of the behavior they are emitting given the underlying state. Given this info, what is the probability a user who has viewed the homepage, signup, and product page a customer? Namely we want to solve this problem:

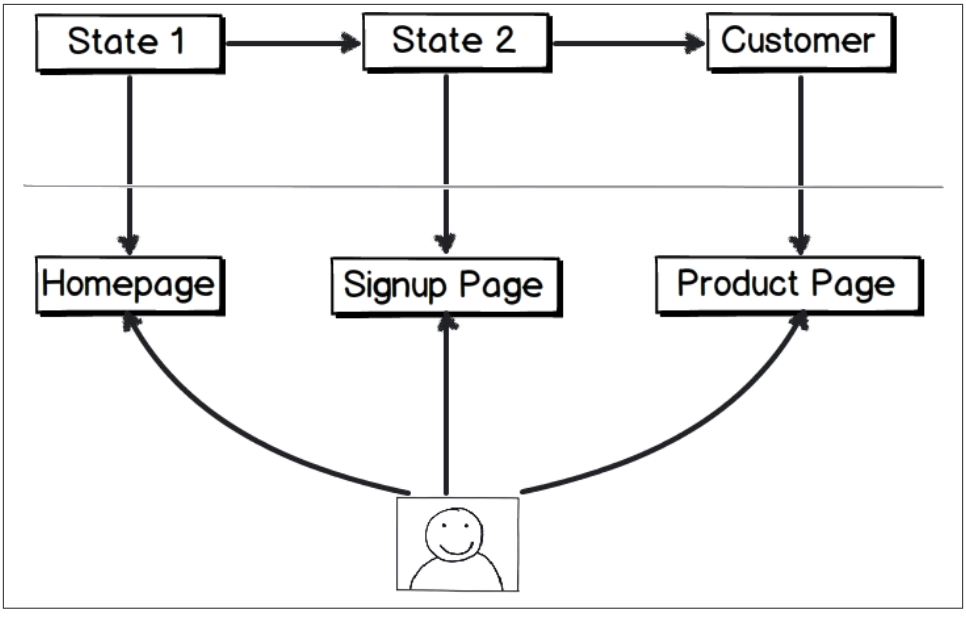


Figure 5-3. You can only observe what your users are doing but there is a hidden state

To figure this out we would need to figure out the probability that a user is in the customer state given all their previous states or $P(Customer \mid S_1, S_2)$, as well as the probability of the user viewing the product page given they were a customer times the probability of signup given the state etc or $P(Product\_Page \mid Customer) * P(Signup\_Page \mid S_2) * P(Homepage \mid S_1)$. The problem here is that there are more unkowns than knowns.

This finite model is difficult to solve because it involves a lot of calculations. Calculating something like $P(Customer \mid S_1, S_2, \cdots, S_N)$ is a complicated problem to solve. To solve this we need to introduce something called the Markov Assumption.

Emissions and Observations

Emissions and Observations are used interchangeably in the hidden markov model nomenclature. They are the same thing, the thing to remember is that it is what a process is emitting or what you can observe. That is all.

Simplification through The Markov Assumption

If you remember from Naive Bayesian classifiers each attribute would independently add to the probability of some events. So for Spam, the probability would be independ-

ently conditional on words like *Prince* and *Buy Now*. In our model that we're building with user behavior though we do want dependence. Mainly we want the previous state to be part of the next state's probability. Matter of fact we would assert that the previous states have a relationship to what the user's state is in now.

In the case of Naive Bayesian classifiers we would make the assumption that the probability of something was independently conditional on other events. So spam was indepenently conditional on each word in the email.

We can do the same with a system like this. We can state that the probability of being in a particular state is primarily based on what happened in the last state. So instead of $P(Customer \mid S_1, S_2, \cdots, S_n)$ it would be $P(Customer \mid S_N)$. But why can we get away with such a gross simplification?

Given a state machine like the one we have defined above, the system infers probalistically and recursively where you have been in the past. For instance if you were in state Customer then you could say that the most probable previous state would be User, and from there the most probable state before that would be Prospect.

This simplification also has one exciting conclusion which leads into something called a Markov Chain.

Using Markov Chains instead of a Finite State Machine

We have been talking purely about one system thus far. And only one outcome but what is powerful about the **Markov assumption** is that you can model a system as it operates forever. Instead of looking locally at what the process is going to do we can figure out how the system behaves always. This leads into the idea of what's called a **Markov chain**.

Markov Chains are exceptional at simulating systems. Queuing theory, finance, weather modeling, and game theory all make heavy use of Markov Chains. They are powerful because they represent behaviors in a concise way. We can also determine quickly how we would assume a system to perform given a Markov chain.

These Markov Chains can analyze and find information out of an underlying process that will operate forever. But that still doesn't solve the underlying problem we have which is that we still need to solve the problem of what state a given customer is in given it's hidden previous state as well as purely based on observations. For that we will need to enhance Markov chains to include a hidden aspect to them.

Hidden Markov Model

We've talked a lot about observation and underlying state transitions but have gone almost back to where we started. We still need to figure out what a user's state is. To do this we will use something called a **hidden Markov model** for which there is always three questions:

1. Evaluation: Given a sequence like *Homepage* → *Signup* → *Product* → *Checkout* how likely is that to come from our transition and observation of users?
2. Decoding: Given the sequence above what is the most likely underlying state sequence look like?
3. Learning: Given an observed sequence what is the next thing the user most likely will do?

In this section we will first talk about using the Forward-Backward algorithm for evaluating a sequence of observatiosn. Second we will delve into how to solve the decoding problem with the Viterbi algorithm works on a conceptual level and finally touch on the idea of learning as just an extension of decoding.

Evaluation: Forward-Backward Algorithm

Evaluation is the question about given a sequence how probable that is. This is important in determining how likely your model actually created the sequence that you are modeling. It can also be quite useful for determining if the sequence *Homepage* → *Homepage* is more probable than *Homepage* → *Signup*. We do this by utilizing an algorithm called the forward-backward algorithm. This algorithm's goal is to figure out what the probability of a hidden state is subject to the observations.

Mathematical representation of the Forward-Backward Algorithm

The Forward Backward Algorithm is the probability of an emission happening given it's underlying states. So that is $P(e_k \mid s)$. On first glance we would have a hard time figuring this out because you would have to compute a lot of probabilities to figure this out. Using the chain rule this could easily become expansive. Instead we can use a simple trick to solve this instead.

The probability of e_k given an observation sequence is proportional to the joint distribution of e_k and the observations.

$$p(e_k \mid s) \propto p(e_k, s)$$

Which we can actually split into two separate pieces using the probability chain rule:

$$p(s_{k+1}, s_{k+2}, \cdots, s_n \mid e_k, s_1, s_2, \cdots, s_k) p(e_k, s_1, s_2, \cdots, s_k)$$

This looks fruitless but we can actually forget about $x_1, \cdots, x_k$ in the first probability because the probabilities are D-Separated. I won't discuss too much what D-Separation is but basically since we're asserting the Markov Assumption in our model we can effectively forget about these variables since they are before what we care about in our probability model.

$p(e_k \mid s) \propto p(s_{k+1}, s_{k+2}, \cdots, s_n \mid e_k) p(e_k, s_1, s_2, \cdots, s_k)$ Which is the forward backward algorithm!

Graphically you can imagine this to be a path through this probability space. Given a specific emission at say index 2 we could calculate the probability by looking at the forward and backward probabilities.

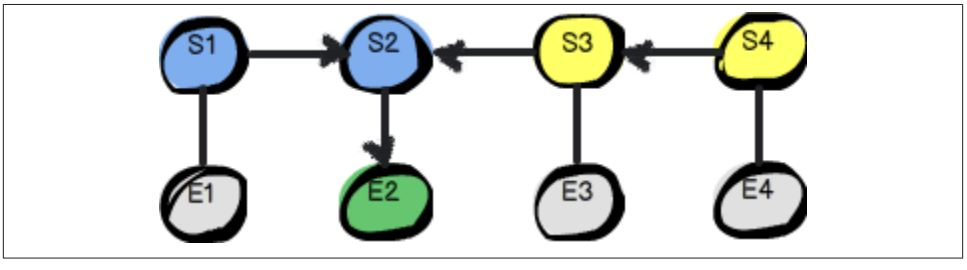


Figure 5-4. Forward Backward algorithm

The first term is looking at the joint probability of the hidden state at point k given all the emissions up to that point. And the backward term is looking at the conditional probability of emissions from k+1 to the end given that hidden point.

Worked out Example using user behavior

Using our above example which is *Homepage* → *Signup* → *Product* → *Checkout* let's calculate what the probability of that happening inside of our algorithm is using the forward algorithm. First let's set up the problem by building a class called Forward-Backward.

```
require 'matrix'
class ForwardBackward
  def initialize
    @observations = ['homepage', 'signup', 'product', 'checkout']

    @states = ['Prospect', 'User', 'Customer']
    @emissions = ['homepage', 'signup', 'product page', 'checkout', 'contact us']
    @start_probability = [0.8, 0.15, 0.05]

    @transition_probability = Matrix[
      [0.8, 0.15, 0.05],
      [0.05, 0.80, 0.15],
      [0.02, 0.95, 0.03]
    ]

    @emission_probability = Matrix[
      [0.4, 0.3, 0.3], # homepage
      [0.1, 0.8, 0.1], # signup
```

```
      [0.1, 0.3, 0.6], # product page
      [0, 0.1, 0.9],   # checkout
      [0.7, 0.1, 0.2]  # contact us
    ]
  end
end
```

Simply we are importing the information that we had from above. The transition probability matrix as well as the emission probabilities. Next we need to define our foward step which is:

```
class ForwardBackward
  # Initialize
  def forward
    forward = []
    f_previous = {}
    @observations.each_with_index do |obs, i|
      f_curr = {}
      @states.each do |state|
        if i.zero?
          prev_f_sum = @start_probability.fetch(state)
        else
          prev_f_sum = @states.reduce(0.0) do |sum, k|
            sum += f_previous.fetch(k, 0.0) * @transition_probability.fetch(k).fetch(state)
          end
        end
        f_curr[state] = @emission_probability.fetch(state).fetch(obs) * prev_f_sum
        forward << f_curr
        f_previous = f_curr
      end
    end

    p_fwd = @states.reduce(0.0) do |sum, k|
      sum += f_previous.fetch(k) * @transition_probability.fetch(k).fetch(@end_state)
    end

    {
      'probability' => p_fwd,
      'sequence' => forward
    }
  end
end
```

The forward algorithm will go through each state at each observation and multiply together to get a forward probability of how the state works in this given context. Next we need to define the backward algorithm which is:

```
class FowardBackward
  # initialize
  # forward

  def backward
    backward = []
```

```
      b_prev = {}

      %w[None].concat(@observations[1..-1].reverse).each_with_index do |x_i_plus, i|
        b_curr = {}
        @states.each do |state|
          if i.zero?
            b_curr[state] = @transition_probability.fetch(state).fetch(@end_state)
          else
            b_curr[state] = @states.reduce(0.0) do |sum, k|
              sum += @transition_probability.fetch(state).fetch(k) * @emission_probability.fetch(k).
            end
          end
        end

        backward.insert(0, b_curr)
        b_prev = b_curr
      end

      p_bkw = @states.reduce(0.0) do |sum, s|
        sum += @start_probability.fetch(s) * @emission_probability.fetch(s).fetch(@observations[0])
      end

      {
        'probability' => p_bkw,
        'sequence' => backward
      }
    end
  end
```

The backward algorithm works pretty much the same way except that it goes the opposite direction. Next we need to try both forward and backward and assert that they are the same otherwise our algorithm is wrong.

```
  class FowardBackward
    # initialize
    # forward
    # backward
    def forward_backward
      size = @observations.length
      fwd, p_fwd = forward.values
      bkwd, p_bkw = backward.values

      # merging the two parts
      posterior = {}
      @states.each do |state|
        posterior[state] = (1..size).map do |i|
          fwd[i][state] * bwkd[i][state] / p_fwd
        end
      end

      return fwd, bkw, posterior
```

```
    end
  end
```

The beauty of the forward backward algorithm is that is effectively testing it self as it runs. Which is quite exciting. This will also solve the problem of evaluation which is to figure out for a given sequence how probable is that likely to be. Next we'll delve into the decoding problem which is figuring out the best sequence of underlying states.

The Decoding Problem through the Viterbi Algorithm

```
The decoding problem is the easiest to describe. Given a sequence of observations and want to pars

To achieve this we use something called the *Viterbi algorithm*. You can think of this as a way of

Graphically it would look something like this.

[[viterbi]]
.A state with high probability will win over time
image::images/viterbi.png[]

What we see in this graphic is how a state like _S1_ will become less relevant over time while a _

What we are attempting to do with this algorithm is traverse a set of states in the most optimal w

The Learning Problem
```

The learning problem is probably the simplest to actually implement. Given a sequence of states and observations then what is the most likely to happen next? We can do that purely by figuring out the next step in the Viterbi sequence. The way one figures out what the next state is is by maximizing the next step given the fact there is no emission available yet. But you can purely figure out the most probable emission from there and also the most probable state and that is the next optimal state emission combo.

If the way of solving this doesn't make sense yet that is fine because in the next section we will delve into using the Viterbi algorithm for part of speech tagging.

Unfortunately there isn't any free and easily accessible data available for analyzing user behaviors over time given pageviews but there is a similar problem we can solve by using a part of speech tagger build purely using a hidden markov model.

Example: Part of Speech Taging with the Brown Corpus

Given the sentence "the quick brown fox." how would you tag those words as parts of speech? We know that english has some parts of speech like determiners, adjectives, and nouns. So we could probably tag this as *determiner*, *adjective*, *adjective*, *noun*. We could do that because we have a basis in grammar but how could we train an algorithm to do that?

Well of course since this is a chapter on hidden Markov models we can use that to figure out the optimal parts of speech. Knowing what we know about hidden markov models we can use the Viterbi algorithm to figure out for a given sequence of words what is the best tagging sequence. For this section we will rely on the Brown Corpus which was the first electronic corpus. It has over a million annotated words with parts of speech in it. The list of tags is long but know that there is all the normal tags like adjectives, nouns, verbs, and others.

The Brown Corpus is setup using a specific kind of annotation. For each sentence of words you will see something like this:

```
Most/ql important/jj of/in all/abn ,/, the/at less/ql developed/vbn coun
tries/nns must/md be/be persuaded/vbn to/to take/vb the/at necessary/jj
steps/nns to/to allocate/vb and/cc commit/vb their/pp$ own/jj resources/
nns ./.
```

In this case *Most* is *ql* which means qualifier, *important* is *jj* (adjective), and on until you hit ./. which is a period tagged as a stop ".".

The only thing that this doesn't have is a *START* character at the beginning. Generally speaking when we're writing markov models we want the word at t and also the word at t - 1. Since most is at the front there is no word before it, so therefore we just use a special name *START* to show that there is a start to this sequence. That way we can measure the probability of going from START to a Qualifier.

Setup Notes

All of the code we're using for this example can be found on GitHub at *https://github.com/thoughtfulml/examples/tree/master/4-hidden-markov-models*.

Ruby is constantly changing so the README is the best place to know how to run the examples.

There are no other dependencies on getting this example to run with Ruby.

The Seam of our Part of Speech Tagger: CorpusParser

The seam of a part of speech tagger is how you feed it data. The important part in this is to be able to feed it proper information so that the part of speech tagger can utilize and learn from data. To get started I have made some of assumptions about how I want this to work. I want to store each transition from word tag combo in an array of two and also wrap in a simple class called CorpusParser::TagWord. My initial test looks like this.

```
# test/lib/corpus_parser_spec.rb
```

```
require 'spec_helper'

describe CorpusParser do
  let (:stream) { "\tSeveral/ap defendants/nns ./.\n" }
  let (:blank) { "\t    \n" }
  it 'will parse a brown corpus line using the standard / notation' do
    cp = CorpusParser.new

    null = CorpusParser::TagWord.new("START", "START")
    several = CorpusParser::TagWord.new("Several", "ap")
    defendants = CorpusParser::TagWord.new("defendants", "nns")
    period = CorpusParser::TagWord.new(".", ".")

    expectations = [
      [null, several],
      [several, defendants],
      [defendants, period]
    ]

    cp.parse(stream) do |ngram|
      ngram.must_equal expectations.shift
    end

    expectations.length.zero?.must_equal true
  end

  it 'does not allow blank lines from happening' do
    cp = CorpusParser.new

    cp.parse(blank) do |ngram|
      raise "Should never happen"
    end
  end
end
```

Simply this takes two cases which is the normal case that comes from Brown and parses that into a sequence of transitions. The second case is a gut check to make sure it ignores blanklines since the brown corpus is full of them.

Filling in the CorpusParser class we would have something that initially looks like this.

```
# lib/corpus_parser.rb

class CorpusParser
  TagWord = Struct.new(:word, :tag)
  NULL_CHARACTER = "START"
  STOP = " \n"
  SPLITTER = '/'

  def initialize
    @ngram = 2
  end
```

```
    def parse(io)
      ngrams = @ngram.times.map { TagWord.new(NULL_CHARACTER, NULL_CHARACTER) }

      word = ''
      pos = ''
      parse_word = true

      io.each_char do |char|
        if char == "\t" || (word.empty? && STOP.include?(char))
          next
        elsif char == SPLITTER
          parse_word = false
        elsif STOP.include?(char)
          ngrams.shift
          ngrams << TagWord.new(word, pos)

          yield ngrams

          word = ''
          pos = ''
          parse_word = true
        elsif parse_word
          word += char
        else
          pos += char
        end
      end

      unless pos.empty? || word.empty?
        ngrams.shift
        ngrams << TagWord.new(word, pos)
        yield ngrams
      end
    end
  end
```

Like the previous chapters implementing a parser using each_char is generally the most performant way of parsing things in Ruby. Now we can get into the much more interesting part writing the part of speech tagger.

The Part of Speech Tagger

With the part of speech tagger we need to be able to do three things. Take data from the CorpusParser, store it internally so we can calculate probabilities of word tag combos, as well as tag transitions. We want this class to be able to tell us how probable a word and tag sequence is as well as determine from a plaintext sentence what the optimal tag sequence is.

To be able to do that we need to first tackle calculating probabilities first, followed by the probability of a tag sequence with word sequence and then finally we'll implement the Viterbi algorithm.

Let's first talk about the probability of a tag given it's previous tag. Using something called a maximum liklihood estimate we can assert that this should equal the count of the two tags together divided by the count of the previous tag. Writing a test for that it would look like this:

```
# test/lib/pos_tagger_spec.rb

require 'spec_helper'
require 'stringio'

describe POSTagger do
  let(:stream) { "A/B C/D C/D A/D A/B ./." }

  let(:pos_tagger) {
    pos_tagger = POSTagger.new([StringIO.new(stream)])
    pos_tagger.train!
    pos_tagger
  }

  it 'calculates tag transition probabilities' do
    pos_tagger.tag_probability("Z", "Z").must_equal 0

    # count(previous_tag, current_tag) / count(previous_tag)
    # count D and D happens 2 times, D happens 3 times so 2/3
    pos_tagger.tag_probability("D", "D").must_equal Rational(2,3)
    pos_tagger.tag_probability("START", "B").must_equal 1
    pos_tagger.tag_probability("B", "D").must_equal Rational(1,2)
    pos_tagger.tag_probability(".", "D").must_equal 0
  end
end
```

Remember that the sequence starts with an implied tag called *START*. So here you see the probability of *D* transitioning to *D* is in fact two divided by three because *D* transitions to *D* three times but *D* shows up three times in that sequence. To make this work we would have to write the following in our POSTagger class.

```
# lib/pos_tagger.rb

class POSTagger
  def initialize(data_io = [])
    @corpus_parser = CorpusParser.new
    @data_io = data_io

    @trained = false
  end

  def train!
```

```
    unless @trained
      @tags = Set.new(["START"])
      @tag_combos = Hash.new(0)
      @tag_frequencies = Hash.new(0)
      @word_tag_combos = Hash.new(0)

      @data_io.each do |io|
        io.each_line do |line|
          @corpus_parser.parse(line) do |ngram|
            write(ngram)
          end
        end
      end
      @trained = true
    end
  end

  def write(ngram)
    if ngram.first.tag == 'START'
      @tag_frequencies['START'] += 1
      @word_tag_combos['START/START'] += 1
    end

    @tags << ngram.last.tag

    @tag_frequencies[ngram.last.tag] += 1
    @word_tag_combos[[ngram.last.word, ngram.last.tag].join("/")] += 1
    @tag_combos[[ngram.first.tag, ngram.last.tag].join("/")] += 1
  end

  # Maximum liklihood estimate
  # count(previous_tag, current_tag) / count(previous_tag)
  def tag_probability(previous_tag, current_tag)
    denom = @tag_frequencies[previous_tag]

    if denom.zero?
      0
    else
      @tag_combos["#{previous_tag}/#{current_tag}"] / denom.to_f
    end
  end
end
```

You'll notice that we're doing a bit of handling of the case when zeros happen only because we will throw a divide by zero error. Next we need to address the probability of word tag combinations and we can do that by introducing the following to our existing test:

```
# test/lib/pos_tagger_spec.rb

describe POSTagger do
  let(:stream) { "A/B C/D C/D A/D A/B ./." }
```

```
  # Maximum Liklihood estimate
  # count (word and tag) / count(tag)
  it 'calculates the probability of a word given a tag' do
    pos_tagger.word_tag_probability("Z", "Z").must_equal 0

    # A and B happens 2 times, count of b happens twice therefore 100%
    pos_tagger.word_tag_probability("A", "B").must_equal 1

    # A and D happens 1 time, count of D happens 3 times so 1/3
    pos_tagger.word_tag_probability("A", "D").must_equal Rational(1,3)

    # START and START happens 1, time, count of start happens 1 so 1
    pos_tagger.word_tag_probability("START", "START").must_equal 1

    pos_tagger.word_tag_probability(".", ".").must_equal 1
  end
end
```

Making this work in the POSTagger we need to write the following.

```
# lib/pos_tagger.rb

class POSTagger
  # initialize
  # train!
  # write
  # tag_probability

  # Maximum Liklihood estimate
  # count (word and tag) / count(tag)
  def word_tag_probability(word, tag)
    denom = @tag_frequencies[tag]

    if denom.zero?
      0
    else
      @word_tag_combos["#{word}/#{tag}"] / denom.to_f
    end
  end
end
```

Now that we have those two things we can answer the one question of given a word and tag sequence how probable is it? That is the probability of the current tag given the previous times the word given the tag. In a test it looks like this.

```
# test/lib/pos_tagger.rb

describe POSTagger do
  it 'calculates probability of sequence of words and tags' do
    words = %w[START A C A A .]
    tags = %w[START B D D B .]
    tagger = pos_tagger
```

```
    tag_probabilities = [
      tagger.tag_probability("B", "D"),
      tagger.tag_probability("D", "D"),
      tagger.tag_probability("D", "B"),
      tagger.tag_probability("B", ".")
    ].reduce(&:*)

    word_probabilities = [
      tagger.word_tag_probability("A", "B"), # 1
      tagger.word_tag_probability("C", "D"),
      tagger.word_tag_probability("A", "D"),
      tagger.word_tag_probability("A", "B"), # 1
    ].reduce(&:*)

    expected = word_probabilities * tag_probabilities

    pos_tagger.probability_of_word_tag(words, tags).must_equal expected
  end
end
```

So basically it is the probabilities of word tag probabilities multiplied by the probability of tag transitions. We can easily implement this in the POSTagger using the following.

```
# lib/pos_tagger.rb

class POSTagger
  # initialize
  # train!
  # write
  # tag_probability
  # word_tag_probability

  def probability_of_word_tag(word_sequence, tag_sequence)
    if word_sequence.length != tag_sequence.length
      raise 'The word and tags must be the same length!'
    end
    # word_sequence %w[START I want to race .]
    # Tag sequence %w[START PRO V TO V .]

    length = word_sequence.length

    probability = Rational(1,1)

    (1...length).each do |i|
      probability *= (
        tag_probability(tag_sequence[i - 1], tag_sequence[i]) *
        word_tag_probability(word_sequence[i], tag_sequence[i])
      )
    end

    probability
  end
end
```

Now we have the ability to figure out how probable a given word and tag sequence is. But really the best would be able to determine given a sentence and training data what the optimal sequence of tags are. For that we need to write a simple test which would look like this:

```
# test/lib/pos_tagger_spec.rb

describe POSTagger do
  describe 'viterbi' do
    let(:training) { "I/PRO want/V to/TO race/V ./. I/PRO like/V cats/N ./." }
    let(:sentence) { 'I want to race.' }
    let(:pos_tagger) {
      pos_tagger = POSTagger.new([StringIO.new(training)])
      pos_tagger.train!
      pos_tagger
    }

    it 'will calculate the best viterbi sequence for I want to race' do
      pos_tagger.viterbi(sentence).must_equal %w[START PRO V TO V .]
    end
  end
end
```

This simple test takes a bit more to implement since the Viterbi algorithm is somewhat involved. So let's go through this step by step. The first problem is that our method accepts a string not a sequence of tokens. We need to split by whitespace and also treat stop characters as their own word. So to do that we can write the following to set up the viterbi algorithm.

```
# lib/pos_tagger.rb

class POSTagger
  # initialize
  # train!
  # write
  # tag_probability
  # word_tag_probability

  def viterbi(sentence)
    parts = sentence.gsub(/[\.\?!]/) {|a| " #{a}" }.split(/\s+/)
  end
end
```

The Viterbi algorithm is an iterative algorithm where at each step it figures out where it should go next based on the previous answer. So we will need to memoize the previous probabilities as well as keep what the best tag was. We can also at this point initialize and figure out what the best is.

```
# lib/pos_tagger.rb

class POSTagger
```

```
    # initialize
    # train!
    # write
    # tag_probability
    # word_tag_probability

    def viterbi(sentence)
      # parts

      last_viterbi = {}
      backpointers = ["START"]

      @tags.each do |tag|
        if tag == 'START'
          next
        else
          probability = (
            tag_probability("START", tag) *
            word_tag_probability(parts.first, tag)
          )

          if probability > 0
            last_viterbi[tag] = probability
          end
        end
      end

      backpointers << (
        last_viterbi.max_by {|k,v| v} ||
        @tag_frequencies.max_by {|k,v| v}
      ).first
    end
  end
```

At this point last_viterbi has only one option which is {"PRO" ⇒ 1.0}. That is because the probability of transitioning from START to anything else is zero. And likewise backpointers will have START and PRO in it. Now that we've set up our initial step all we need to do is iterate through the rest.

```
  # lib/pos_tagger.rb

  class POSTagger
    # initialize
    # train!
    # write
    # tag_probability
    # word_tag_probability

    def viterbi(sentence)
      # parts
      # initialization
```

```
      parts[1..-1].each do |part|
        viterbi = {}
        @tags.each do |tag|
          next if tag == 'START'
          break if last_viterbi.empty?

          best_previous = last_viterbi.max_by do |prev_tag, probability|
            (
              probability *
              tag_probability(prev_tag, tag) *
              word_tag_probability(part, tag)
            )
          end

          best_tag = best_previous.first

          probability = (
            last_viterbi[best_tag] *
            tag_probability(best_tag, tag) *
            word_tag_probability(part, tag)
          )

          if probability > 0
            viterbi[tag] = probability
          end
        end


        last_viterbi = viterbi

        backpointers << (
          last_viterbi.max_by{|k,v| v} ||
          @tag_frequencies.max_by{|k,v| v }
        ).first
      end
      backpointers
    end
  end
```

What we are doing is only storing relevant information and if the case happens where last_viterbi is empty we'll use the @tag_frequencies instead. That case is really when we have pruned too far. But this approach is much faster than storing all of the information in memory.S

At this point things should work! But how well?

Cross Validating to get Confidence in the Model

At this point it is prudent to write a cross validation test. This is using a naïve model but we would want to see at least a 20% accuracy. So let's write this into a 10 fold cross

validation spec. Instead of requiring that this model be within a range of confidence we will just display to the user what the error rate is. In running the test on my machine I got around 30% error rate. We will talk about how to improve this but for my use I think it's good given that it only looks at two probabilities.

```
# test/cross_validation_spec.rb

require 'spec_helper'

describe "Cross Validation" do
  let(:files) { Dir['./data/brown/c***'] }

  FOLDS = 10

  FOLDS.times do |i|
    let(:validation_indexes) do
      splits = files.length / FOLDS
      ((i * splits)..((i + 1) * splits)).to_a
    end

    let(:training_indexes) do
      files.length.times.to_a - validation_indexes
    end

    let(:validation_files) do
      files.select.with_index {|f, i| validation_indexes.include?(i) }
    end

    let(:training_files) do
      files.select.with_index {|f, i| training_indexes.include?(i) }
    end

    it "cross validates with a low error for fold #{i}" do
      pos_tagger = POSTagger.from_filepaths(training_files, true)
      misses = 0
      successes = 0

      validation_files.each do |vf|
        File.open(vf, 'rb').each_line do |l|
          if l =~ /\A\s+\z/
            next
          else
            words = []
            parts_of_speech = ['START']
            l.strip.split(/\s+/).each do |ppp|
              z = ppp.split('/')
              words << z.first
              parts_of_speech << z.last
            end

            tag_seq = pos_tagger.viterbi(words.join(' '))
            misses += tag_seq.zip(parts_of_speech).count {|k,v| k != v }
```

```
            successes += tag_seq.zip(parts_of_speech).count {|k,v| k == v }
          end
        end
        puts Rational(misses, successes + misses).to_f
      end
      skip("Error rate was #{misses / (successes + misses).to_f}")
    end
  end
end
```

This will yield around a 20-30% error rate. Which realistically isn't accurate. Part of the problem though is that the brown corpus makes a lot of distinction of tags so really the actual error rate is much lower if you were to not care about posessive pronouns vs just regular pronouns.

How to make this better?

Like all of our coding examples the best way to make this better is to first determine how well this works and to iterate. A quick way to making this model operate better would be to look back more than one word at a time. So instead of the probability of a tag given the previous tag it would be the probability of a tag given the previous two tags. You could do that by modifying the corpus tagger.

But the example does work well and is simple to make!

Conclusion

Hidden markov models are some of the most interesting models when it relates to determining underlying data from a system given some observable data. For example you can determine the real state of a user, the underlying tag of a word, or even follow musical scores.

In this chapter you learned about how state machines can be generalized into Markov chains, which then can be used to model system behavior forever. As well we added a hidden component to determine underlying state of a model given emissions that we can easily observe. You also learned that the three questions when using hidden Markov models are Evaluation, Decoding, and Learning as well as how to approach solving those problems. Finally we tagged parts of speech using the Brown Corpus and the Viterbi algorithm.

CHAPTER 6

Support Vector Machines

What makes a user loyal or not? Loyalty in commerce generally has to do with users returning to buy things consistently over time. But how do you actually measure the deciding factor between loyal and disloyal?

In this chapter we'll set out to solve that problem conceptually by utilizing support vector machines. This tool can utilize many feature objects while yielding a deciding line between one class or the other. Finally we'll finish up with a worked out example about attaching sentiment to movie reviews.

Solving the Loyalty Mapping Problem

We have two sets of customers, loyal and disloyal. On one hand you have customers who return to the site consistently, and buy from the company while on the other you have either window shoppers, tightwads, or spend thrifts who don't care about the company. The goal we want to achieve is to determine what makes a customer loyal and disloyal with respects to amount of orders and average order size.

Let's imagine our data looks something like this

Graphic showing discernible difference between loyal and disloyal customers

Approaching this problem for the first time there are many ways we could build something to decide whether users are loyal or disloyal. We could utilize a K-Nearest Neighbor classification program, which effectively clusters things together around a centroid.

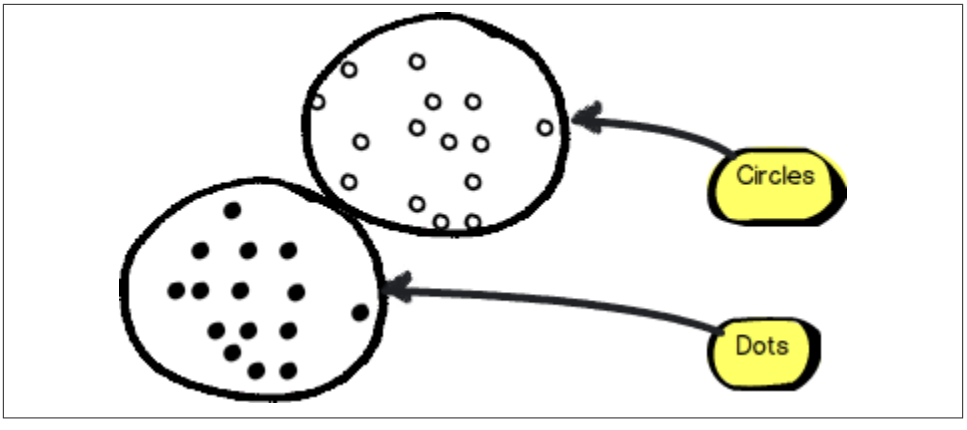


Figure 6-1. Using clustering we'd build a classification like this

But really this is not exactly what we want. We don't want to know what the average loyal user looks like, or the average disloyal user instead we'd prefer to find the *decision boundary* between the two. This *decision boundary* would be just a line drawn between the two classes of data. Unfortunately though this isn't as easy as it sounds because there is an infinite amount of decision boundaries we can draw.

Luckily for us there is an algorithm that can help in this problem called a Support Vector Machine. Support Vector Machines were originally introduced in the 1980s as a way of classifying data. Though the modern interpretation is less stringent than the first and was introduced in 1995. *http://link.springer.com/article/10.1007%2FBF00994018*. The original motivation for Support Vector Machines was to help solve a two-group classification problem. These types of problems can either be boolean (true, false) or ids (3,4) or negative positive (1,-1). What makes Support Vector Machines so special is that it operates well in high dimensions, and avoids the curse of dimensionality. As well as it is fast to compute.

Instead of picking an arbitrary line between two sets of data instead the algorithm maximizes the distance between the two sets of data. So for instance in our loyal vs disloyal problem we would be able to determine the best decision boundary between the two classes. Knowing that decision boundary we could answer the problem of what *makes* a loyal customer vs what doesn't.

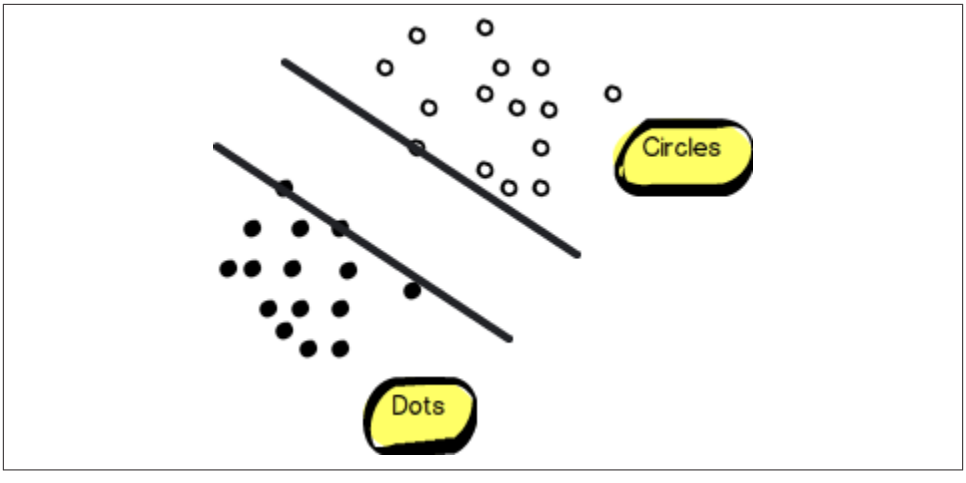


Figure 6-2. Using Support Vector Machines we separate using a margin instead

The other benefits of this algorithm is that it is generally faster to computer, and can operate well into infinite amount of dimensions.

Derivation of SVM's

Conceptually we understand that support vector machines maximize the distance between the two sets but how does it actually do it?

Our goal is to solve for w for this function $wx - b = 0$. This function could also be rewritten to be $b = \sum_{i=1}^{n} w_i x_i$ where x_i is the value at that dimension and w_i is yet to be determined. These equations define a hyperplane or flat surface in an n-dimensional space. Hyperplanes are just fancy words for n-dimensional lines between things. Really what is important to realize here is that we're drawing a flat surface between two sets and determining the distance between the most outlier of datapoints.

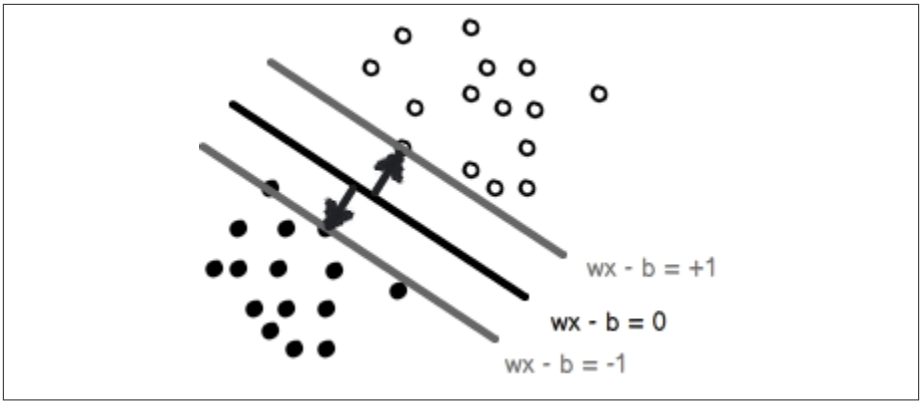


Figure 6-3. Using wx-b=0 we define the plane that separates the two data classes

But this is just an arbitrary space between two classes of data. We want to find the maximum separation of both sets. For that we will need to define two more hyperplanes: one above and one below. In our case we can just define them as $wx + b = 1$ for the hyperplane above and $wx + b = -1$ for the one below. This point we don't know any actual data but we have a margin above and a margin below.

Now that we have three hyperplanes defined we want to maximize the margin between the upper and lower hyperplane. The way we do this is through geometry. We have two parallel lines that we need to find the general distance between. The only way of determining this is by finding a hyperplane that moves perpendicular to all the hyperplanes. In our case the perpendicular segment between the upper and lower hyperplanes happens to be $\frac{2}{||w||}$. Since our original goal is to maximize the margin we can simplify this even further by stating that our goal is to minimize $||w||$ which is the euclidean distance between the two hyperplanes.

This has many benefits which is that $||w|| = \sqrt{w\dotw}$ meaning that this is a convex function. Convex functions are solved for quickly due to many algorithms for this, but even better we can redefine this to be $\frac{1}{2}||w||^2$ which is called a Quadratic Program. Using the Karush-Kuhn Tucker conditions this problem is quickly solved.

At this point we have a way of maximizing the margin between two data sets. On top of that it is a simple quadratic program and can be quickly computed. But there still is one problem and that is what happens when there is data that is not linear.

At this point using support vector machines we know exactly where the decision point is but there's unfortunately an issue which is that most data is not linear.

Non Linear Data

You have probably noticed that all of our examples are linear. Most data as we know isn't linear and has many dimensions. We would like to be able to transform it into something that is more robust, and non-linear. There is a way of doing this with support vector machines and it is called the Kernel Trick. Effectively it converts a linear aspect of the model into a non-linear one, and handles many issues related with non-linear data.

Kernel Trick

Imagine that you are given the following graphic that cannot be separated using any linear decision boundaries.

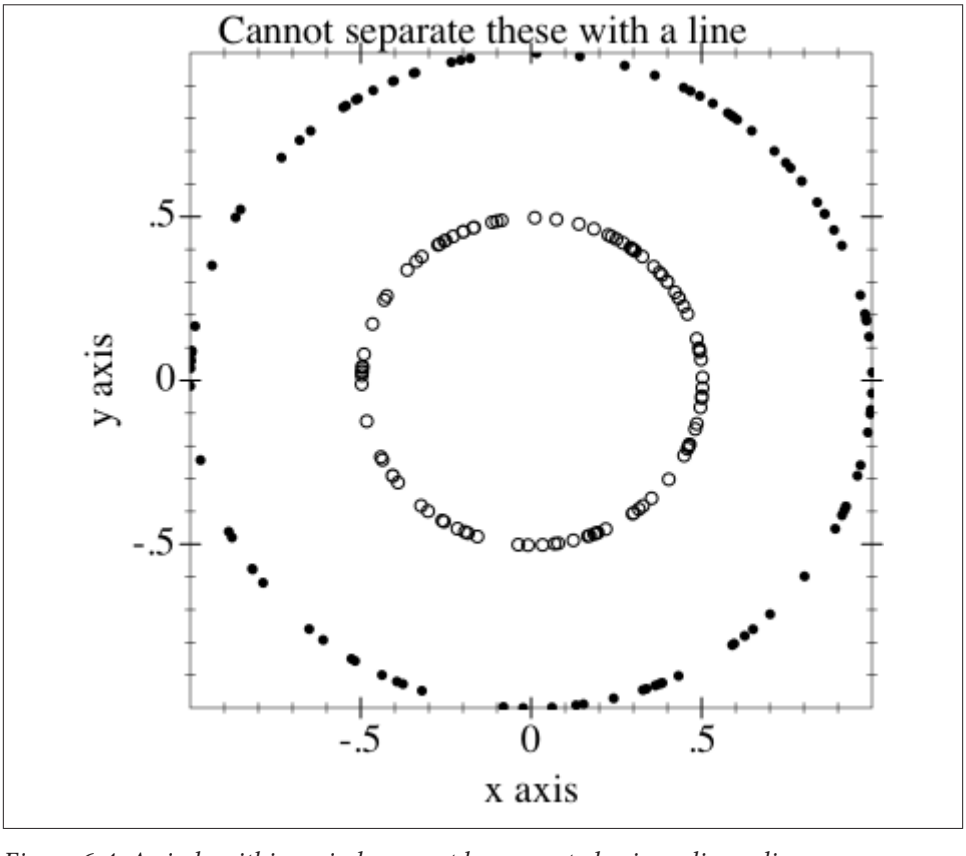


Figure 6-4. A circle within a circle cannot be separated using a linear line

You'll notice that there is no way of drawing a single line to separate this data. Instead we would need something non linear like a circle to actually separate this data into different pieces. Using any sort of linear decision boundary we wouldn't be able to separate the classes here into two different circles. Luckily there is a trick to overcoming this! The Kernel trick. Simply what it does is transforms data from one to another. For instance instead of looking at this circle in a two dimensional space what if we were to transform $< x,\ y >$ to $< x^2, \sqrt{2}xy,\ y^2 >$. What you would see is the following.

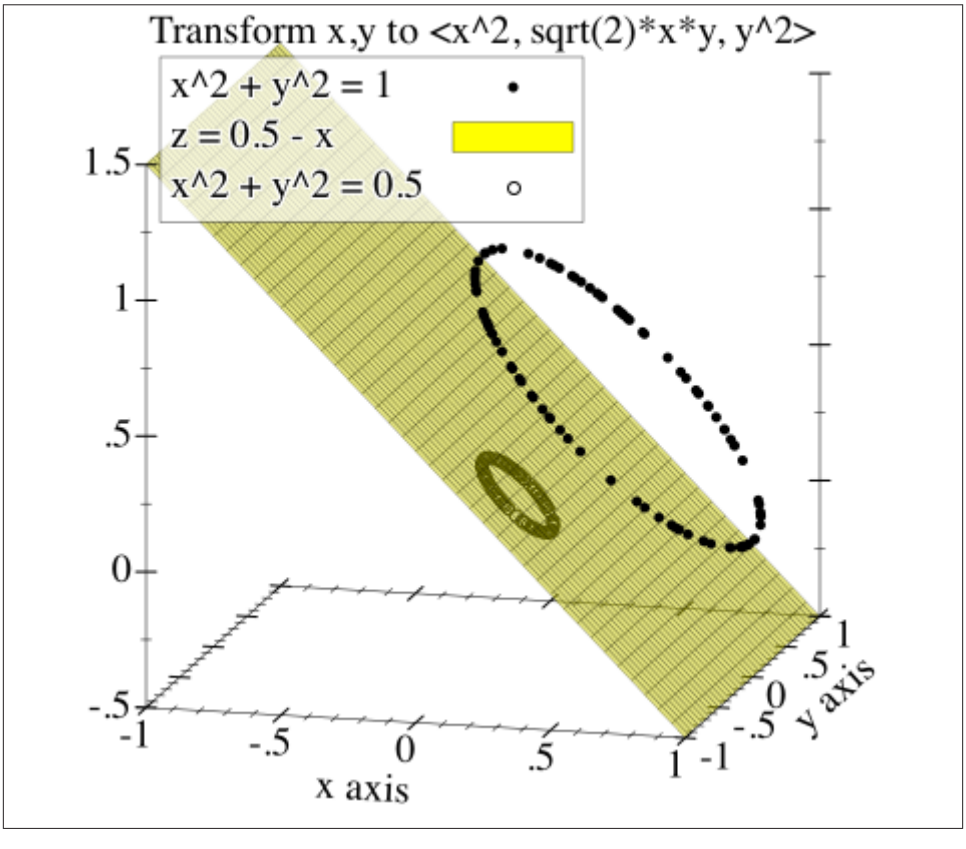


Figure 6-5. This is a different way of looking at the same exact data

This is the kernel trick in a nutshell. Basically taking data that is of one dimension and projecting it into more dimensions that will easily let us use lines to separate data. In this case we are usi ng what is called a polynomial kernel.

The kernel trick in summation is taking a linear X and turning it into $\phi(x)$ or a function of x that suits the data better.

There are a few problems with this approach:

1. $\phi(x_i)\phi(x_j)$ being calculated many times might take a long time
2. ϕ itself might be a complicated function that is hard to compute
3. If the training data is extensive then in general this kernel trick will be slow to compute

So instead of converting all of our x's into $\phi(x)$ perhaps we could try something different where we say $k(x_i, x_j) = \phi(x_j) * \phi(x_i)$. Instead of just mapping the original x to a new function we map the dot product of $phi(x)$.

We would need something that is cheap to compute as well. The good news though is that these actually exist! They are called Kernel Functions. These specific type of functions have the property of having the inner dot product already calculated so avoids that step, they come pre-calculated and are optimized for speed.

These kernel functions also have the good property of: $K(x, y) = < \phi(x), \phi(y) >$.

There are many classes of kernel functions to be used but the most common are:

1. Homogenous Polynomial
2. Heterogenous Polynomial
3. Radial Basis Function

Homogenous Polynomial

The simplest case of a kernel function is a homogenous polynomial. .Homogenous Polynomial $K(x_i, x_j) = (x_i^T x_j)^d$

d is the degree of the polynomial and can really take any number above 0. This function is used extensively in handwriting detection because of it's performance and ease of use.

A polynomial kernel is useful because it uses similarity as well as combinations of data. For instance let's look at the case of $d = 2$.

Expanded out it would look like this. $K(x, y) = (\sum_{i=1}^{n} x_i y_i)^2 = \sum_{i=1}^{n} x_i^2 y_i^2 + \sum_{i=2}^{n} \sum_{j=1}^{i-1} \sqrt{2} x_i y_i \sqrt{2} x_j x_i$

While that looks like gobbledegook there's something very interesting going on here: there is the interaction of $x_i^2 y_i^2$ which is to be expected but also the combinations of $x_i y_i * x_j y_j$. This is very useful for the case where certain dimensions should be grouped together in perception.

Note that in we showed what a homogenous polynomial is. This is using the degree of 2 which in most cases works the best.

Heterogenous Polynomial

Like the homogenous polynomial this introduces a constant c which is strictly greater than 0.

This looks like: $K(x_i, x_j) = (x_i^T x_j + c)^d$

Intuitively this added constant c increases the relevance of higher-order features instead of the lower.

Radial Basis Function

Radial Basis Function which is used more often because of it's high performance in higher dimensions takes the form of:

$$K(x_i, x_j) = exp\left(- \frac{||x_i - x_j||_2^2}{2 * \sigma^2} \right)$$

The numerator is the squared euclidean distance. σ is a free parameter.

Unfortunately there isn't a neat way of visualizing the Radial Basis Function. The thing to note here is that the Kernel will effectively create infinite amount of new dimensions instead of the case with homogenous polynomial only creating one more dimension. This is actually a huge feature for RBF functions because you overcome the issues related with the curse of dimensionality just by using an RBF function.

When should you use each kernel?

The discussion of when to use a heterogeneous polynomial vs a *radial basis function* is a tricky one. Many times libraries that work with support vector machines utilize *radial basis functions* as a default. But this isn't great reason to use something. According to [*http://cbio.ensmp.fr/~jvert/svn/bibli/local/Smola1998connection.pdf*] these kernels have the ability to regularize your data and basically add a low pass filter on it.

Conceptually a polynomial kernel of degree 2 will take data from 2 dimensions to 3 trying to fit the flatest surface to that. Many times though it is unfortunately thought that RBF's and polynomial kernels are needed when in fact they are overfitting their data.

Soft Margins

There is a problem with what we have discussed so far which is that things need to be purely separable. Imagine the case where our customer is not really loyal but appears loyal. Instead of picking a complex polynomial or RBF to fit this really there's just an error.

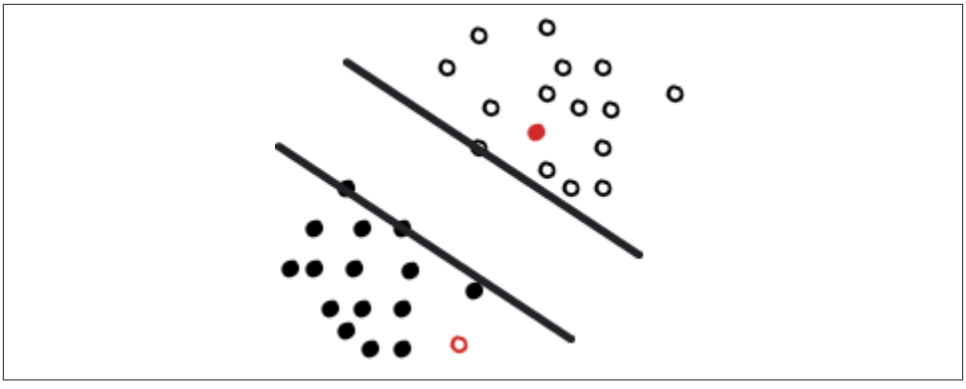


Figure 6-6. Errors happen which is why slack is important

In this case our disloyal customer is in the loyal camp. The data is still linearly separable but there is just a few errors. Instead of trying to find the perfect kernel function we should really just ignore those errors. Originally when Vapnik introduced Support Vector Machines the assumption was that data was separable using either a linear function or some kind of kernel function. But in most cases data isn't that pretty instead it looks something more like the troop who likes to visit other territories.

Vapnik followed up with this[cite], in 1995 to introduce a more robust model which introduced some changes:

1. Optimize with Slack
2. Introduce a new parameter called *C*.

Optimizing with Slack

In all of our above examples you'll notice that data looks pretty but if we were to optimize something that wasn't purely separated then our optimization problems would break down quickly. There is an elegant way of optimizing this by introducing something called a *slack variable*.

Instead of the optimization problem purely minimizing $\frac{1}{2} \mid\mid w \mid\mid^2$ now we'll introduce the variables ξ_i which will trade off the error. These *slack variables* are really just margin errors. Since we want to minimize margin errors as well we can add this into our optimization problem as: $\frac{1}{2} \mid\mid w \mid\mid^2 + \sum_{i=1}^{n} \xi_i$. Effectively we are adding the error parameters to the minimization problem.

Trading off Margin Maximization with Slack Variable Minimization using *C*

In the above minimization problem you'll notice that we are weighting the maximization of margins or $\frac{1}{2} \mid\mid w \mid\mid^2$ equal with our sum of slack variables or the margin errors.

As you know Machine Learning is full of trade offs and this might work and it might not.

Instead what Vapnik did was introduce a complexity parameter which could weight the slack variables. This parameter which is user defined, meaning you define it, weights the slack variables against the original margin maximization problem.

Simply our minimization function is now:

$\frac{1}{2} \mid \mid w \mid \mid ^2 + C\Sigma_{i=1}^{n} \xi_i$

Practically this complexity parameter is positive, and can generally be found through cross validation. We will get more to how to find this complexity parameter in the next section, where we will build a sentiment analyzer using support vector machines!

Using Support Vector Machines to Determine Sentiment: Example

Imagine that you want to attach sentiment to a string of words. How would you go about doing that? Language is full of contextual clues, sarcasm, and lots of complexity. Determining the sentiment of a sentence like "Let's get stupid" is ambiguous because it could be either positive or negative.

In this section we'll build a sentiment analyzer that determines the sentiment of movie reviews. We'll first talk about conceptually what this tool will look like in a class diagram. Then after identifying the pieces of the tool we will delve into building a Corpus class, a CorpusSet class and finally the SentimentClassifier class. Corpus and CorpusSet are about transforming the text into numerical information. SentimentClassifier is where we will then use the support vector machine algorithm to build this sentiment analyzer.

Setup Notes

All of the code we are using for this example can be found on GitHub at *https://github.com/thoughtfulml/examples/tree/master/5-support-vector-machines*

Ruby is constantly changing so the README is the best place to come up to speed on running the examples.

There are no additional dependencies except a running Ruby version to run this example.

Class Diagram

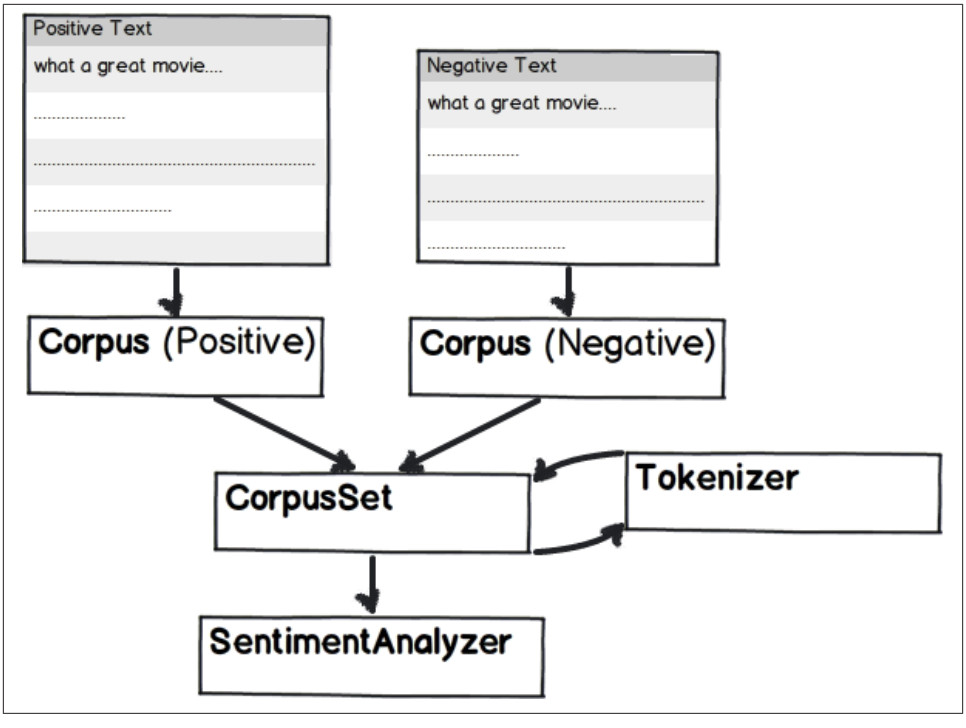


Figure 6-7. General Class Diagram of how to Analyze sentiment with an SVM

Our tool will take a set of training words and sentences that are either negative or positive. Using these sentences we will then build up a corpus of information. Into a Corpus class we have either negative or positive leaning text delimited by stop symbols (punctuation symbols like !, ?, or .). Once we have built up a Corpus of negative or positive leaning text we need to put them together into a CorpusSet class. This merely puts two or more CorpusSet together into one object.

From there the CorpusSet class will then be used by the SentimentClassifier model to build a sentiment analysis guess for use in the future.

What does Corpus and Corpora mean?

Corpus like corpse means a body of some kind except corpus means a body of writings. This word is used heavily in the natural language processing community to signal that this is a big group of previous writings that can be used to infer knowledge. In our case we are using the word corpus to signal a body of writings around a certain sentiment.

Corpora is the plural of corpus.

Corpus Class

There are a lot of questions that our Corpus class needs to answer:

1. Tokenization of text
2. Sentiment leaning whether :negative or :positive
3. Mapping from sentiment leaning to a numerical value
4. Return a unique set of words from the corpus

Tokenization of Text

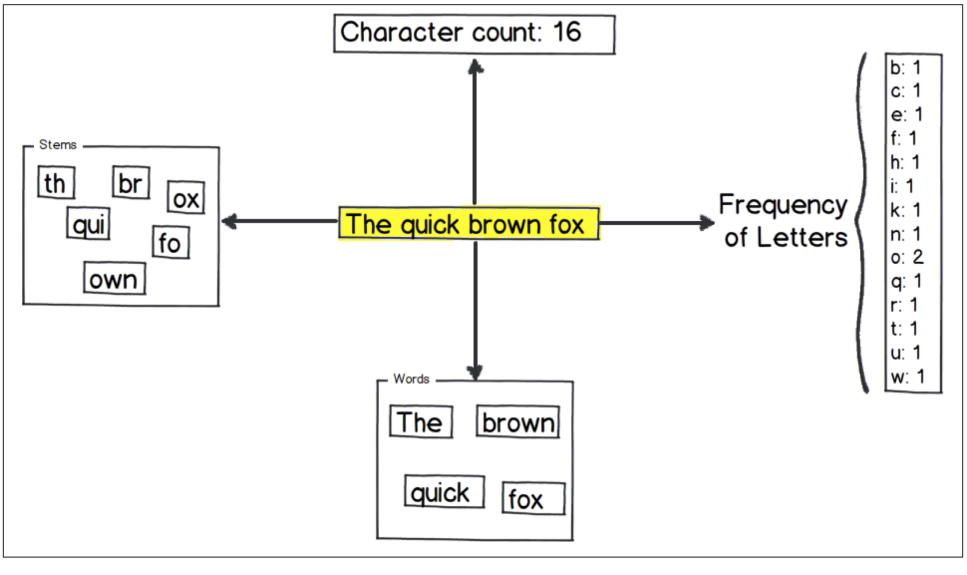


Figure 6-8. Many ways to tokenize sentences and text

There are many different ways of tokenizing text: extracting out stems, frequency of letters, emoticons, words, and many others. For our purposes we will just tokenize words. These are defined as strings that are between non alpha characters. So out of a string like "The quick brown fox." we would extract out ["the", "quick", "brown", "fox"]. As well we don't care about punctuation and we want to be able to skip unicode spaces and non words.

Writing a test about our tokenization we would have something that looks like this:

```
# test/lib/corpus_spec.rb

require 'spec_helper'
require 'stringio'
```

```
describe Corpus do
  describe "tokenize" do
    it "downcases all the word tokens" do
      Corpus.tokenize("Quick Brown Fox").must_equal %w[quick brown fox]
    end

    it "ignores all stop symbols" do
      Corpus.tokenize("\"'hello!?!?!.'\"  ").must_equal %w[hello]
    end

    it "ignores the unicode space" do
      Corpus.tokenize("hello\u00A0bob").must_equal %w[hello bob]
    end
  end
end
```

From here we can build out our class method ::tokenize to actually work given our test cases.

```
# lib/corpus.rb

class Corpus
  def self.tokenize(string)
    string.downcase.gsub(/['"\.\?\!]/, '').split(/[[:space:]]/)
  end
end
```

We are using // which in Ruby means that we will split by unicode and non-unicode spaces.

While you could spend a lot of time optimizing this tokenization step this works well enough for our purposes. Many times in machine learning getting more data points beats optimizing the minute details.

Sentiment Leaning :positive or :negative

For each Corpus we need to attach a certain leaning whether it's positive or negative. In most cases we could attach just an arbitrary number like -1 or 1 but since those are arbitrary we should use something more specific. Instead let's use symbols :positive and :negative.

All we care about is whether the symbol comes back so a test for that would look like

```
# test/lib/corpus_spec.rb

describe Corpus do
  let(:positive) { StringIO.new('loved movie!! loved') }
  let(:positive_corpus) { Corpus.new(positive, :positive) }
```

```
  it 'consumes a positive training set' do
    positive_corpus.sentiment.must_equal :positive
  end
end
```

To make this work lets build a skeleton class that takes in file and sentiment and makes an *attr_reader* for *:sentiment*. Like so:

```
# lib/corpus.rb

class Corpus
  # tokenize

  attr_reader :sentiment
  def initialize(file, sentiment)
    @file = file
    @sentiment = sentiment
  end
end
```

That was a simple fix. Notice that we have also introduced a file for the Corpus which will point at our training files.

Sentiment Codes for :positive and :negative

As you can imagine above there is an issue with our code. You could pass anything in as a sentiment and really it's not mapping to any numerical sentiment whether -1 or 1. Really useless for our purposes if we want to use the support vector machine algorithm. Instead we should test for the two cases that are correct and one that is incorrect.

```
# test/lib/corpus_spec.rb

describe Corpus do
  it 'defines a sentiment_code of 1 for positive' do
    Corpus.new(StringIO.new(''), :positive).sentiment_code.must_equal 1
  end

  it 'defines a sentiment_code of 1 for positive' do
    Corpus.new(StringIO.new(''), :negative).sentiment_code.must_equal -1
  end
end
```

This is a simplistic mapping we will use for our Support Vector Machine to train against. To achieve this we just write the following:

```
# lib/corpus.rb

class Corpus
  # initialize
  # tokenize

  attr_reader :sentiment
```

```
  def sentiment_code
    {
      :positive => 1,
      :negative => -1
    }.fetch(@sentiment)
  end
end
```

Return a unique set of words from the corpus

Now we have one last step which is to return the unique set of words from this training file. The goal here is to return words that are located in the corpus so that the CorpusSet class can take this and zip CorpusSet together. Since in most cases we are assuming files are being brought in we can use a StringIO object to act like a file instead therefore mitigating the need to write tempfiles.

```
# test/lib/corpus_spec.rb

describe Corpus do
  let(:positive) { StringIO.new('loved movie!! loved') }
  let(:positive_corpus) { Corpus.new(positive, :positive) }

  it 'consumes a positive training set and unique set of words' do
    positive_corpus.words.must_equal Set.new(%w[loved movie])
  end
end
```

At this point we need to implement the method *#words* on our Corpus. It should return a set of words that have been passed in. This would probably look something like:

```
# lib/corpus.rb

class Corpus
  # initialize
  # tokenize
  # sentiment_code

  attr_reader :sentiment

  def words
    @words ||= begin
      set = Set.new
      @io.each_line do |line|
        Corpus.tokenize(line).each do |word|
          set << word
        end
      end
      @io.rewind
      set
    end
```

```
    end
  end
```

Now that we have taken care of tokenization, as well as storing all words into a unique set we can move onto the only seam of our tool the CorpusSet class.

The CorpusSet Class

Next we need to define what the CorpusSet class looks like. This class takes multiple Corpus objects and zips them together into one set of words as well as builds a vector for the SentimentClassifier to use. This is where the seam exists between our data and the Support Vector Machine is.

We need to test these things:

1. Zipping up Two Corpus objects
2. Building a sparse vector of both corpora

Zip up two corpus objects

Our first test case will be about taking two Corpus objects and combining them into one CorpusSet class. A test case will look like this:

```
# test/lib/corpus_set_spec.rb

require 'spec_helper'

describe CorpusSet do
  let(:positive) { StringIO.new('I love this country') }
  let(:negative) { StringIO.new('I hate this man') }

  let(:positive_corp) { Corpus.new(positive, :positive) }
  let(:negative_corp) { Corpus.new(negative, :negative) }

  let(:corpus_set) { CorpusSet.new([positive_corp, negative_corp]) }

  it 'composes two corpuses together' do
    corpus_set.words.must_equal %w[love country hate man]
  end
end
```

This test takes two different Corpus objects and mixes them together into the word set. Building this on the CorpusSet side we would have something similar to this.

```
# lib/corpus_set.rb

class CorpusSet
  attr_reader :words

  def initialize(corpora)
```

```
      @corpora = corpora
      @words = corpora.reduce(Set.new) do |set, corpus|
        set.merge(corpus.words)
      end.to_a
    end
  end
```

This is a simple idea just merge the sets together into one big set. But now we need to define the seam that will tie into our SentimentClassifier.

Build Sparse vector that ties into SentimentClassifier

At this point we have a set of words that are in our CorpusSet and we need a way of translating them to something that a support vector machine can use. The most common way of doing this would be to convert strings incoming into a vector of ones and zeros. For instance imagine if we have the corpus of "The quick brown fox" and we want to determine the vector for "the fox". The first step would be to take "the quick brown fox" and split it up into indices like this:

Table 6-1. CorpusSet words

Word	Index
the	0
quick	1
brown	2
fox	3

Now that we know what indices are attached to what we can take our string "the fox" and make a new row vector which would look like this:

Table 6-2. Row Vector

Words	the	quick	brown	fox
Indices	0	1	2	3
"the fox"	1	0	0	1

So in the case of "the fox" we only set the index 0 and 3 to 1. This seems like a good idea until you realize that most times a corpus of training data can contain thousands of words and indices. So our row vectors would be over 90% zeros and under 10% ones for each string. Instead we should think about using a sparse vector or Hash in Ruby.

Sparse Vector and Matrixes

Sparse Vectors are simply a compression technique for storing vectors or matrixes. Say for instance you have the vector written in ruby:

```
require 'objspace'
sparse_array = 30_000.times.map {|i| (i % 3000 == 0) ? 1 : 0}
sparse_array.size #=> 30000
ObjectSpace.memsize_of(sparse_array) #=> 302,672 bytes
```

This is a thirty thousand length array! 29,990 of those are just zeros. Instead of storing all of those zeros we can transform it into a hash that only stores index relationships where the number is non zero.

```
sparse_hash = Hash.new(0)

sparse_array.each_with_index do |val, i|
  if val.nonzero?
    sparse_hash[i] = val
  else
    # Skip
  end
end

sparse_hash.size #=> 10
ObjectSpace.memsize_of(sparse_hash) #=> 616 bytes
```

Notice the enormous reduction in size. We went from thirty thousand to ten! Sparse vectors can also be generalized to be used with matrixes as well.

```
require 'matrix'
require 'objspace'
matrix = Matrix.build(300, 100) do |row, col|
  if  row % 3 == 0 && col % 300 == 0
    1
  else
    0
  end
end

matrix.row_size * matrix.column_size #=> 30000

# memsize_of doesn't work unless it's a C level object like an array
ObjectSpace.memsize_of(matrix.to_a.flatten) #=> 312,968 bytes

sparse_matrix = Hash.new(0)

matrix.each_with_index do |e, row, col|
  if e.nonzero?
    sparse_matrix[[row, col]] = e
  else
    # This is zero and therefore we skip it
  end
end

sparse_matrix.length #=> 100
ObjectSpace.memsize_of(sparse_matrix) #=> 5,144 bytes
```

> Using a sparse vector is important for memory considerations and speed cause there's no need to store something that takes up more space than needed.

Using a sparse hash instead of a vector let's build a seam test that ensures that our sentiment analyzer receives the proper information from a CorpusSet. In a test it would look like this:

```
# test/lib/corpus_set_spec.rb

describe CorpusSet do
  it 'returns a set of sparse vectors to train on' do
    expected_ys = [1, -1]
    expected_xes = [[0,1], [2,3]]
    expected_xes.map! do |x|
      Libsvm::Node.features(Hash[x.map {|i| [i, 1]}])
    end

    ys, xes = corpus_set.to_sparse_vectors

    ys.must_equal expected_ys

    xes.flatten.zip(expected_xes.flatten).each do |x, xp|
      x.value.must_equal xp.value
      x.index.must_equal xp.index
    end
  end
end
```

To implement this we would write the code in the following fashion:

```
# lib/corpus_set.rb

class CorpusSet
  # initialize

  attr_reader :words

  def to_sparse_vectors
    calculate_sparse_vectors!
    [@yes, @xes]
  end

  private
  def calculate_sparse_vectors!
    return if @state == :calculated
    @yes = []
    @xes = []
    @corpora.each do |corpus|
      vectors = load_corpus(corpus)
      @xes.concat(vectors)
      @yes.concat([corpus.sentiment_code] * vectors.length)
```

```
    end
    @state = :calculated
  end

  def load_corpus(corpus)
    vectors = []
    corpus.sentences do |sentence|
      vectors << sparse_vector(sentence)
    end
    vectors
  end
end
```

Now that the CorpusSet can receive multiple Corpus objects and convert that into sparse hashes of information for the SentimentClassifier to use. This is where we will actually start using the Support Vector Machine algorithm, and train the data.

The SentimentClassifier class

Now that we have the portion of the app that takes training data of both positive and negative text we can build the support vector machine portion of the app (the SentimentClassifier class). The point of this class is to take information from a CorpusSet and convert it into a support vector machine model. After it has done that this serves as a way of taking new information and mapping either to :negative or :negative.

At this point we have a few issues to still resolve before our tool will work:

1. The API for CorpusSet is complicated to use.
2. We need something to handle the Support Vector Machine algorithm
3. We need something to train on and cross validate

Refactoring the interaction with CorpusSet

The SentimentClssifier takes one argument which is a CorpusSet. This is simply the corpus set of all training data. Unfortunately with having our argument be a CorpusSet we might run into the following type of syntax:

```
# lib/sentiment_classifier.rb

class SentimentClassifier
  def initialize(corpus_set)
    # Initialization
  end
end

positive = Corpus.new(positive_file_path, :positive)
negative = Corpus.new(negative_file_path, :negative)
corpus_set = CorpusSet.new([positive, negative])
classifier = SentimentClassifier.new(corpus_set)
```

This is not good API design. It requires a lot of previous information, to build a Corpus, and then a CorpusSet. In reality what we want is something more like a factory method that builds a SentimentClassifier. This method *.build* would take multiple arguments pointing at training data. Instead of passing in a hash we'll just assume that positive text will have the file extension *.pos* and negative will have *.neg*.

Making a factory method called *.build* will really help us and doesn't require us to explicitly build everything and relies on the filesystem type we can fill in the blanks with our *.build* class method.

```
# lib/sentiment_classifier.rb

class SentimentClassifier
  def self.build(files)
    mapping = {
      '.pos' => :positive,
      '.neg' => :negative
    }

    corpora = files.map { |file| Corpus.new(file, mapping.fetch(File.extname(file)) }
    corpus_set = CorpusSet.new(corpora)

    new(corpus_set)
  end
end
```

Now that we're at this conjunction we still have a few questions that we need to answer: what library to use to build our SVM model, and where to find training data.

Library to handle Support Vector Machines: LibSVM

When it comes to support vector machines generally most people grab LibSVM. It has the longest track record, is written in C and has many bindings, from python, to java, to ruby. One caveat here though is that there are a few Rubygems for LibSVM and not all are superb. The gem *rb-libsvm* which is what we will use supports sparse vectors and therefore is the best suited for our problems. There are other out there using swig adapters which unfortunately don't support sparse matrices as well.

Training Data

Up until this point we haven't talked about training data for our tool. We need some text that is mapped as either negative or positive. These data would be organized into lines and stored in files. There are many different sources of data but what we'll use is from here:https://github.com/jperla/sentiment-data. Inside of this there is a set of data from Pang Lee about movie reviews sentiment.

This is a highly specific dataset and will only work for movie reviews from IMDB but it is good enough to explain things. If you were to use this with any other program most likely you would use a specific dataset to what you were trying to solve. So for instance

twitter sentiment would come from actual tweets that were mapped to negative and positive. Remember that it's not too difficult to build your own data set by building a survey form and partitioning out work to Mechanical Turk by Amazon.

Cross Validating with the Movie Review Data

Cross validation is the best way to ensure that our data is trained well and that our model works properly. The basic idea is to take a big data set split it into two or more pieces and then use one of those pieces of data as training while using the other to validate against it.

In a test form it would look more like this:

```
# test/cross_validation_spec.rb

describe 'Cross Validation' do
  include TestMacros

  def self.test_order
    :alpha
  end

  (-15..15).each do |exponent|
    it "runs cross validation for C=#{2**exponent}" do
      neg = split_file("./config/rt-polaritydata/rt-polarity.neg")
      pos = split_file("./config/rt-polaritydata/rt-polarity.pos")

      classifier = SentimentClassifier.build([
        neg.fetch(:training),
        pos.fetch(:training)
      ])

      # find the minimum

      c = 2 ** exponent
      classifier.c = c

      n_er = validate(classifier, neg.fetch(:validation), :negative)
      p_er = validate(classifier, pos.fetch(:validation), :positive)
      total = Rational(n_er.numerator + p_er.numerator, n_er.denominator + p_er.denominator)

      skip("Total error rate for C=#{2 ** exponent} is: #{total.to_f}")
    end
  end
end
```

We're using skip and self.test_order here. The *skip* method is used to skip over a test and basically say that while it is not finished it ran. This is useful for reporting sake since these are not true or false they just are. This method while somewhat of a mismatch works for this case for us to see results. Also notice that we override test_order here and set it to alpha. That is because mini-test by default uses :random order. Meaning that as

we're going through the series from -15 to 15 we will get data out of order. IT is much easier to interpret results when looking at them in order.

Also notice that we have introduced two new methods split_file and validate. These are in our test macros module as:

```
# test/test_macros.rb

module TestMacros
  def validate(classifier, file, sentiment)
    total = 0
    misses = 0

    File.open(file, 'rb').each_line do |line|
      if classifier.classify(line) != sentiment
        misses += 1
      else
      end
      total += 1
    end

    Rational(misses, total)
  end

  def split_file(filepath)
    ext = File.extname(filepath)
    validation = File.open("./test/fixtures/validation#{ext}", "wb")
    training = File.open("./test/fixtures/training#{ext}", "wb")

    counter = 0
    File.open(filepath, 'rb').each_line do |l|
      if (counter) % 2 == 0
        validation.write(l)
      else
        training.write(l)
      end
      counter += 1
    end
    training.close
    validation.close
  end
end
```

In this test we iterate through 2 to the -15 all the way up to 15. This will cover most of the territory we want. And after the cross validation is done we can pick the best C and use that for our model. Technically speaking this is called a grid search which will attempt to find a good enough solution over a set of trial runs.

To make this work we need to work on the backend of how the SentimentClassifier works. This is where we end up using LibSVM by building our model and making a tiny state machine.

```
# lib/sentiment_classifier.rb

class SentimentClassifier
  # build
  def initialize(corpus_set)
    @corpus_set = corpus_set
    @c = 2 ** 7
  end

  def c=(cc)
    @c = cc
    @model = nil
  end

  def words
    @corpus_set.words
  end

  def classify(string)
    if trained?
      prediction = @model.predict(@corpus_set.sparse_vector(string))
      present_answer(prediction)
    else
      @model = model
      classify(string)
    end
  end

  def trained?
    !!@model
  end

  def model
    puts 'starting to get sparse vectors'
    y_vec, x_mat = @corpus_set.to_sparse_vectors

    prob = Libsvm::Problem.new
    parameter = Libsvm::SvmParameter.new
    parameter.cache_size = 1000

    parameter.gamma = Rational(1, y_vec.length).to_f
    parameter.eps = 0.001

    parameter.c = @c
    parameter.kernel_type = Libsvm::KernelType::LINEAR

    prob.set_examples(y_vec, x_mat)
    Libsvm::Model.train(prob, parameter)
  end
end
```

This is where things get more interesting and we actually build the support vector machine to work with the rest of the problem. As you can see we are using the LibSVM library which is a standard. We first build our sparse_vectors load up a new LibSV problem and give it very default parameters.

After running the cross validation we see that the best C is 128 which happens to create a ~30% error rate.

Improving results over time.

There are a few different ways of improving the 30% error rate which involves experimentation.

A few possibilities are

1. Stripping out stop words
2. Improvement of tokenization
3. Different Kernels Polynomial

You could also investigate trying out a few other algorithms with the same data to see which one works better or not.

Conclusion

Support Vector Machines are very well suited for classifying two separable classes. They can be modified to separate more than 2 classes but also are well suited because they don't suffer from the same curse of dimensionality that K-Nearest-Neighbors does. This chapter you learned how SVM's can be used to separate out who is loyal and who isn't as well as how to assign sentiment to movie data.

CHAPTER 7

Neural Networks

Neural networks are effective at mapping observed data to a function. Researchers have been able to use neural networks for things like handwriting detection, computer vision, and speech recognition with breakthrough results.

Essentially, neural nets are an effective way of learning from data and has had a long history dating back to the 1800s. In this chapter, we're going to discuss how neural nets came to be, what goes into them and how they work as well as a practical example of classifying languages based on character frequencies.

Neural Networks in Brief

Neural Networks are excellent at function approximation and supervised learning problems. They have little restrictions on what they can do and have proven quite successful in practice. They do have the limitation of operating on binary inputs and can suffer from a complexity and speed problem.

History of Neural Networks

Neural networks when introduced were about studying how the brain operates. Neurons, which are in our brains, work together in a network to process and make sense of inputs and stimuli. Alexander Bain and William James both proposed that brains operated in a network that could process lots of information. This network of neurons, has the ability to recognize patterns and learn from previous data. For instance, if a child is shown a picture of 8 dogs they start to understand what a dog looks like.

This research was expanded to include a more artificial bent when Warren McCulloch and Walter Pitts invented threshold logic. Threshold logic combines binary information together to determine logical truthness. They suggested using something called a step

function which attached a threshold to either accept or reject a summation of previous information.

After many years of research, neural networks and threshold logic were combined to form what we call an Artificial Neural Network. This is using the paradigm of a network to recognize patterns, as well as the threshold logic studied by McCulloch and Pitts

What is an Artificial Neural Network?

A neural network is a robust function that takes an arbitrary set of inputs and fits it to an aribrary set of outputs that are binary. They are excellent at fuzzy matching, and building robust functions to match just about anything. In practice they are used in deep learning research to match images to features and much more.

What makes Neural Networks special is their use of a hidden layer of weighted functions called neurons. Without a hidden layer of functions they would be a set of simple weighted functions, but instead with a hidden layer (or more) you can effectively build a function that maps a lot of functions.

Neural networks are also denoted by the number of neurons per layer. For example: If we have 20 neurons in our input layer, 10 in one hidden layer, and 5 in an output layer; it would be a 20-10-5 network. If there is more than one hidden layer, then we would denote it as 20-7-7-5 - the two middle 7s being hidden layers with 7 nodes a piece.

Neural networks have 5 major parts

1. The Input Layer
2. The Hidden Layer(s)
3. Neurons
4. The Output Layer
5. The Training Algorithm

In this section we'll explain what each of these parts do and how they work.

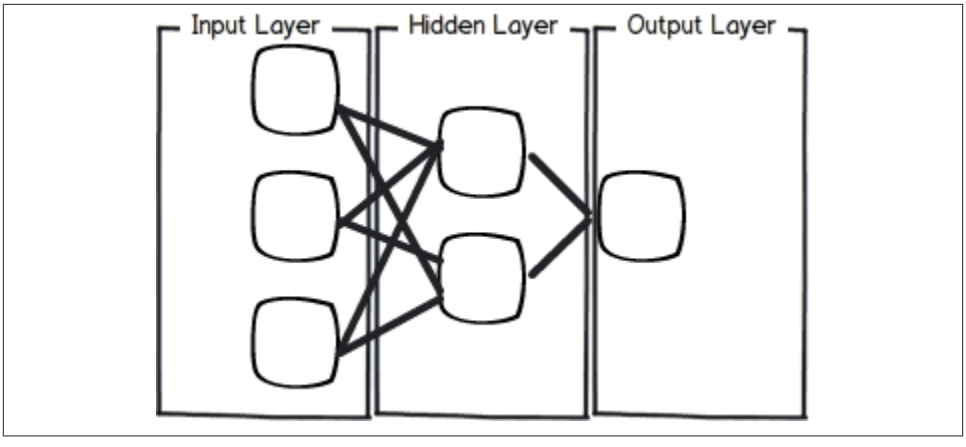


Figure 7-1. A visual representation of a artificial neural network looks like this

Input Layer

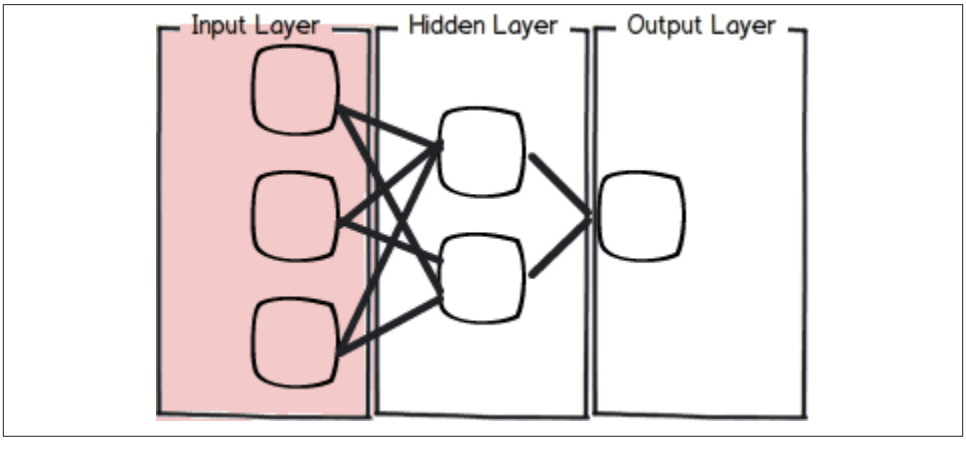


Figure 7-2. The input layer of a Neural Net

The input layer is the entry point of a neural network. It is where the inputs you are giving to the model enter. There are no neurons in this layer because its main purpose is to serve as a conduit to the hidden layers. The type of input is important as neural networks only work with two types: it either needs to be symmetric or standard.

With training a neural network we have observations and inputs. Taking the simple example of an exclusive or (also known as XOR) we have the following truth table:

Table 7-1. XOR Truth Table

Input A	Input B	Output
false	false	false
false	true	true
true	false	true
true	true	false

In this case we would have four observations and two inputs which could either be true or false. Neural networks do not work off of true or false though and knowing how to code it is key. For that we translate to either a standard or symmetric inputs.

Standard Inputs

The standard range for input values is between 0 and 1. In our XOR example above we would code true as 1 and false as 0.

This style of input has one downside which is that if your data is sparse, meaning full of zeros, this can skew results. Having a data set with lots of zeros means we risk the model breaking down. Only use standard if you know that there isn't sparse data.

Symmetric Inputs

Symmetric inputs avoids the issue with zeros. These are between -1 and 1. In our above example -1 would be false and 1 would be true.

This has the benefit of our model not breaking down with lots of a zeroing out effect. In addition to that, we also put less emphasis on the middle of a distribution of inputs. If we introduced a "maybe" into the XOR calculation we could map that as 0 and ignore it.

Inputs can be used in either the symmetric or standard format but need to be marked as such since the way we calculate the output of neurons depends on this.

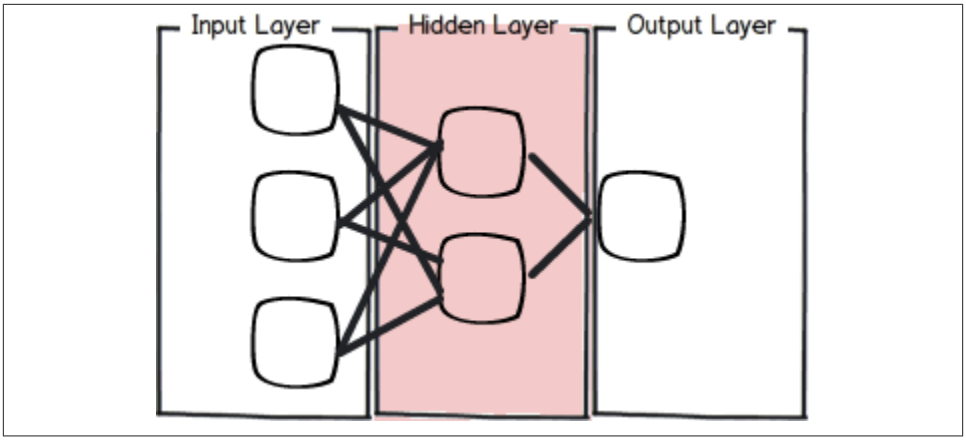


Figure 7-3. Hidden Layer of a Neural Network

Hidden Layers

Without hidden layers neural networks would be a set of weighted linear combinations. Neural networks have the ability to model non linear data because there are hidden layers. Each hidden layer contains a set of neurons which then passes to the output layer. Hidden layers are a group of neurons.

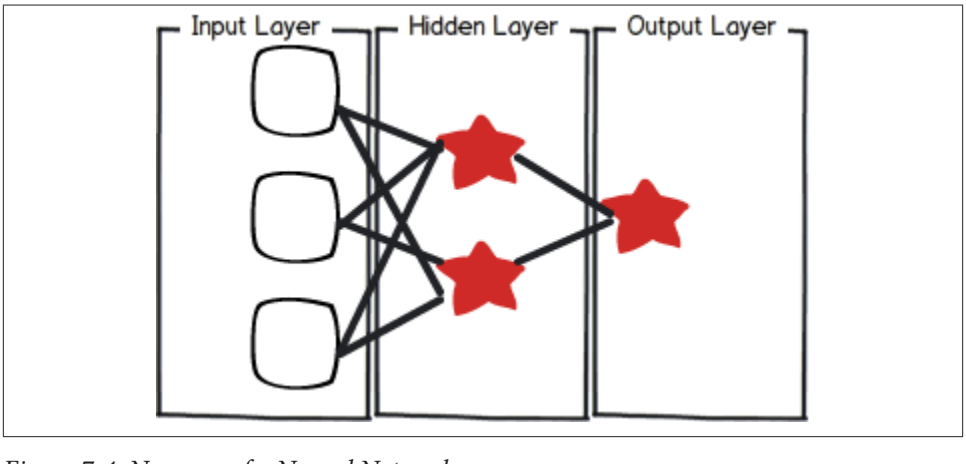
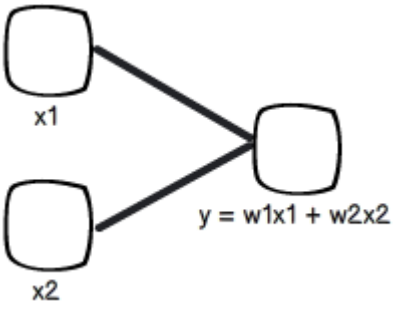


Figure 7-4. Neurons of a Neural Network

Neurons

Neurons are weighted linear combinations that are wrapped in an activation function. The weighted linear combination (or sum) is a way of aggregating all of the previous

neurons data into one output for the next layer to consume as input. Activation functions serve as a way to normalize data to be either symmetric or standard.



A simple neuron.

As a network is feeding information forward, it is aggregating previous inputs into weighted sums. At this point we take the value y and compute the activated value based off of an activation function.

Activation Functions

Activation functions serve as a way to normalize data between either the standard or symmetric ranges. They also are differentiable. The reason activation functions need to be differentiable is because of how we find weights in a training algorithm.

The big advantage of using an activation function is they serve as a way of buffering incoming values at each layer. This is useful because neural networks have a way of finding patterns and forgetting about the noise.

There are two main categories for activation functions: sloped or periodic. The periodic functions are used for data that is random and is the cosine and sine. In most cases the sloped activation functions are a suitable default choice.

Table 7-2. Some common activation functions are (but not limited to)

Name	Standard	Symmetric
Sigmoid	$\frac{1}{1 + e^{-2sum}}$	$\frac{2}{1 + e^{-2sum}} - 1$
Cosine	$\frac{\cos(sum)}{2} + 0.5$	$\cos(sum)$
Sine	$\frac{\sin(sum)}{2} + 0.5$	$\sin(sum)$
Gaussian	$\frac{1}{e^{sum^2}}$	$\frac{2}{e^{sum^2}} - 1$
Elliott	$\frac{0.5sum}{1 + \lvert sum \rvert} + 0.5$	$\frac{sum}{1 + \lvert sum \rvert}$
Linear	sum > 1 ? 1 : (sum < 0 : sum)	sum > 1 ? 1 : (sum < -1 ? -1 : sum)
Threshold	sum < 0 ? 0 : 1	sum < 0 ? -1 : 1

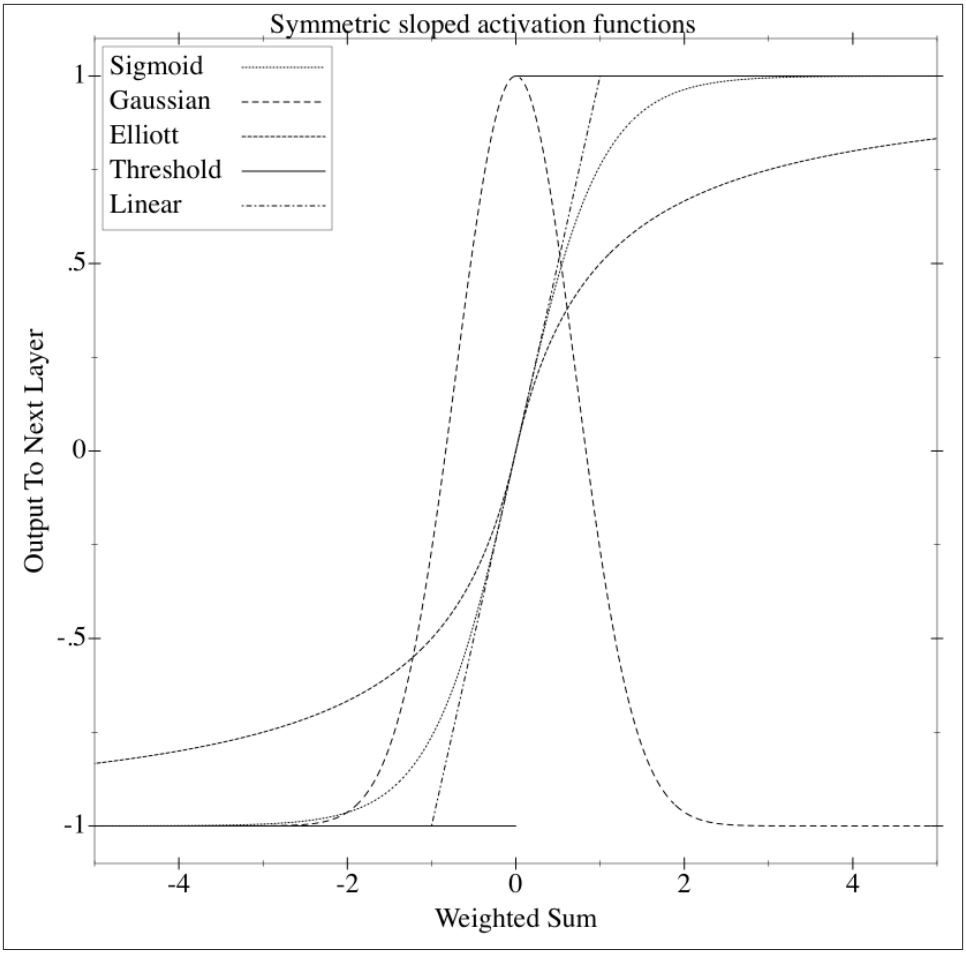


Figure 7-5. Symmetric sloped Activation functions

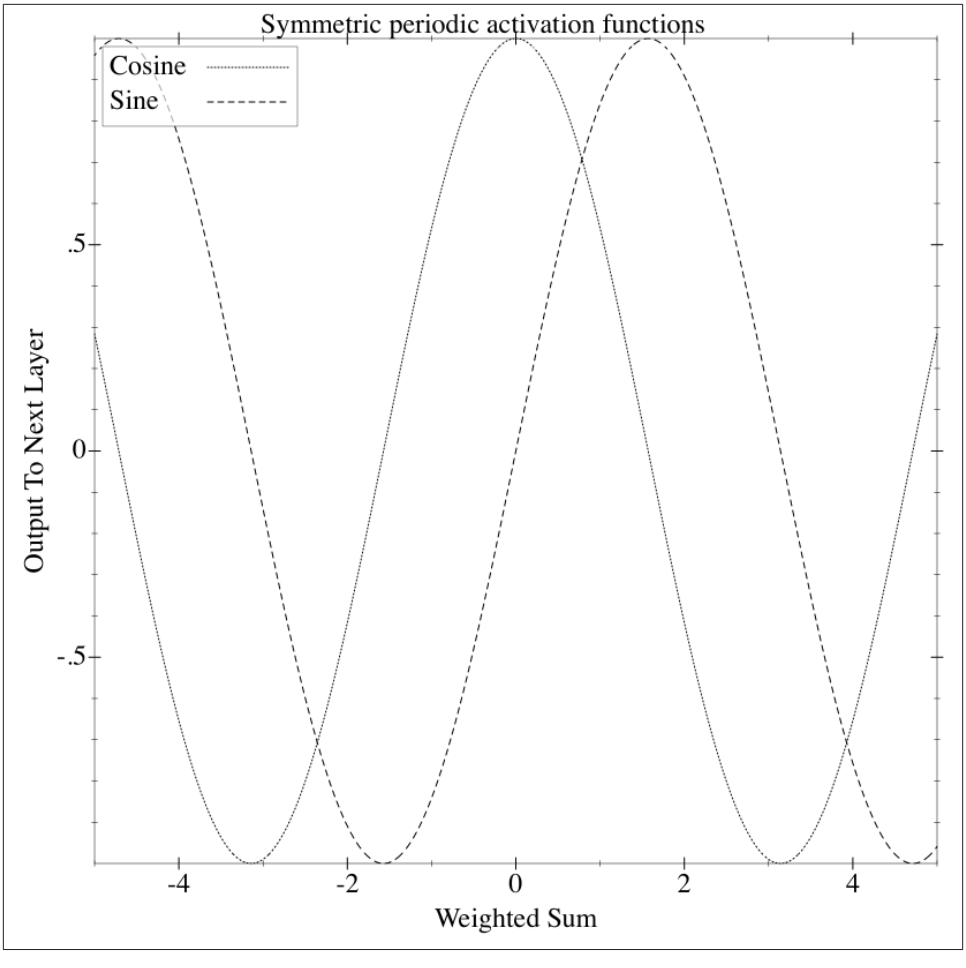


Figure 7-6. Symmetric periodic activation functions

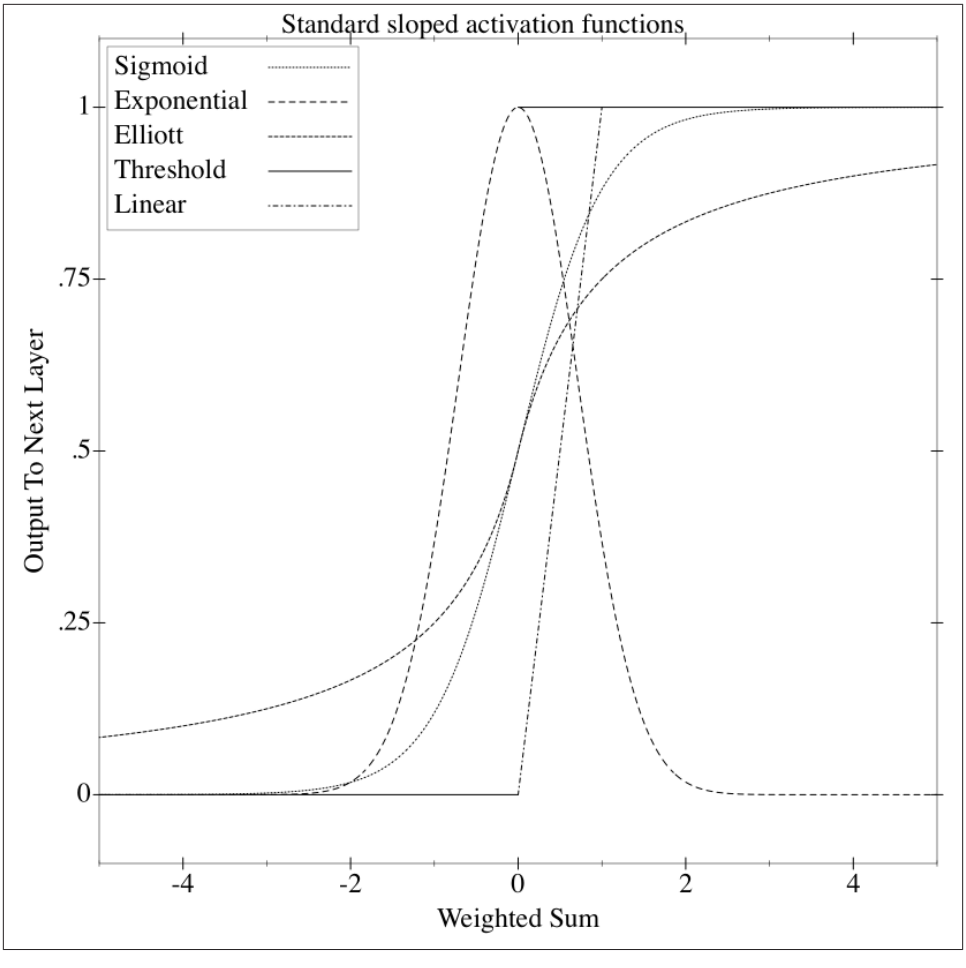


Figure 7-7. Standard Sloped Activation Functions

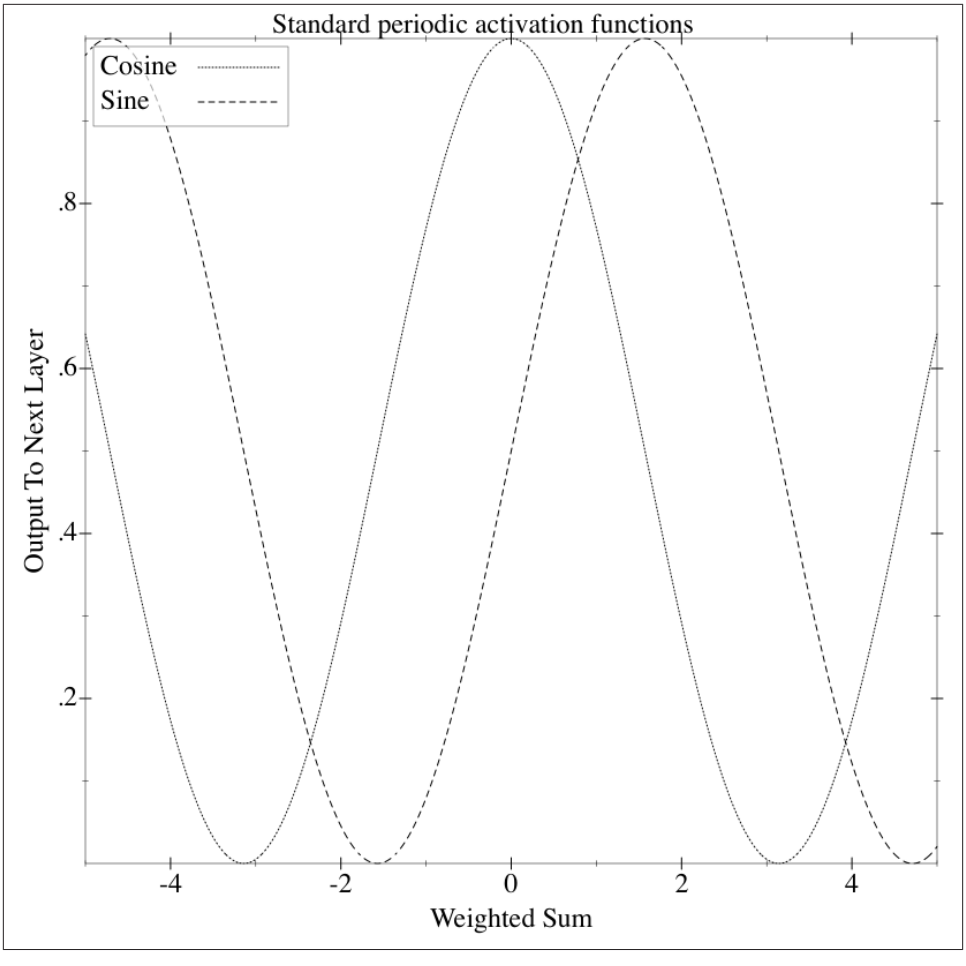
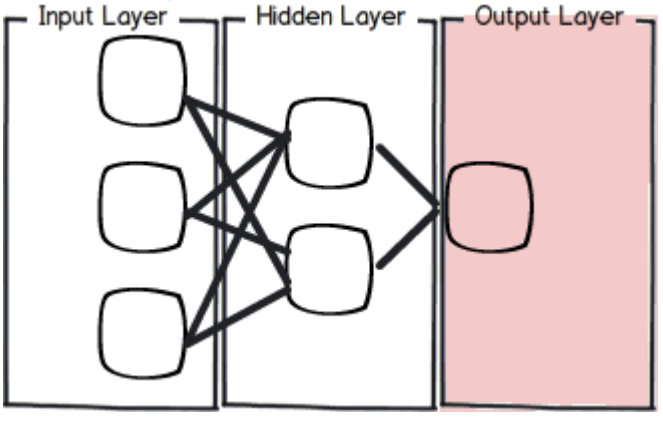


Figure 7-8. Standard Periodic Activation Functions

Sigmoid is the default function to be used with neurons because of its ability to smooth out the decision. Elliott is a sigmoidal function that is quicker to compute so the choice I make. Cosine and sine waves are used when you are mapping something that has a random looking process associated with it. In most cases these trigonometric functions aren't as useful to our problems.

Neurons are where all of the work is done. They are a weighted sum of previous inputs put through an activation function that either bounds it to 0 to 1 or -1 to 1. In the case of a neuron where we have two inputs before it the function for the neuron would be $y = \phi(w_1\dot{x}_1 + w_2\dot{x}_2)$ where ϕ is an activation function like sigmoid, and w_i is weights determined by a training algorithm.

Output layer of a Neural Network.



Output Layer

The output layer is similar to input layer except that it has neurons in it. This is where the data comes out of the model. Just like the input layer this will either be symmetric or standard data.

Output layers have the decision of choosing how many neutrons are in it, which is a function of what we're modeling. In the case of a function that outputs whether a stop light is red green or yellow, we'd have three outputs one for each. Each one of those outputs would contain an approximation for what we want.

Training Algorithm

The weights for each neuron I said came from a training algorithm. There are many algorithms but the most common are:

- Back propagation
- QuickProp
- RProp

All of these algorithms take the approach of finding optimal weights for each neuron. This is done through iterations also called epochs. For each epoch, a training algorithm goes through the entire neural net and compares it against what is expected. At this point it learns from past miscalculations.

These algorithms have one thing in common they are trying to find the optimal solution in a convex error surface. Convex error surface could also be thought of as a bowl with a minimum in it. Imagine that you are at the top of a hill and want to make it to a valley, but the valley is full of trees. You can't see much in front of you but you know that you

want to hit the bottom valley. The way you would do this is by proceeding based off of local inputs and make a guess where you want to go next. The training algorithms do the same thing; they are looking to minimize error by using local information.

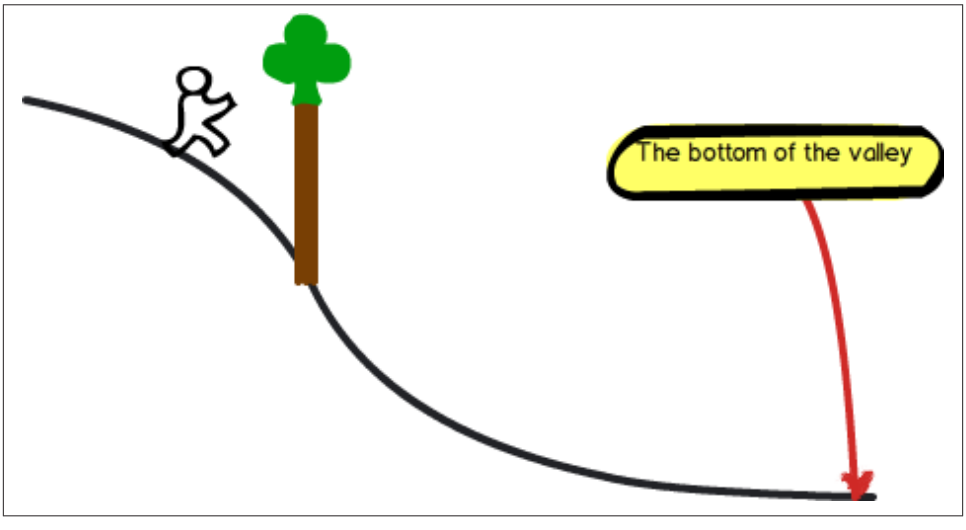


Figure 7-9. Gradient descent algorithm in a nutshell

The Delta Rule

While we could solve a massive amount of equations it would be faster to iterate. Instead of trying to calculate the derivative of the error function with respect to the weight what we calculate instead of a change in weight for each neuron's weights.

The delta rule is as follows:

$$\delta w_{ji} = \alpha(t_j - \phi(h_j))\phi'(h_j)x_i$$

This is stating that the change in weight for the neuron j's weight number i is

```
alpha * (expected - calculated) * derivative_of_calculated * input_at_i
```

alpha is called the learning rate and is a small constant. This is derived in an appendix if you want to figure out how this was calculated. This initial idea though is what powers the idea behind back propagation, or the general case of the delta rule.

Back propagation

Back propagation is the simplest of the three algorithms. If you define error as (expected - actual) ^ 2 where expected is the expected output and actual is the calculated number from the neurons. At this point we want to find where the derivative of that equals zero which is the minimum.

$$\delta w(t) = -\alpha(t - y)\phi' x_i + \epsilon \delta w(t - 1)$$

ϵ is called the momentum factor which propels previous weight changes into our current weight change α is called the learning rate

Back propagation has the problem of taking many epochs to calculate. Up until 1988 researchers were struggling to train simple neural networks. There was research on how to improve this which lead to a whole new algorithm called QuickProp.

QuickProp

Scott Fahlman introduced an algorithm called QuickProp after he studied how to make back propagation better. He asserted that back propagation took too long to converge to a solution. Instead he proposed that we take the biggest steps without overstepping the solution.

What he described is that there are two ways to make back propagation better: making momentum and learning rate dynamic, and to make use of a second derivative of the error with respect to each weight. In the first case, one could better optimize for each weight, and with second derivatives you could utilize Newton's method of approximating functions.

With QuickProp the main difference is that you keep a copy of the error derivative computed during the previous epoch, along with the difference between the current and previous values of this weight.

To calculate a weight change at time t the following function is used

$$\delta w(t) = \frac{S(t)}{S(t - 1) - S(t)} \delta w(t - 1)$$

This has the risk of changing the weights too much so there is a new parameter for maximum growth. No weight is allowed to be greater in magnitude than the maximum growth rate times the previous step for that weight.

RProp

RProp is the most used algorithm because it converges fast. It was introduced by Martin Riedmiler in the 1990s and has had some improvements since then. The reason for its speed is because it converges on a solution fast due to its insight that the algorithm can update the weights many times through an epoch. Instead of calculating weight changes based on a formula it uses only the sign for change and uses a increase factor and decrease factor.

Going through what this algorithm looks like in code we need to define a few constants (or defaults). These are a way of making sure the algorithm doesn't operate forever or become volatile. These defaults were taken from the FANN library.

The basic algorithm was easier to explain in Ruby instead of writing out the partial derivatives.

For ease of reading note that I am not calculating the error gradients which are the change in error with respect to that specific weight term:

```
neurons = 3
inputs = 4

delta_zero = 0.1
increase_factor = 1.2
decrease_factor = 0.5
delta_max = 50.0
delta_min = 0
max_epoch = 100
deltas = Array.new(inputs) { Array.new(neurons) { delta_zero }}
last_gradient = Array.new(inputs) { Array.new(neurons) { 0.0 } }


sign = ->(x) {
  if x > 0
    1
  elsif x < 0
    -1
  else
    0
  end
}

weights = inputs.times.map {|i| rand(-1.0..1.0) }

1.upto(max_epoch) do |j|
  weights.each_with_index do |i, weight|
    # Current gradient is derrived from the change of each value at each layer
    gradient_momentum = last_gradient[i][j] * current_gradient[i][j]

    if gradient_momentum > 0
      deltas[i][j] = [deltas[i][j] * increase_factor, delta_max].min
      change_weight = -sign.(current_gradient[i][j]) * deltas[i][j]
      weights[i] = weight + change_weight
      last_gradient[i][j] = current_gradient[i][j]
    elsif gradient_momentum < 0
      deltas[i][j] = [deltas[i][j] * decrease_factor, delta_min].max
      last_gradient[i][j] = 0
    else
      change_weight = -sign.(current_gradient[i][j]) * deltas[i][j]
      weights[i] = weights[i] + change_weight
      last_gradient[i][j] = current_gradient[i][j]
```

```
      end
    end
  end
```

An understanding is needed to be able to build a neural network. Next we'll talk about how to build a neural network and what decisions we should make to build an effective network.

How to go about building neural networks

There are a few things that you have to ask before you go about building a neural net

1. How many hidden layers to use
2. How many neurons per layer
3. What is your tolerance for error?

How many hidden layers?

The thing that makes neural nets unique is their usage of hidden layers. If you took out hidden layers, you'd have a linear combination problem. You aren't bound to use any number of hidden layers but there are three heuristics that help.

Do not use more than 2 hidden layers, otherwise you might overfit the data. With too many layers the network starts to memorize the training data. Instead we want it to find patterns.

One hidden layer will do the job of approximating a continuous mapping. This is the common case. Most neural networks only have one hidden layer in them.

Two hidden layers will be able to push past a continuous mapping. This is an uncommon case but if you know that you don't have a continuous mapping two hidden layers can be used.

There is no steadfast rule holding you to these heuristics for picking the amount of hidden layers. It comes down to trying to minimize the risk of overfitting or underfitting your data.

How many neurons for each layer?

Neural nets are excellent aggregators and terrible expanders. Since neurons themselves are weighted sums of previous neurons they have a tendency of not being able to expand out as well as they combine. If you think about it a hidden layer of 2 that goes to an output layer of 30 would mean that for each output neuron there would be two inputs. There just isn't enough entropy or data to make a well fitted model.

This idea of emphasizing aggregation over expansion flows into the next set of heuristics

- The amount of hidden neurons should be between the amount of the inputs and the amount of neurons at the output layer.
- The number of hidden neurons should be ⅔ the size of the input layer, plus the size of the output layer.
- The number of hidden neurons should be less than twice the size of the input layer.

This comes down to trial and error though since the amount of hidden neurons will influence how well the model cross validates, as well as the convergence on a solution. This is just a starting point.

Tolerance for Error and Max epochs

The tolerance for error gives us a time to stop training. We will never get to a perfect solution but instead converge on one. If you want an algorithm that performs well, then the error rate might be low like 0.01%. But in most cases that will take a long time to train due to its intolerance for error.

Many start with an error tolerance of 1%. Through cross validation, this might need to be tightened even more. In neural network parlance the tolerance is internal and measured as a mean squared error and defines a stopping place for the network.

Neural networks are trained over epochs and this is set before the training algorithm even starts. If an algorithm is taking 10,000 iterations to get to a solution then there might be a high risk for over training and have a sensitive network. A starting point for training is 1000 epochs or iterations to train over. This way the model can model some complexity without getting too carried away.

Both max epochs and maximum error defines our converging points. This will serve as a way to signal when the training algorithm can stop and yield the neural network. At this point we've learned enough to get our hands dirty and try a real world example.

Code Example: Using a Neural Network to classify a Language

Characters used in a language have a direct correlation with the language itself. Mandarin is recognizable due to its characters, since each character means a specific word. The same is true with many Latin based languages as well but instead on letter frequency.

If we look at the difference of "The quick brown fox jumped over the lazy dog." in English and the German equivalent "Der schnelle braune Fuchs sprang über den faulen Hund." we'd be left with the following frequency chart:

Table 7-3. Difference in frequency between English and German Sentence

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	ü
English	1	1	1	2	4	1	1	2	1	1	1	1	1	1	4	1	1	2	0	2	2	1	1	1	1	1	0
German	3	2	2	3	7	2	1	3	0	0	0	3	0	6	0	1	0	4	2	0	4	0	0	0	0	1	1
Difference	2	1	1	1	3	1	0	1	1	1	1	2	1	5	4	0	1	2	2	2	2	1	1	1	1	0	1

There is a subtle difference between German and English. German uses quite a bit more N's while English uses a lot of O's. If we wanted to expand this to a few more European languages how would we do that? More specific how can we built a model to classify sentences written in English, Polish, German, Finnish, Swedish, or Norwegian?

In this case we'll build a simple model to predict a language on the frequency of the characters in the sentences. But before we start we need to have some data. For that I am using the most translated book in the world, the Bible. I'm extracting all the chapters out of Matthew and Acts.

The approach we will take is to extract all the sentences out of these text files that I have downloaded and create vectors of frequency normalized between 0 and 1. From that we will train a network which will take those inputs and then match them to a vector of 6. The vector of 6 is defined as the index of the language equaling 1. If our language we are using to train is index 3 the vector would look like [0,0,0,1,0,0] (zero based indexing).

Setup Notes

All of the code we're using for this example can be found on GitHub at *https://github.com/thoughtfulml/examples/tree/master/6-neural-networks*.

Ruby is constantly changing so the README is the best place to come up to speed on running the examples.

Grabbing the data

If you want to grab the data I wrote the following script to download it from biblegateway.com.

```
require 'nokogiri'
require 'open-uri'

url = "http://www.biblegateway.com/passage/"

languages = {
  'English' => {
    'version' => 'ESV',
    'search' => ['Matthew', 'Acts']
  },
```

```
    'Polish' => {
      'version' => 'SZ-PL',
      'search' => ['Ewangelia+według+św.+Mateusza', 'Dzieje+Apostolskie']
    },
    'German' => {
      'version' => 'HOF',
      'search' => ['Matthaeus', 'Apostelgeschichte']
    },
    'Finnish' => {
      'version' => 'R1933',
      'search' => ['Matteuksen', 'Teot']
    },
    'Swedish' => {
      'version' => 'SVL',
      'search' => ['Matteus', 'Apostlagärningarna']
    },
    'Norwegian' => {
      'version' => 'DNB1930',
      'search' => ['Matteus', 'Apostlenes-gjerninge']
    }
  }

  languages.each do |language, search_pattern|
    text = ''

    search_pattern['search'].each_with_index do |search, i|
      1.upto(28).each do |page|
        puts "Querying #{language} #{search} chapter #{page}"
        uri = [
          url,
          URI.encode_www_form({
            search: "#{URI.escape(search)}+#{page}",
            version: "#{search_pattern.fetch('version')}"
          })
        ].join('?')
        puts uri
        doc = Nokogiri::HTML.parse(open(uri))
        doc.css('.passage p').each do |verse|
          text += verse.inner_text.downcase.gsub(/[\d,;:\\\-\"]/,'')
        end
      end
      File.open("#{language}_#{i}.txt", 'wb') {|f| f.write(text)}
    end
  end
```

This will download and store Acts and Matthew verses using the multiple languages. You are free to try more languages!

Writing the seam test for Language

To take our training data we need to build a class to parse that and interface with our neural network. For that we will use the class name Language. The Language's purpose is to take a file of text in a given language and load it into a distribution of character frequencies. When needed the Language will output a vector of these characters all summing up to 1. All of our inputs will be between 0 and 1.

We are concerned with a few things on how Language works

1. We want to make sure that our data is correct and sums up to 1.
2. We don't want characters like utf-8 spaces or punctuation entering our data.
3. We want to downcase all characters. "A" should be translated as "a". "Ä" should also be "ä".

This helps us to make sure that our Language class which takes a text file and outputs an array of hashes is correct.

```
# encoding: utf-8
# test/lib/language_spec.rb

require 'spec_helper'
require 'stringio'


describe Language do
  let(:language_data) {
    <<-EOL
    abcdefghijklmnopqrstuvwxyz.
    ABCDEFGHIJKLMNOPQRSTUVWXYZ.
    \u00A0.
    !~@#$\%^&*()_\+'?[]“”‘’—<>»«›‹-„/.
    ïëéüòèöÄÖßÜøæåÅØóąłżŻśęńŚćźŁ.
    EOL
  }

  let(:special_characters) { language_data.split("\n").last.strip }

  let(:language_io) { StringIO.new(language_data) }

  let(:language) { Language.new(language_io, 'English') }

  it 'has the proper keys for each vector' do
    language.vectors.first.keys.must_equal ('a'..'z').to_a
    language.vectors[1].keys.must_equal ('a'..'z').to_a

    special_chars = "ïëéüòèöäößüøæååØóąłżŻśęńśćź".split(//).uniq.sort

    language.vectors.last.keys.sort.must_equal special_chars
  end
```

```
  it 'sums to 1 for all vectors' do
    language.vectors.each do |vector|
      vector.values.inject(&:+).must_equal 1
    end
  end

  it 'returns characters that is a unique set of characters used' do
    chars = ('a'..'z').to_a
    chars.concat "ïëéüòèöäößüøæååØóąłżźśęńśćź".split(//).uniq

    language.characters.to_a.sort.must_equal chars.sort
  end
end
```

At this point we have not written Language and all of our tests fail. For the first goal, let's get something that counts all the alpha characters and stops on a sentence. For that it'd look like this:

```
# encoding: utf-8

# lib/language.rb

class Language
  attr_reader :name, :characters, :vectors
  def initialize(language_io, name)
    @name = name
    @vectors, @characters = Tokenizer.tokenize(language_io)
  end
end

# lib/tokenizer.rb

module Tokenizer
  extend self
  PUNCTUATION = %w[~ @ # $ % ^ & * ( ) _ + ' [ ] “ ” ‘ ’ – < > » « › ‹ - „ /]
  SPACES = [" ", "\u00A0", "\n"]
  STOP_CHARACTERS = ['.', '?', '!']

  def tokenize(blob)
    unless blob.respond_to?(:each_char)
      raise 'Must implement each_char on blob'
    end

    vectors = []
    dist = Hash.new(0)

    characters = Set.new
    blob.each_char do |char|
      if STOP_CHARACTERS.include?(char)
        vectors << normalize(dist) unless dist.empty?
        dist = Hash.new(0)
```

```
        elsif SPACES.include?(char) || PUNCTUATION.include?(char)

        else
          character = char.downcase.tr("ÅÄÜÖËÏŚŻŁ", "ääüöëiśżł")
          characters << character
          dist[character] += 1
        end
      end
      vectors << normalize(dist) unless dist.empty?

      [vectors, characters]
    end
  end
```

At this point we have something that should work. Notice a few things interesting about this though which is that special characters like Ä do not get downcased to ä. For that you have to use String#tr. Also another peculiar problem is that there is a unicode space which is denoted as `\u00a0`.

Now we have a new problem though which is that data does not add up to 1. We will introduce a new function normalize which takes a hash of values and applies the function x / sum(x) to all values. Note that I use Rational which is a 1.9.x feature. This is because it increases the reliability of calculations and doesn't do floating point arithmetic until needed.

```
# lib/tokenizer.rb

module Tokenizer
  # tokenize

  def normalize(hash)
    sum = hash.values.inject(&:+)
    Hash[
      hash.map do |k,v|
        [k, Rational(v,  sum)]
      end
    ]
  end
end
```

Everything is green and things look great for Language. We have full test coverage on a class that will be used to interface with our Network. Now we can move onto building a Network class.

Cross validating our way to a Network class

I used the Bible to find training data for our language classification. It is the most translated book in our history, all religion and politics aside. For the data I decided to download all the chapters to Matthew and Acts in English, Finnish, German, Norwegian, Polish, and Swedish. With this natural divide between Acts and Matthew we can define

twelve tests of testing a model trained with Acts and how it compares to Matthew data vice versa.

The code looks like:

```
# test/cross_validation_spec.rb
# encoding: utf-8

require 'spec_helper'

# This is important since training neural networks every time a test is run
# can be a bit slow. This hash caches the networks.

networks = {}

describe Network do
  def language_name(text_file)
    File.basename(text_file, '.txt').split('_').first
  end

  def compare(network, text_file)
    misses = 0.0
    hits = 0.0

    file = File.read(text_file)

    file.split(/[\.!\?]/).each do |sentence|
      sentence_name = network.run(sentence).name

      if sentence_name == language_name(text_file)
        hits += 1
      else
        misses += 1
      end
    end

    total = misses + hits

    assert(
      misses < (0.05 * total),
      "#{text_file} has failed with a miss rate of #{misses / total}"
    )
  end

  def load_glob(glob)
    Dir[File.expand_path(glob, __FILE__)].map do |m|
      Language.new(File.open(m, 'r+'), language_name(m))
    end
  end

  let(:matthew_languages) { load_glob('../../data/*_0.txt') }
  let(:acts_languages) { load_glob('../../data/*_1.txt') }
```

```
  let(:matthew_verses) {
    networks[:matthew] ||= Network.new(matthew_languages).train!
    networks[:matthew]
  }

  let(:acts_verses) {
    networks[:acts] ||= Network.new(acts_languages).train!
    networks[:acts]
  }

  %w[English Finnish German Norwegian Polish Swedish].each do |lang|
    it "Trains and cross-validates with an error of 5% for #{lang}" do
      compare(matthew_verses, "./data/#{lang}_1.txt")
      compare(acts_verses, "./data/#{lang}_0.txt")
    end
  end
end
```

There is a folder called data in the root of the project that contains files in the form of of English_0.txt and English_1.txt where English is the language name and _1 is the index. The indexes map to matthew or acts: 1 maps to Acts verses and 0 maps to Matthew verses.

It takes a while to train a neural network so I'm only training two networks, one for Acts chapters and one for Matthew chapters. At this point we have 12 tests defined which I explained above. Of course nothing will work now because we haven't written the Network class. To start out the Network class we define an initial class as taking an array of Language classes. Secondly, since we don't want to write all the neural network by hand we're using a library called Ruby-Fann which interfaces with FANN. Our main goal to begin with is to get a neural network to accept and train.

The file now looks like this.

```
# lib/network.rb

require 'ruby-fann'

class Network
  def initialize(languages, error = 0.005)
    @languages = languages
    @inputs = @languages.map {|l| l.characters.to_a }.flatten.uniq.sort
    @fann = :not_ran
    @trainer = :not_trained
    @error = error
  end

  def train!
```

```
    build_trainer!
    build_standard_fann!
    @fann.train_on_data(@trainer, 1000, 10, @error)
    self
  end

  def code(vector)
    return [] if vector.nil?
    @inputs.map do |i|
      vector.fetch(i, 0.0)
    end
  end

  private
  def build_trainer!
    payload = {
      :inputs => [],
      :desired_outputs => []
    }

    @languages.each_with_index do |language, index|
      inputs = []
      desired_outputs = [0] * @languages.length
      desired_outputs[index] = 1

      language.vectors.each do |vector|
        inputs << code(vector)
      end

      payload[:inputs].concat(inputs)

      language.vectors.length.times do
        payload[:desired_outputs] << desired_outputs
      end
    end

    @trainer = RubyFann::TrainData.new(payload)
  end

  def build_standard_fann!
    hidden_neurons = (2 * (@inputs.length + @languages.length)) / 3

    @fann = RubyFann::Standard.new(
      :num_inputs => @inputs.length,
      :hidden_neurons => [ hidden_neurons ],
      :num_outputs => @languages.length
    )

    # Note that the library misspells Elliott
    @fann.set_activation_function_hidden(:elliot)
  end
end
```

Note now that we have the proper inputs and the proper outputs the model is set up and we should be able to run the whole cross_validation_test.rb. But of course there is an error because we cannot run new data against the network. For this we need to build a function called #run. At this point we have something that works and looks like this:

```
# lib/network.rb

require 'ruby-fann'
class Network
  # initialize
  # train!
  # code

  def run(sentence)
    if @trainer == :not_trained || @fann == :not_ran
      raise 'Must train first call method train!'
    else
      vectors, characters = Tokenizer.tokenize(sentence)
      output_vector = @fann.run(code(vectors.first))
      @languages[output_vector.index(output_vector.max)]
    end
  end
end
```

This is where things get interesting and we tune the model. Since it appears that German, English, Swedish and Norwegian are all failing our test. Since our code works now we are at the stage where we tune our neural network based on unit tests.

We have set our standards high and we can reach them through tuning the network.

Tuning the neural network

At this point I change the activation function to the Elliott function which improves the results by only having Norwegian, Swedish and German now. English dropped out of the errors but our epochs went up just a bit. Halving the internal error rate to 0.005 is my next step this is done by changing @fann.train_on_data to have 0.005 as the last argument. At this point things work and we've achieved our goal. We can stop here since we are satisfied.

I'll leave this up as an exercise in playing around with what works and what does not. You can try many different activation functions, as well as internal rates of decay or errors. The idea to realize is that with an initial test to base accuracy against you can try many different avenues.

Convergence testing

Before continuing on you can reset the max epochs in the network class to have 20-50% over what you saw just to make sure that over time things don't start taking forever. In

our case I saw around 200 epochs for the model to train so I'll reset the max epochs to 300.

Precision and Recall for Neural Networks

Taking this a step further when we deploy this neural network code to a production environment we need to close the information loop by introducing a precision and recall metric to track over time. This metric will be calculated off of user input.

This can be measured by asking the question in our user interface whether our prediction was correct or not. From this text, we can capture the blurb and what is the correct classification and feed that back into our model next time we train.

To learn more about monitoring precision and recall read forward to Chapter 9.

What is needed to monitor the performance of this neural network in production is a metric for how many times a classification was run, as well as how many times it was wrong.

Wrapup of example

Neural networks are a fun way of mapping information and learning through iterations and it works well for our case of mapping sentences to languages. Loading this up in an irb session I had fun trying out phrases like "meep moop" which is classified as Norwegian. Feel free to play with the code. It is all here on bitbucket [*https://bitbucket.org/hexgnu/neural-network-language-predictor* |Note I intend on showing them a git repo to play around with].

Conclusion

Neural Networks are a powerful tool in a machine learning toolkit. They serve as a way of mapping previous observations through a functional model. While they are touted as black box models, they can be understood with a little bit of mathematics and illustration. You can use neural networks for many things like mapping letter frequencies to languages or handwriting detection. There are many problems being worked on right now with regards to this and more in depth books about it as well.

This chapter we introduced Neural Networks as an Artificial version of our brain and explained how they work by summing up inputs using a weighted function. These weighted functions are then normalized between a certain range. Many algorithms exist to train these weight values but the most prevalent is the RProp algorithm. Then we summed it all up with a practical example on mapping sentences to languages.

CHAPTER 8
Clustering

If you've been to a library you've seen clustering at work. The Dewey Decimal system is a form of clustering. Dewey came up with a system that would attach a number of increasing granularity and revolutionized libraries.

This idea of categorizing datapoints, or books, into groups is useful for organizing informatin. Since we don't know what category they should belong to we just want to split the books into a set of categories.

This type of problem is much different than what we've encountered before. All of the problems we have looked at thus far have been attempting to figure out the best functional approximation to assign to a given data set and it's labels. Now we are more concered with the data itself not the labels.

As you will see during this chapter clustering has one bad side which is that it doesn't abide to use using it as well as other algoritms. This is called the **impossibility theorem**.

In this chapter we will talk about clustering in general as it applies to cohorts of users, and then introduce K-Means Clustering and then EM Clustering. Finally we'll finish with an example about clustering i jazz records into groups based on styles.

Clustering in Brief

Clustering is an unsupervised learning problem and is great for grouping data. They can have issues though with application to problems since they suffer from the impossibility theorem.

User Cohorts

Grouping people into cohorts makes a lot of business and marketing sense. For instance your first customer is different from your ten thousandth customer or millionth customer. This problem of defining users into *cohorts* is a common one. If we were able to effectively split our customers into different buckets based on behavior and time of signup then we could better serve them by diversifying our marketing strategy.

The problem is that we don't have a pre-defined label for customer cohorts. To get over this problem you could look at what month and year they became a customer. But that is making a big assumption about that being the defining factor that splits customers into groups. What if time of first purchase had nothing to do with whether they were in one cohort or the other? For example they could only have made one purchase or have made many.

Instead we should group users into cohorts, or clusters based on what we do know about our users. For instance let's say that we know when they signed up, how much they buy and what their favorite color is. Over the last two years we've only had 10 users sign up (well I hope you have more than that over two years but let's keep this simple).

Table 8-1. Data Collected over 2 Years

User Id	Signup Date	Money Spent	Favorite Color
1	Jan 14	40	N/A
2	Nov 3	50	Orange
3	Jan 30	53	Green
4	Oct 3	100	Magenta
5	Feb 1	0	Cyan
6	Dec 31	0	Purple
7	Sep 3	0	Mauve
8	Dec 31	0	Yellow
9	Jan 13	14	Blue
10	Jan 1	50	Beige

Given these data we want to define a mapping from each user to a cohort. Looking at these rows you notice that the favorite colors are irrelevant data. There is no information as to whether users should be in a cohort. That leaves us with *Money Spent* and *Signup Date*. There seems to be a group of users who spend money, and those that don't. In the signup date column you'll notice that there is a lot of users who signup around the beginning of the year and end of the previous as well as around September, October, or November.

We have a choice to make which is the number of clusters we want to make. Since our dataset is so small we'll just split it into two pieces. That means that we can split the cohorts into what looks something like this.

Table 8-2. Manual cohort assignment to the original data set

User Id	Signup Date (Days to Jan 1)	Money Spent	Cohort
1	Jan 14 (13)	40	1
2	Nov 3 (59)	50	2
3	Jan 30 (29)	53	1
4	Oct 3 (90)	100	2
5	Feb 1 (31)	0	1
6	Dec 31 (1)	0	1
7	Sep 3 (120)	0	2
8	Dec 31 (1)	0	1
9	Jan 13 (12)	14	1
10	Jan 1 (0)	50	1

We have divided the group into two groups where 7 are in group 1 which we could call the beginning of the year signups and group 2 is the other category.

What if we were to do this more algorithmically though what would we do? In the next sections we'll touch on what KMeans clustering is, as well as introduce EM Clustering in the theoretical sense. Finally we'll wrap up this chapter with an example of how to categorize a record collection onto a bookshelf.

K-Means Clustering ~~~~~~~

There are a lot of clustering algorithms like linkage clustering or **Diana** one of the most common is K-Means clustering. Using a pre-defined K which is the number of clusters that one wants to split the data into K-Means will find the most optimal centroids of clusters. The nice properties of K-Means clustering is that the clusters will be strict, spherical in nature and it converges to a solution.

In this section we will briefly talk about how K-Means clustering works.

The K-Means Algorithm

The K-Means algorithm starts with a base case. Pick K random points in the data set and define them to be centroids. At that point for each point assign them to a cluster nubmer that is closest to each different centroid. At this point we have a clustering based on the original randomized centroid. But that is not exactly what we want to end with so we update where the centroids are using a mean of the data. At that point we repeat until the centroids no longer move.

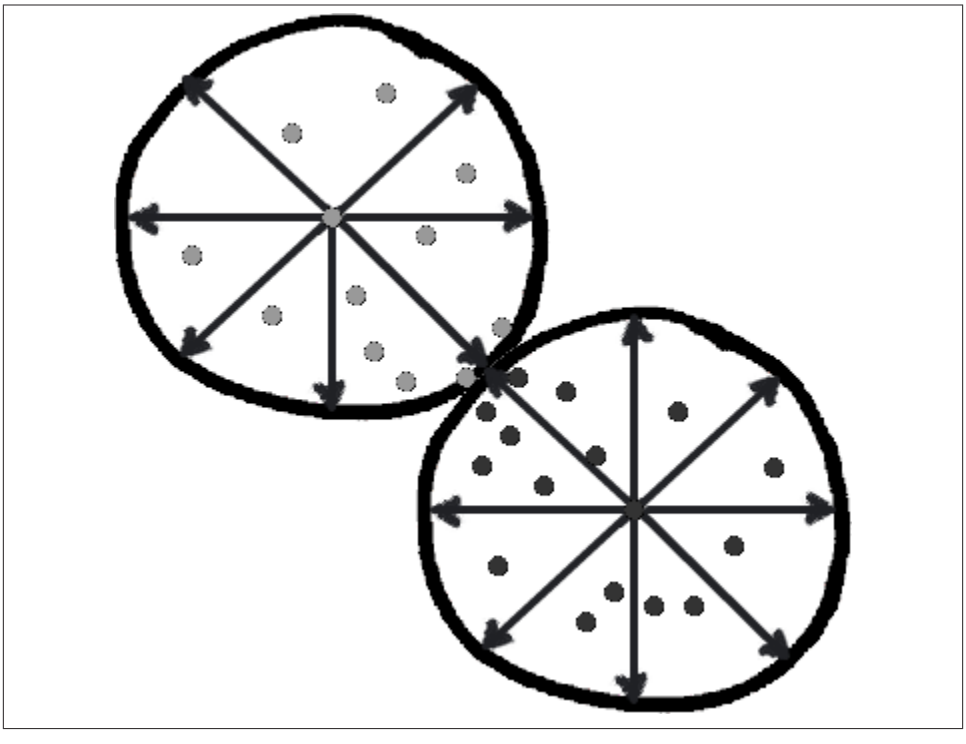


Figure 8-1. This shows how K-Means is very circular.

Note that there are a lot of the same considerations we have from before regarding distance metrics. For review those metrics we can use to build a K-Means algorithm are:

1. Manhattan Distance: $d_{manhattan}(x,\ y) = \sum_{i=1}^{n} \mid x_i - y_i \mid$
2. Euclidean Distance: $d_{euclid}(x,\ y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$
3. Minkowski distance: $d(x,\ y) = (\sum_{i=1}^{n} \left| x_i - y_i \right|^p)^{\frac{1}{p}}$
4. Mahalanobis Distance: $d(x,\ y) = \sqrt{\sum_{i=1}^{n} \frac{(x_i - y_i)^2}{s_i^2}}$

Downside of KMeans clustering

The downside of K-Means clustering is that everything must have a hard boundary. This means that a data point can only be in one cluster and not straddle the line between the two. On top of that K-Means prefers spherical data since most of the time the Euclidean distance is being used. It is obvious the downsides when looking at a graphic like this where the data in the middle could really go either direction to cluster 1 or 2.

Expectation Maximization (EM) Clustering

Expectation Maximization clustering instead of focusing on finding a centroid and then datapoints that relate to it focuses on solving a different problem. Let's say that you want to split your data points into two sections either cluster 1 or 2. You want a good guess whether the data is in either cluster but don't care if there's some fuzziness. Instead of getting an assignment we really want a probability that the data point is in each cluster.

Unlike KMeans clustering which focuses on making definite boundaries between clusters EM Clustering is robust to datapoints that might be in either cluster. This can be quite useful for classifying data that doesn't have a definite boundary.

So to do that we make a vector called $z_k = < 0.5, 0.5 >$ which holds the probability of whether the row vector k is in either cluster. Through iterations we find out something more like $z_k = < 0.9, 0.1 >$. Graphically imagine you have a set of data points and instead of circling different clusters like before we assign a shade to each one. Where the darker it is the more Cluster 2 it becomes while the lighter greys become Cluster 1.

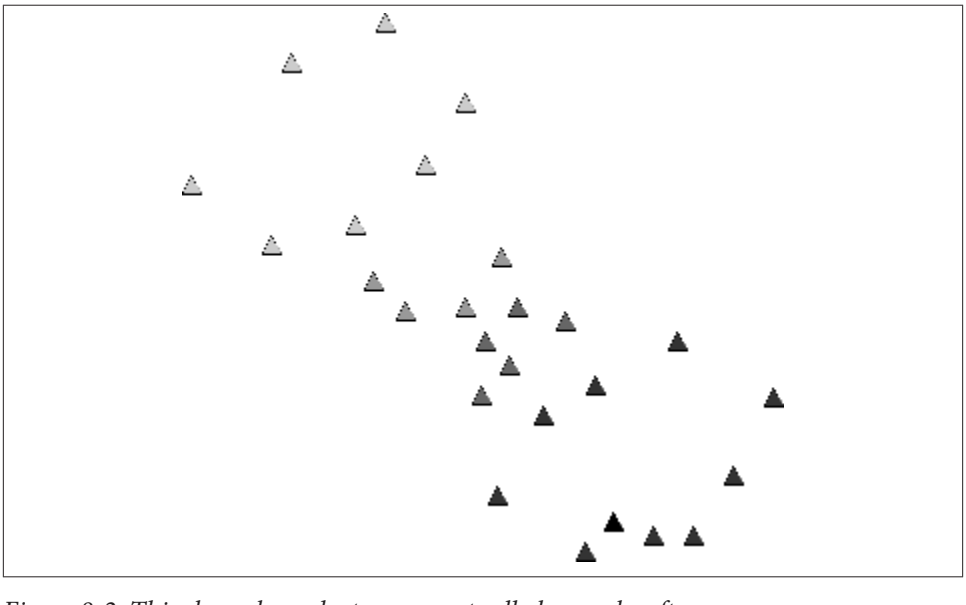


Figure 8-2. This shows how clusters can actually be much softer

Algorithm

The EM Algorithm is split into two pieces, the Expectation Step and the Maximization Step.

The expectation step is where the probabilities get calculated given what the data currently looks like and given what our initial z_k is. We calculate the log liklihood function with respect to the conditional distribution of Z given X under the current estimate of the parameters of Theta(t):

$Q(\theta \parallel \theta_t) = E_{Z \parallel X, \theta_t} logL\ (\theta; X, Z)]$

Next the maximization step is finding the parameter θ that maximizes the probability of θ given θ_t.

$\theta_t = \arg\max_\theta Q(\theta \parallel \theta_t)$

The unfortunate thing about EM clustering is that it does not converge nessecarily and can falter when mapping data with singular covariances. We will delve into more of the issues related with EM Clustering in the example section. First though we need to talk about one thing that all clustering algorithms share in common which is the **impossibility theorem**.

The Imposisbility Theorem

There is no such thing as free lunch and clustering is the same thing. The benefit we get out of clustering to magically map data points to particular groupings comes at a cost and this was laid out by Jon Kleinberg. He touts it as the **Impossibility Theorem**.

What this states is out of three attributes:

- Richness
- Scale invariance
- Consistency

You can never have more than two.

Richness is the notion that there exists a distance function that will yield all different types of partitions. What this means intuitively is that a clustering algorithm has the ability to create all types of mappings from data points to cluster assignments.

Scale invariance is simple to understand. Imagine that you were building a rocket ship and started calculating everything in kilometers until your boss said that you need to use miles instead. There wouldn't be a problem switching you just need to divide by a constant on all your measurements. It is scale invariant. Basically if the numbers are all multiplied by 20 then the same cluster should happen.

Consistency is more subtle. Similar to scale invariance, if we shrunk the distance between points inside of a cluster and expanded them then the cluster should yield the same result. At this point you probably understand that clustering isn't as good as many

originally think. It has a lot of issues and consistency is definitely one of those that should be called out.

For our purposes KMeans and EM Clustering satisfy richness and Scale Invariance but not Consistency. This fact makes testing clustering just about impossible. The only way we really can test is by anecdote and example. But that is ok for analysis purposes.

In the next section we will analyze jazz music using K-Means and EM Clustering.

Example: Categorizing Music

Music has a deep history of recordings, as well as composed pieces. You could have an entire degree and study musicology just to be able to effectively categorize these sheets of music.

The ways we can split music into categories is endless. Personally I split my own record collection by artist name. But artists will play with others, on top of that sometimes we can categorize based on genre. But what about the fact that genres are broad. Like jazz for instance. According to Montreux Jazz Festival Jazz is anything you can improv over.

How can we effectively build a library of music where we can split up our collection into similar pieces of work.

Instead let's approach this by using K-Means and EM Clustering. By the end of clustering the music we would have a soft clustering of music pieces that we could utilize to build a taxonomy of music.

In this section we will first determine where we will get our data from, what sort of attributes we can extract on as well as determine what we can validate based on. We will also discuss why clustering sounds great in theory but in practice doesn't give us much except for clusters.

Setup Notes

All of the code we're using for this example can be found on GitHub at *https://github.com/thoughtfulml/examples/tree/master/7-expectation-maximization-clustering*.

Ruby is constantly changing so the README is the best place to come up to speed on running the examples.

We will not be using a test driven approach to writing a clusterer. The reason behind that is because clustering is a problem that doesn't bode well for hypothesis testing. This is a very important thing to realize about clustering. On the surface it seems like a great solution to all problems but in reality it doesn't bode well for actually testing our assumptions.

When it comes to the impossibility theorem remember from before that we can't have a clustering algorithm that is consistent, rich and scale invariant. In many ways clustering is a data analysis tool but not something we should use to solve problems that we want to control.

Gathering the Data

There is a massive amount of data on music from the 1100s through today. We have MP3's, CD's, Vinyl Records, and written music. Without trying to classifying the entire world of music let's determine a small subsection that we can use. Since I don't want to engage in any copyright suits we will only use public information on albums. This would be Artist, Song Name, Genre (if available), as well as characteristics which we can find on the music. To achieve this we have access to a plethora of information contained in Discogs.com. They have made available lots of xml data dumps of records and songs.

Also since I we aren't trying to cluster the entire data set of albums in the world let's just focus on Jazz. Most people would agree that Jazz is a genre that is hard to really classify into any category. It could be fusion, it could be steel drums.

So to get our data set I downloaded the best jazz albums according to a website *http://www.scaruffi.com/jazz/best100.html*.

The data goes back to 1940 and well into the 2000's. In total I was able to download about 1200 unique records. All great albums!

But that isn't enough information. On top of that I annotated the information by using the Discogs API to determine the style of jazz music in each.

After annotating the original data set I found that there is 128 unique styles associated with Jazz (at least according to Discogs). They range from Aboriginal to Vocal.

Analyzing the data with K-Means

Like we did with K-Nearest Neighbors we need to figure out an optimal K. Unfortunately with clustering there really isn't much we can test with except to just see whether we split into two different clusters.

But let's say that we want to fit all of our records on a shelf and have 25 slots. Similar to the Ikea bookshelf. We could run a clustering of all of our data using K = 25.

Doing that would requires little code because we have the AI4R gem to rely on.

```
# lib/kmeans_clusterer.rb

require 'csv'
require 'ai4r'

data = []

artists = []
CSV.foreach('./annotated_jazz_albums.csv', :headers => true) do |row|
  @headers ||= row.headers[2..-1]
  artists << row['artist_album']
  data << row.to_h.values[2..-1].map(&:to_i)
end

ds = Ai4r::Data::DataSet.new(:data_items => data, :data_labels => @headers)
clusterer = Ai4r::Clusterers::KMeans.new
clusterer.build(ds, 25)

CSV.open('./clustered_kmeans.csv', 'wb') do |csv|
  csv << %w[artist_album year cluster]
  ds.data_items.each_with_index do |dd, i|
    csv << [artists[i], dd.first, clusterer.eval(dd)]
  end
end
```

That's it! Of course clustering without looking at what this actually tells us is useless. This does split the data into 25 different categories but what does it all mean?

Looking at a graphic of year versus assigned cluster number yields interesting results.

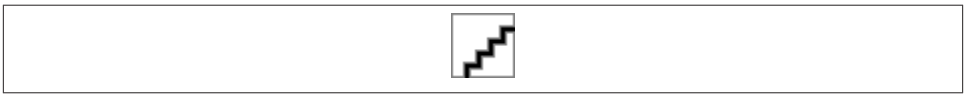


Figure 8-3. K Means Applied to Jazz Albums

As you can see Jazz starts out in the Big Band era pretty much in the same cluster, it transitions into the Cool Jazz and then around 1959 it starts to go everywhere until about 1990 when things cool down a bit.

What's fascinating is how well that syncs up with jazz history.

What happens when we cluster the data using EM Clustering?

EM Clustering

With EM Clustering remember that we are probalistically assigning to different clusters, it isn't 100% one or the other. This could be highly useful for our purposes since Jazz has so much cross over.

There are no ruby gems that have EM Clustering in them. So we'll have to write our own version of the tool.

Let's go through the process of building our own and then utilize it to map the same data that we have from our jazz data set. And see what happens.

Our first step is to be able to initialize the cluster. If you remember we need to have indicator variables z_i which follows a uniform distribution. These tell us the probability that each data point is in each cluster. to do this we have

```
# lib/em_clusterer.rb

require 'matrix'

class EMClusterer
  attr_reader :partitions, :data, :labels, :classes

  def initialize(k, data)
    @k = k
    @data = data
    setup_cluster!
  end

  def setup_cluster!
    @labels = Array.new(@data.row_size) { Array.new(@k) { 1.0 / @k }}

    @width = @data.column_size
    @s = 0.2

    pick_k_random_indices = @data.row_size.times.to_a.shuffle.sample(@k)

    @classes = @k.times.map do |cc|
      {
        :means => @data.row(pick_k_random_indices.shift),
        :covariance => @s * Matrix.identity(@width)
      }
    end
    @partitions = []
  end
end
```

At this point we have set up all of our base case stuff. We have @k which is the number of clusters, @data is the data we pass in we want to cluster, @labels are an array full of the probability that the row is in each cluster, @classes holds onto an array of means

and covariances which tells us where the distribution of data is. And finally @partitions are the assignments of each data row to cluster index.

Now we need to build our expectation step which is to figure out what the probability of each data row is in each cluster. To do this we need to write a new method `expect` which will do this.

```
# lib/em_clusterer.rb

class EMClusterer
  # initialize
  # setup_cluster!

  def expect
    @classes.each_with_index do |klass, i|
      puts "Expectation for class #{i}"

      inv_cov = if klass[:covariance].regular?
        klass[:covariance].inv
      else
        puts "Applying shrinkage"
        (klass[:covariance] - (0.0001 * Matrix.identity(@width))).inv
      end

      d = Math::sqrt(klass[:covariance].det)

      @data.row_vectors.each_with_index do |row, j|
        rel = row - klass[:means]

        p = d * Math::exp(-0.5 * fast_product(rel, inv_cov))
        @labels[j][i] = p
      end
    end

    @labels = @labels.map.each_with_index do |probabilities, i|
      sum = probabilities.inject(&:+)

      @partitions[i] = probabilities.index(probabilities.max)

      if sum.zero?
        probabilities.map { 1.0 / @k }
      else
        probabilities.map {|p| p / sum.to_f }
      end
    end
  end

  def fast_product(rel, inv_cov)
    sum = 0

    inv_cov.column_count.times do |j|
      local_sum = 0
```

```
        (0 ... rel.size).each do |k|
          local_sum += rel[k] * inv_cov[k, j]
        end
        sum += local_sum * rel[j]
      end

      sum
    end
  end
```

The first part of this is to iterate through all classes which holds onto the means and covariances of each cluster. From there we want to find the inverse covariance matrix as well as the determinant of the covariance. For each row we calculate a value that is proportional to the probability that the row is in a cluster. This is $p_{ij} = det(C)e^{-\frac{1}{2}(x_j-\mu_i)C^{-1}(x_j-\mu_i)}$. This is effectively a Gaussian distance metric to help us determine how far outside of the mean our data is.

Let's say that the row vector is exactly the mean. That would mean that this would reduce down to $p_{ij} = det(C)$ which is just the determinant of the covariance matrix. This is actually the highest value you can get out of this function. If for instance the row vector was far away from the mean vector then p_{ij} would become smaller and smaller due to the exponentiation and negative fraction in the front.

The nice thing is that this is proportional to the Gaussian probability the row vector is in the mean. Since this is proportional and not equal to that is why the last part we end up normalizing to sum up to 1.

You'll notice one last thing I did here which is to introduce a `fast_product` method. This is because the Matrix library in Ruby is slow and builds Array within Array which is memory inefficient. In this case things won't change that is why I made things optimized for that.

Now we can move onto the maximization step

```
# lib/em_clusterer.rb

class EMClusterer
  # initialize
  # setup_cluster!
  # expect
  # fast_product

  def maximize
    @classes.each_with_index do |klass, i|
      puts "Maximizing for class #{i}"
      sum = Array.new(@width) { 0 }
      num = 0

      @data.each_with_index do |row, j|
        p = @labels[j][i]
```

```
          @width.times do |k|
            sum[k] += p * @data[j,k]
          end

          num += p
        end

        mean = sum.map {|s| s / num }
        covariance = Matrix.zero(@width, @width)

        @data.row_vectors.each_with_index do |row, j|
          p = @labels[j][i]
          rel = row - Vector[*mean]
          covariance += Matrix.build(@width, @width) do |m,n|
            rel[m] * rel[n] * p
          end
        end

        covariance = (1.0 / num) * covariance

        @classes[i][:means] = Vector[*mean]
        @classes[i][:covariance] = covariance
      end
    end
  end
```

Again here we are iterating over the clusters called @classes. We first make an array called sum which holds onto the weighted sum of the data happening. From there we normalize to build a weighted mean. To calculate the covariance matrix we start with a zero matrix that is square and the width of our clusters. From there we iterate through all row_vectors and incrementally add on the weighted difference of the row and the mean for each combination of the matrix. Again at this point we normalize and store.

Now we can get down to using this. To do that we add two convenience methods which help in querying the data.

```
# lib/em_clusterer.rb

class EMClusterer
  # initialize
  # setup_cluster!
  # expect
  # fast_product
  # maximize

  def cluster(iterations = 5)
    iterations.times do |i|
      puts "Iteration #{i}"
      expect_maximize
    end
  end
```

```
  def expect_maximize
    expect
    maximize
  end
end
```

> **No Tests?**
>
> You'll notice that we haven't written any tests. Unfortunately with clustering there's not much you can test. Things are not deterministic, or really testable in any way. So I would suggest you spend some time testing manually. Probably the biggest downsides to using clustering is this.

The Results from the EM Jazz Clustering

Back to our results of EM Clustering with our jazz music. To actually run the analysis we run the following script.

```
data = []

artists = []

CSV.foreach('./annotated_jazz_albums.csv', :headers => true) do |row|
  @headers ||= row.headers[2..-1]
  artists << row['artist_album']
  data << row.to_h.values[2..-1].map(&:to_i)
end

data = Matrix[*data]


e = EMClusterer.new(25, data)
e.cluster
```

The first thing you'll notice about EM Clustering is that it's slow. It's not as quick as calculating new centroids and iterating. It has to calculate covariances, and means which are all inefficient. Ockham's Razor would tell us here that most likely EM Clustering is not a good use for clustering big amounts of data.

The other thing that you'll notice is that our annotated jazz music will not work and that is due to the fact that the covariance matrix of this is singular. This is not a good thing, matter of fact this problem is ill suited for EM Clustering due to this fact so we have to transform it into a different problem all together.

We do that by collapsing the dimensions into the top two genres by index.

```
require 'csv'

CSV.open('./less_covariance_jazz_albums.csv', 'wb') do |csv|
  csv << %w[artist_album key_index year].concat(2.times.map {|a| "Genre_#{a}" })

  CSV.foreach('./annotated_jazz_albums.csv', :headers => true) do |row|
    genre_count = 0
    genres = Array.new(2) { 0 }
    genre_idx = 0

    row.to_h.values[4..-1].each_with_index do |g, i|
      break if genre_idx == 2
      if g == '1'
        genres[genre_idx] = i
        genre_idx += 1
      end
    end

    next if genres.count {|a| a == 0 } == genres.length

    csv << [row['artist_album'], row['key_index'], row['year']].concat(genres)
  end
end
```

Basically what we are doing here is saying for the first two Genres let's assign a genre index to it and store it. Also the other problem is some albums just have zero information assigned to them so we skip those.

At this point we are able to run the EM Clustering algorithm except that things become too difficult to actually cluster. This is an important lesson with EM clustering. The data we have actually doesn't cluser because the matrix has become too instable to invert. I will leave this as an exercise to try out to see where it fails but the covariance matrix just becomes impossible to invert.

Conclusion

Clustering is useful but can be a bit hard to control since it is unsupervised. Add onto that we are dealing with the impossibility of having consistency, richness and scale-invariance all at once clustering can be a bit useless in many contexts. But don't let that get you down, clustering can be useful for analyzing datasets and splitting data into arbitrary clusters. If you don't care how they are split and just want them split up then clustering is good. Just know that there are sometimes odd circumstances.

CHAPTER 9

Kernel Ridge Regression

Regression is probably one of the most ubiquitous tools in any machine learning toolkit. The idea is simple: fit a line to some data that mapped from X to a Y. You have probably seen lots of regressions already. In a lot of ways it models the most common case and our naive base case. As you will see in this chapter linear regression is a good starting point for predicting data but breaks down quickly when trying to model data that has a small amount of data points, or isn't linear.

We will first introduce the problem of collaborative filtering and recommendation algorithms and then through the chapter improve how we approach the problem to finally settle on ridge regression. Finally at the end of the chapter we will code up our results and figure out whether our assumptions are correct or not.

(Kernel Ridge) Regression in Brief

Regression, and by proxy Kernel Ridge Regressions are a supervised learning method. They have little restriction on what they can solve but prefer to use continuous variables. They also have the benefit of evening out data and glossing over outliers.

Collaborative Filtering

If you use Amazon to buy things then you have seen collaborative filtering in action. In Amazon's case they want to recommend products interest you so that you end up buying things. So for instance if you buy lots of beer then a good recommendation would be some beer for your consuption.

But where collaborative filtering becomes more interesting is how it relates to other users. Given the fact that you are a beer drinker you might also like kegs, glasses, or even

a set of coasters. Even though you haven't actually bought any of those items other users like you (beer drinkers) have bougtht the items so most likely you'd like them too.

Grahically it'd look something like this:

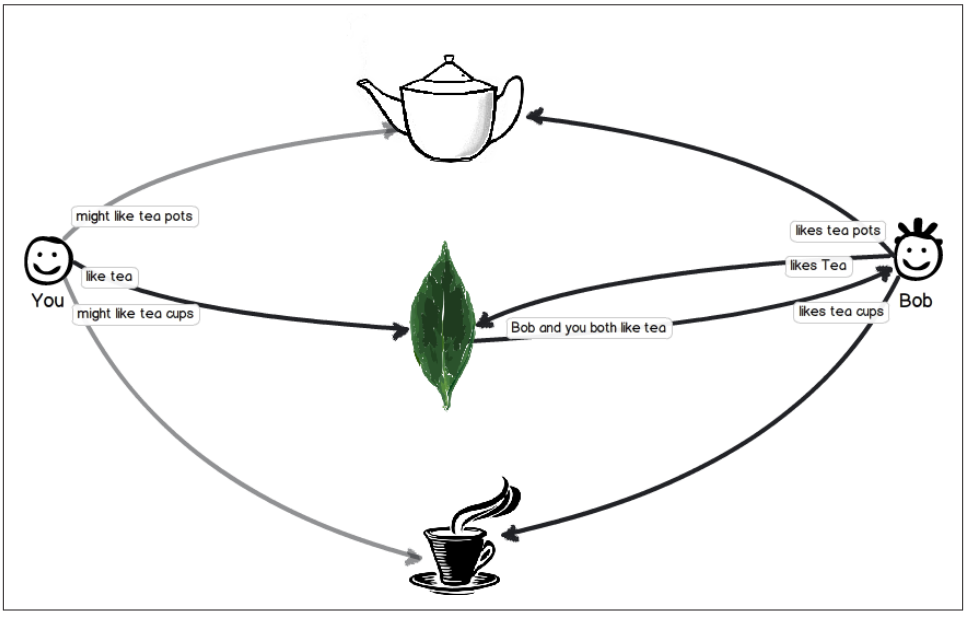


Figure 9-1. You and Bob share a love of beer so therefore you might also like kegs and glasses

There are generally two parts to all collaborative filtering problems:

1. Find users who share the same tastes as you. Otherwise find other beer drinkers.
2. Use the ratings from those like minded users to make recommendations.

Of course this is vague and there are numerous ways of approaching this:

1. Brute force. Naive approach which will become exponentially slow over time
2. Nearest Neighbor Search. Which we learned about in Chapter 2.
3. Regression.

Brute force is really the base case. We could iterate through all possible connections and somehow yield the optimal recommendation for the user, but given the fact that the user will most likely want lots of recommendations on every page this will prove to be slow. Nearest Neighbor search would work well, it's lazy so it wouldn't require a lot of upfront cost but at the same time you are making an assumption about the data and

there isn't much you can determine about a user from nearest neighbor search except that they are like other users.

Lastly we have regression, which would yield a line that fits features to whether the user would like a product or not. The benefit here is that we could determine what a user likes based on the coefficients of the regression as well as use matrix algebra to quickly compute answers. This probably is the best bet. And also simple.

Linear Regression Applied to Collaborative Filtering

You have probably heard of linear regression. The idea is simple, given a dataset fit the best line that approximates that data. For instance let's imagine that we wanted to predict weight given a height. We conceptually know that weight and height are correlated at least. Using the data from the 2012 London Olympics of all eleven thousand contestants (which can be found here: *http://thoughtfulml.com/resources/olympics.csv*).

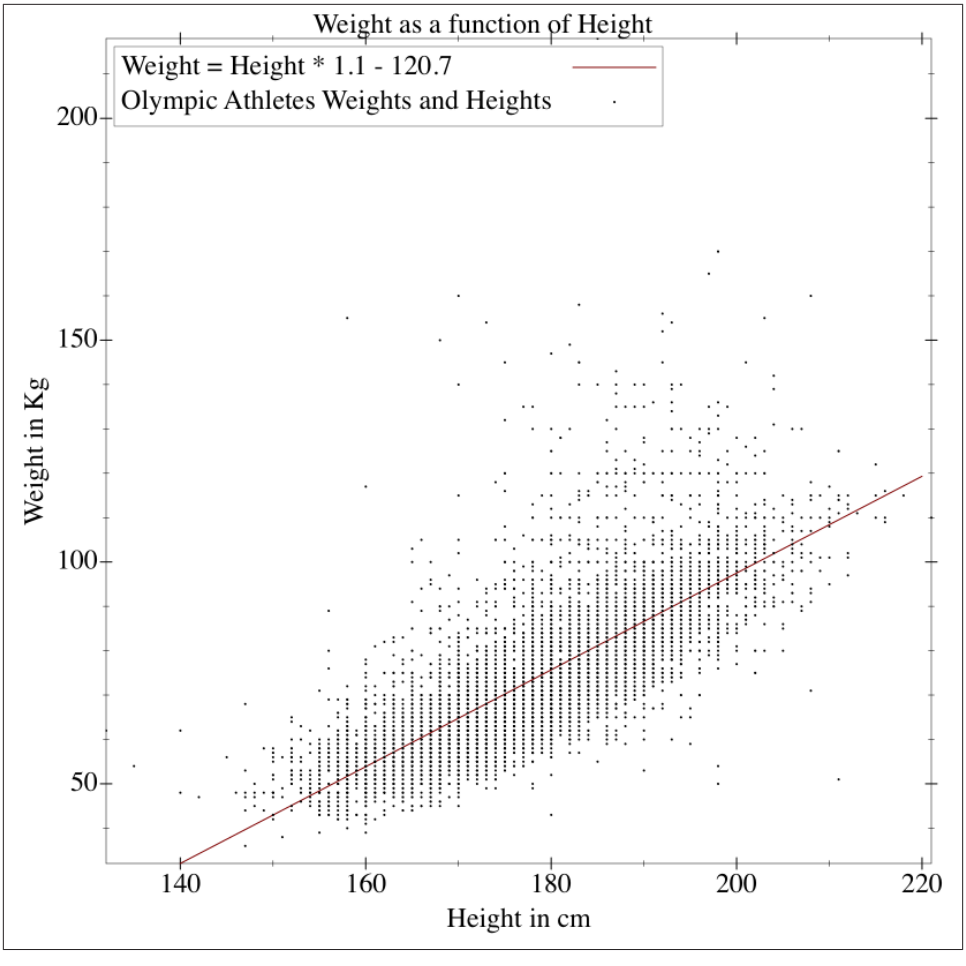


Figure 9-2. We can fit a line that more or less follows the data

This of course makes sense, if I am 5'6" (or 1.68 meters) tall then most likely I will weigh below 200 lbs (91kg). On the other hand if I am 6'8" (2.03 meters) tall then I probably weigh over 250 lbs (113.4 kg). Of course this is glossing over a lot of outliers but that's ok! The whole point of regressions is to *regress back to the mean* the line that ends up happening is actually the mean of the data points in a lot of cases. Now to get into some techincal detail linear regression's main goal is to minimize the square errors of data:

Namely to: $\min \sum_i^n (\hat{y} - y)^2$ Where $\hat{y}$ is what we get out of our regression model. In a linear regression case it would take the form of $\hat{y} = \alpha + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n$. Those βs are just coefficients that we find by minimizing that above function.

The real power of linear regression models is that to find the coefficients β one only needs to do a simple transformation of the original matrix to achieve all of the coefficients.

Using something called the Moore-Penrose psuedoinverse we can actually achieve the answer to the above optimization problem using the following formula.

$$\vec{\beta} = (X^T X)^{-1} X^T y$$

Basically all we need to do is take a transpose (change the i,j element of the matrix to the j,i element) of the training data X and multiply it by itself X. This will yield a square matrix always and then we can invert that using a matrix inverse (I won't get into how to do this cause it's out of the scope of this book). Finally we multply that by the transpose again. What this yields is a vector of coefficients that best matches the data.

A really nice part of this formula is that you effectively get a mean answer for your data. Let's assume that we have a one dimensional problem which is to find a point that is most optimal. Let's say our data points are 1,2,3,4,5,6 and that effectively this is a one dimensional problem. Well we can map this to a two dimensional problem by simply saying that X = 1 for all points and Y equals 1 through 6. What do we find for a beta when we have this problem?

```
require 'matrix'

y = Matrix[[1],[2],[3],[4],[5],[6]]
x = Matrix[[1],[1],[1],[1],[1],[1]]

(x.transpose * x).inverse * x.transpose * y #=> Matrix[[7/2]]
```

What this gives us is the mean! So that is all linear regression is doing is regression to a mean.

A big problem with linear regression though is that the matrix times it's transpose will sometimes be singular (meaning it is not invertible). Singular matricies are also called ill-conditioned because there just isn't enough data to make a justified answer to any equation. They are matricies that have a determinant of zero.

More concretely though it looks something like this in ruby code

```
require 'matrix'

y = Matrix[[1],[2]]

ill_conditioned = Matrix[[1,2,3,4,5,6,7,8,9,10], [10,9,8,7,6,5,4,3,2,1]]
(ill_conditioned.transpose * ill_conditioned).singular? #=> true
(ill_conditioned.transpose * ill_conditioned).inverse #=> Throws an error

conditioned = Matrix[[1,2], [2,1]]
(conditioned.transpose * conditioned).inverse * conditioned.transpose * y #=> Matrix[[(1/1)], [(0/
```

What this shows you is that if for instance you have a lot of variables like the first ill_conditioned problem then there is no suitable way of solving this problem using linear regression just cause there isn't enough data points to find the optimal minimized least squares. The inner matrix will become singular when there are more features than there are data points.

But let's think about this problem a little more.

Introducing Regularization or Ridge regression

From above you remember our regression problem to be ill conditioned was

Table 9-1. Ill conditioned problem

Y	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
1	1	2	3	4	5	6	7	8	9	10
2	10	9	8	7	6	5	4	3	2	1

Even without any algorithm to solve this you can see that a lot of the data here is pretty useless. What we are looking for is a function that will yield 1 in the first case and 2 in the second case, so we're probably wanting to find information that is twice as big in the second case and 1 x in the first. That means that columns like X_1, X_2, X_5, X_6, X_9, and X_10 are probably useless. So let's take it out and look at that.

Table 9-2. Ill Conditioned problem with less columns

Y	X_3	X_4	X_7	X_8
1	3	4	7	8
2	8	7	4	3

Matter of fact X_7 and X_8 are not really needed either so let's forget about them as well. That leaves us with X_3 and X_4. And we can actually solve this.

```
require 'matrix'

y = Matrix[[1],[2]]
simplified = Matrix[[3,4], [8,7]]

betas = (simplified.transpose * simplified).inverse * simplified.transpose * y
# => Matrix[[(1/11)], [(2/11)]]
```

This solves the problem! All we did was get rid of the extraneous variables that didn't make a difference. But the question is can we make this more algorithmic instead of just anecdotatal?

We can use something called a ridge regression or regularlized regression.

The basic idea is to introduce a ridge parameter which will help with ill conditioned problems like what we have seen above

```
require 'matrix'
y = Matrix[[1],[2]]
ill_conditioned = Matrix[[1,2,3,4,5,6,7,8,9,10],[10,9,8,7,6,5,4,3,2,1]]

shrinkage = 0.0001

left_half = ill_conditioned.transpose * ill_conditioned + shrinkage * Matrix.identity(ill_conditio
left_half.singular? #=> false

betas = left_half.inverse * ill_conditioned.transpose * y

betas.transpose * ill_conditioned.row(0) #=> Vector[1.0000000549097194]
betas.transpose * ill_conditioned.row(1) #=> Vector[1.9999994492261521]
```

As you can see adding in this shrinkage factor has helped us get over the problem of singular matricies and we can actually solve this ill-posed problem easily. But there is still one snag with what we are doing which is non-linearity.

Kernel Ridge Regression

If you remember back to the support vector machine chapter, we introduced kernels which would take data that are nonlinear and transform that dataset into a new feature space where all of a sudden it was linear. It's a powerful tool and is well suited for problems where by the data is not linear.

To refresh your memory here are the kernel functions we discussed before in the SVM chapter.

Kernel Functions

1. Homogenous Polynomial: $K(x_i, x_j) = (x_i^T x_j)^d$
2. Heterogenous Polynomial: $K(x_i, x_j) = (x_i^T x_j + c)^d$
3. Radial Basis Function: $K(x_i, x_j) = e^{-\frac{|||x_i - x_j|||_2^2}{2*\sigma^2}}$

The important thing to know here is that we don't actually have to end up calculating a lot of this since a a lot of the time these since they are in addition to the part of the regression where $X^T X$. Basically without delving into the deep mathematics around this we can replace that term with something called K which is a kernel representing a new non-linear space.

So now our equation looks similar $\hat{y} = y^T (K + \lambda I)^{-1} \kappa$

$K_{ij} = f(x_i, x_j)\ \kappa_i = f(x_i, x')$. This is simply a change.

Wrapup of Theory

Kernel Ridge Regressions can be useful for finding simple functions to map an ill posed problem. In the next section we'll see how to actually use this to recommend beer styles to people based on user preferences based on reviews.

Example: Collaborative filtering with beer styles

Remember our collaborative filtering with beer drinkers? What if we were to apply the same to an actual dataset. Namely beer styles (*http://nbviewer.ipython.org/gist/glamp/20a18d52c539b87de2af*). Each of the reviewers entered reviews as to whether they liked the taste, appearance, and other attributes of different selections.

Data Set

This dataset has beer styles, beers, breweries, reviewers and reviews. There are 1,586,615 reviews, 62,260 unique beers, 33,388 reviewers and 5743 breweries and 104 unique beer styles. Up until this point we've been loading everything into memory and analyzing this way but this dataset is large so I think a better approach would be to load this into some sort of database.

Setup Notes

All of the code we're using for this example can be found on GitHub at *https://github.com/thoughtfulml/examples/tree/master/8-kernel-ridge-regression*.

Ruby is constantly changing so the README is the best place to come up to speed on running the examples.

Why Regression for Collaborative filtering?

If you have ever read any other books on machine learning or data science most likely you haven't heard of regression being used for collaborative filtering. In most cases people will build what is called matrix factorization to find recommendations.

We use regression in this because it is well suited for determining the linear combination of factors that will determine what someone wants. The beauty is that you can use this to figure out what someone likes more than not. So for instance in beer reviews we can figure out whether someone likes alcohol more than palate etc. While we can do that with matrix factorization as well this is similar but different.

The tools we will need to accomplish this

To accomplish this we need to build some tables into Postgres. Instead of using ActiveRecord I think Sequel is much easier to use for small projects like this so I'll just use that.

First let's define some tables to use and some models. We will need tables for Beers, Reviews, Reviewers, Breweries, Reviews and Beer Styles.

To start out let's just create our file boostrap which will make sure the tables exist and everything migrates correctly.

```
# script/load_db.rb

# Note that this doesn't have to be postgres you can use sqlite or mysql too

DB = Sequel.connect('postgres://localhost/beer_reviews')

DB.create_table? :beers do
  primary_key :id
  Integer :beer_style_id, :index => true
  Integer :brewery_id, :index => true
  String :name
  Float :abv
end
```

Beers have a beer_style_id, name, abv and brewery_id. We want to make this fairly spread out so a beer_style_id is a foreign key to beer_styles which we will make. ABV is simply alcohol by volume and lastly brewery_id is a foreign key to breweries. Next let's build our breweries as well as everything else:

```
# script/load_db.rb

# DB
# create_table :beers

DB.create_table? :breweries do
  primary_key :id
  String :name
end

DB.create_table? :reviewers do
  primary_key :id
  String :name
end

DB.create_table? :reviews do
  primary_key :id
  Integer :reviewer_id, :index => true
  Integer :beer_id, :index => true
  Float :overall
  Float :aroma
```

```
    Float :appearance
    Float :palate
    Float :taste
  end

  DB.create_table? :beer_styles do
    primary_key :id
    String :name, :index => true
  end
```

Now that we have our data loaded we need to load the information which we can do through this following script:

```
  # script/load_db.rb

  # DB
  # create_table :beers
  # create_table :breweries
  # create_table :reviewers
  # create_table :reviews
  # create_table :beer_styles

  require 'csv'
  require 'set'

  # brewery_id,brewery_name,review_time,review_overall,review_aroma,review_appearance,review_profile
  breweries = {}
  reviewers = {}
  beer_styles = {}

  if !File.exists?('./beer_reviews/beer_reviews.csv')
    system('bzip2 -cd ./beer_reviews/beer_reviews.csv.bz2 > ./beer_reviews/beer_reviews.csv') or die
  end

  CSV.foreach('./beer_reviews/beer_reviews.csv', :headers => true) do |line|
    puts line
    if !breweries.has_key?(line.fetch('brewery_name'))
      b = Brewery.create(:name => line.fetch('brewery_name'))
      breweries[line.fetch('brewery_name')] = b.id
    end

    if !reviewers.has_key?(line.fetch('review_profilename'))
      r = Reviewer.create(:name => line.fetch('review_profilename'))
      reviewers[line.fetch('review_profilename')] = r.id
    end

    if !beer_styles.has_key?(line.fetch('beer_style'))
      bs = BeerStyle.create(:name => line.fetch('beer_style'))
      beer_styles[line.fetch('beer_style')] = bs.id
    end

    beer = Beer.create({
      :beer_style_id => beer_styles.fetch(line.fetch('beer_style')),
```

```
      :name => line.fetch('beer_name'),
      :abv => line.fetch('beer_abv'),
      :brewery_id => breweries.fetch(line.fetch('brewery_name'))
    })

    Review.create({
      :reviewer_id => reviewers.fetch(line.fetch('review_profilename')),
      :beer_id => beer.id,
      :overall => line.fetch('review_overall'),
      :aroma => line.fetch('review_aroma'),
      :appearance => line.fetch('review_appearance'),
      :palate => line.fetch('review_palate'),
      :taste => line.fetch('review_taste')
    })
  end
```

Now that we have loaded our data we can move onto testing and building out our recommendation algorithm using Ridge Regression.

Reviewer

Our first step is to quickly put together associations between all models to do that we write the following:

```
# lib/models/reviewer.rb

class Reviewer < Sequel::Model
  one_to_many :reviews
  one_to_many :user_preferences
end

# lib/models/brewery.rb
class Brewery < Sequel::Model
  one_to_many :beers
end

# lib/models/beer_style.rb
class BeerStyle < Sequel::Model
  one_to_many :beers
end

# lib/models/review.rb
class Review < Sequel::Model
  many_to_one :reviewer
end

# lib/models/user_preference.rb
class UserPreference < Sequel::Model
  many_to_one :reviewer
  many_to_one :beer_style
end
```

At this point we need to build a test for two different scenarios. One of them is that for each rated style we want a assign a nonzero constant. The second one is that we want the highest slope to be the most favorite style, as well we want the smallest slope to equal the least liked beer style.

To test for a correct calculation we write the following test:

```
# test/lib/models/reviewer_spec.rb
describe Reviewer do
  let(:reviewer) { Reviewer.find(:id => 3) }

  it 'calculates a preference for a user correctly' do
    pref = reviewer.preference

    reviewed_styles = reviewer.reviews.map {|r| r.beer.beer_style_id }

    pref.each_with_index do |r,i|
      if reviewed_styles.include?(i + 1)
        r.wont_equal 0
      else
        r.must_equal 0
      end
    end
  end
end
```

Let's just assume that you have loaded in Reviewer into the database and that you can pick a random reviewer to test like id = 3. This test will make sure that there is only zeros for styles not rated and a non zero constant for rated styles.

From here we can test the real question which is whether we can come up with a ranking for beer style likes. We do this by writing the following test:

```
# test/lib/models/reviewer_spec.rb

describe Reviewer do
  let (:reviewer) { Reviewer.find(:id => 3) }

  # Test from above

  it 'gives the highest rated beer_style the highest constant' do
    pref = reviewer.preference

    most_liked = pref.index(pref.max) + 1

    least_liked = pref.index(pref.select(&:nonzero?).min) + 1

    reviews = {}
    reviewer.reviews.each do |r|
      reviews[r.beer.beer_style_id] ||= []
      reviews[r.beer.beer_style_id] <<  r.overall
    end
```

```
    review_ratings = Hash[reviews.map {|k,v| [k, v.inject(&:+) / v.length.to_f] }]

    assert review_ratings.fetch(most_liked) > review_ratings.fetch(least_liked)

    best_fit = review_ratings.max_by(&:last)
    worst_fit = review_ratings.min_by(&:last)

    assert best_fit.first == most_liked || best_fit.last == review_ratings[most_liked]
    assert worst_fit.first == least_liked || worst_fit.last == review_ratings[least_liked]
  end
end
```

Now of course we need to write the actual code to make this work.

Writing the code to figure out someone's preference.

The problem we are trying to solve with these two tests is really finding a linear combination of beer styles to average ratings overall. As we know there are around 104 beer styles and most users won't review that often. So therefore we will probably have singular matricies and regression won't work right out of the box. So instead we have to build an algorithm that will shrink the matrix enough so that it can invert. We do this by exponentially increasing the shrinkage parameter until it works.

You will notice that I am using the NMatrix library which is a subset of NArray. This is purely for speed. Unfortunatly the Matrix library in Ruby is slow and inefficient so to do lots of calculations we have to utilize the NMatrix library. There are downsides to this though which are that NMatrix is really just a simple hack on top of NArray and doesn't have some nice features like determinants or other real neat tools. So I created a class called MatrixDeterminance that takes a matrix and calculates it's determinant.

```
# lib/matrix_determinance.rb

require 'narray'
require 'nmatrix'

class MatrixDeterminance
  def initialize(matrix)
    @matrix = matrix
  end

  def determinant
    raise "Must be square" unless square?
    size = @matrix.sizes[1]
    last = size - 1
    a = @matrix.to_a
    no_pivot = Proc.new{ return 0 }
    sign = +1
    pivot = 1.0
    size.times do |k|
```

```
        previous_pivot = pivot
        if (pivot = a[k][k].to_f).zero?
          switch = (k+1 ... size).find(no_pivot) {|row|
            a[row][k] != 0
          }
          a[switch], a[k] = a[k], a[switch]
          pivot = a[k][k]
          sign = -sign
        end
        (k+1).upto(last) do |i|
          ai = a[i]
          (k+1).upto(last) do |j|
            ai[j] = (pivot * ai[j] - ai[k] * a[k][j]) / previous_pivot
          end
        end
      end
      sign * pivot
    end

    def singular?
      determinant == 0
    end

    def square?
      @matrix.sizes[0] == @matrix.sizes[1]
    end

    def regular?
      !singular?
    end
  end
```

The way to use this is simple: Just initialize a new MatrixDeterminance object and you can calculate whether that matrix is singular, regular or whate the determinant is.

Back to the original problem we can calculate similar to what we did above using a shrinkage parameter and matrix transformations like such.

```
# lib/models/reviewer.rb

class Reviewer < Sequel::Model
  one_to_many :reviews
  one_to_many :user_preferences

  IDENTITY = NMatrix[
    *Array.new(104) { |i|
      Array.new(104) { |j|
        (i == j) ? 1.0 : 0.0
      }
    }
  ]

  def preference
```

```
      @max_beer_id = BeerStyle.count
      return [] if reviews.empty?
      rows = []
      overall = []

      context = DB.fetch(<<-SQL)
        SELECT
          AVG(reviews.overall) AS overall
          , beers.beer_style_id AS beer_style_id
        FROM reviews
        JOIN beers ON beers.id = reviews.beer_id
        WHERE reviewer_id = #{self.id}
        GROUP BY beer_style_id;
      SQL

      context.each do |review|
        overall << review.fetch(:overall)
        beers = Array.new(@max_beer_id) { 0 }
        beers[review.fetch(:beer_style_id) - 1] = 1
        rows << beers
      end

      x = NMatrix[*rows]
      shrinkage = 0

      left = nil
      iteration = 6

      xtx = (x.transpose * x).to_f

      left = xtx + shrinkage * IDENTITY

      until MatrixDeterminance.new(left).regular?
        puts "Shrinking iteration #{iteration}"
        shrinkage = (2 ** iteration) * 10e-6

        (left * x.transpose * NMatrix[overall].transpose).to_a.flatten
      end
    end
  end
```

You can see here that this is a pretty fat method but let's go through it together. The first step is to find what the maximum BeerStyle id is. Our eventual vector will be this wide. Next we set up a context which is simply the average review for each reviewed beer style. Next we set the reviewed beer style ids to 1.

Finally we get into the actual regression problem which is where we are trying to map overall review to the beer styles. From here we iterate the shrinkage parameter until the matrix is invertible so we can actually regress. Finally we find the slope parameters and return that.

Of course you are probably wondering what we can do with this. It is just a slope of how much someone likes a beer. This preference can persist by storing it in the table user_preferences. This would be a beer_style_id and a preference. From that table we can then graphically move from top preferences to other reviewers who reviewed the same beer and liked it.

This is a form of collaborative filtering. We have used the preference of the user and using that we can move in a graph to the next user.

Collaborative Filtering with User Preferences

To achieve what is effectively a form of collaborative filtering we want to find users who have a similar taste and then find out what they like that we haven't tried yet. Making this work in code we would do the following:

```
# lib/models/reviewer.rb

class Reviewer < Sequel::Model
  def friend
    skip_these = styles_tasted - [favorite.id]

    someone_else = UserPreference.where(
      'beer_style_id = ? AND beer_style_id NOT IN ? AND reviewer_id != ?',
      favorite.id,
      skip_these,
      self.id
    ).order(:preference).last.reviewer
  end

  def styles_tasted
    reviews.map { |r| r.beer.beer_style_id }.uniq
  end

  def recommend_new_style
    UserPreference.where(
      'beer_style_id NOT IN ? AND reviewer_id = ?',
      styles_tasted,
      friend.id
    ).order(:preference).last.beer_style
  end
end
```

What this yields us is a method `recommend_new_style` that will tell us a new style to taste. The best part about this is that we don't really need to test explicitly because we have already tested the preference. That's all we need.

Conclusion

Ridge Regression is a useful tool to find quick solutions to problems that are ill posed. If the x variables is higher than the actual observations. This happens a lot with reviews. Users get bored and want to move on. While you could use other methods a ridge regression will work perfectly fine.

As you saw they are powerful considering that we found a preference of beer styles and implemented a collaborative filter of sorts. After calculating the preference then it can find a similar preference and recommend from that.

CHAPTER 10

Improving Models and Data Extraction

Sometimes no matter how good an algorithm is it just doesn't work. Or worse it doesn't pick up anything. Data can be quite noisy and sometimes it's just about impossible to figure out what went wrong. This chapter is about improving what you have already by either selecting better features, or transforming your features into a new set. We do this by monitoring metrics as it relates to either cross validations or production monitoring.

This chapter will be somewhat of a smorgasbord of tastes when it comes to fixing up your models. That is because there are many ways of fixing models.

The problem with the Curse of Dimensionality

As we've talked about before the curse of dimensionality is a big problem with distance based machine learning algorithms. Generally speaking as the amount of dimensions goes up the average distance also goes up. Take for instance this case where we look at a perfect sphere centered around 0,0,0.

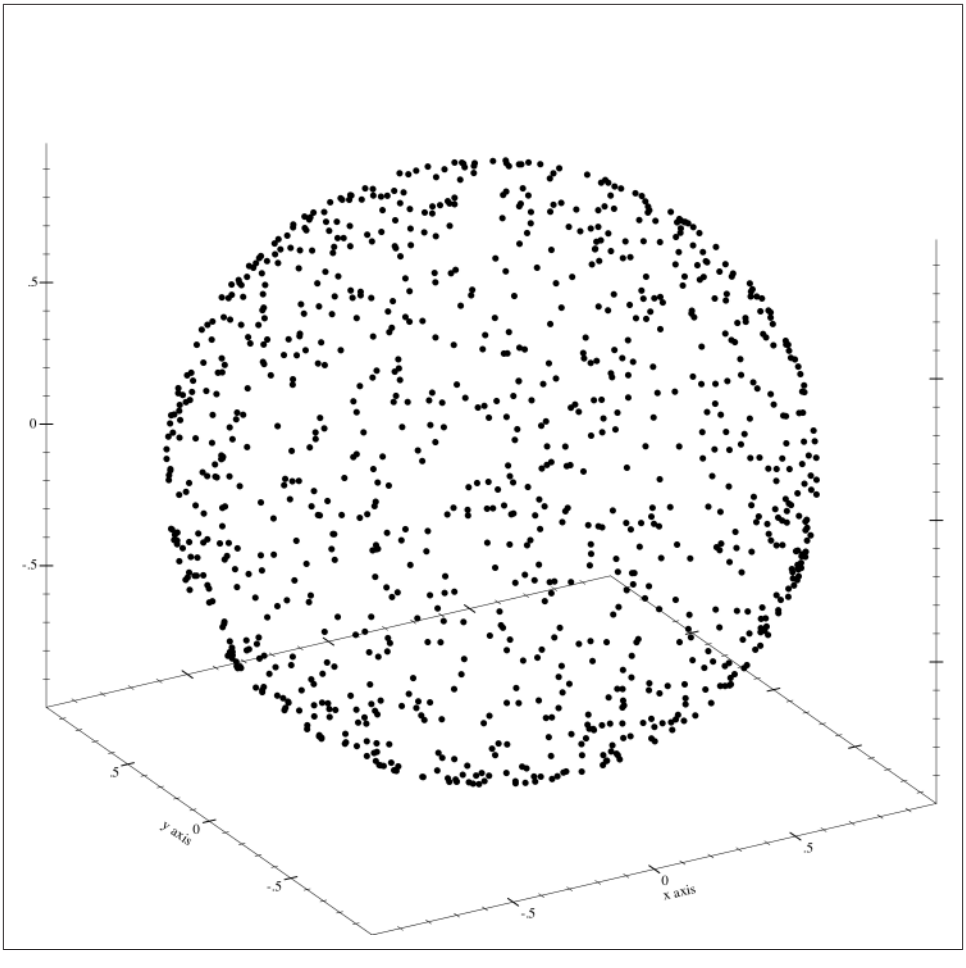


Figure 10-1. In the case of 3 dimensions the average distance is 1 because it is perfect

Everything is fine in 3 dimensions but what if we project only onto two dimensions What ends up happening is quite illuminating.

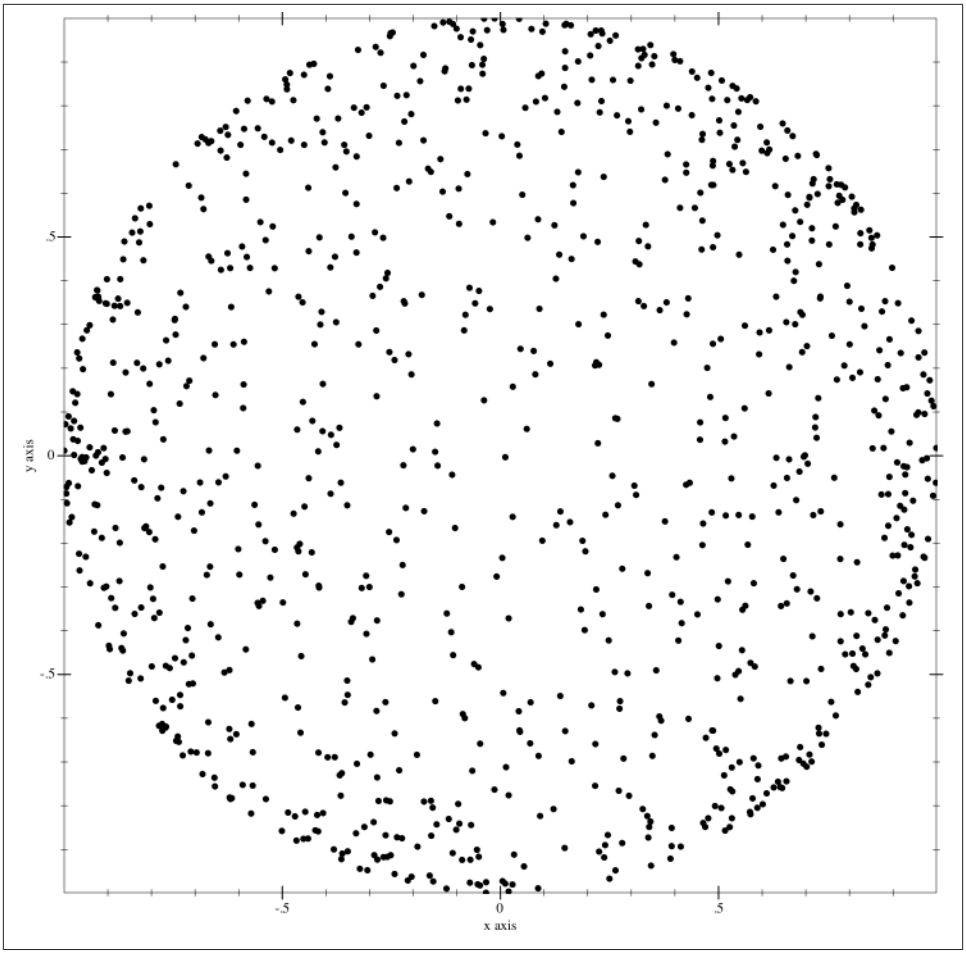


Figure 10-2. In this case of dimenison = 2 the average distance is 0.74

In these graphics you'll see a unit sphere on the left. This unit sphere was created using random points along the outer edge. On the right you see a circle but it is in fact the same exact sphere projected onto a two dimensional space. In the first case the sphere is a unit so therefore the distance to the edge is exactly 1 whereas the average in the circle is 0.74. This means that as you project onto less and less dimensions the distances become shorter. We have already solved this in the K-Nearest Neighbor chapter by introducing SURF for feature extraction. Instead of trying to find the nearest neighbor of all pixels we needed to use a smaller data set. As we know the only way of getting over the curse is to reduce the dimensions.

In this section we will discuss two different methods of overcoming the curse of dimensionality, "Feature Selection" and "Feature Transformation".

Feature Selection

Let's think about some data that doesn't make a whole lot of sense. Let's say that we want to measure weather data and want to be able to predict temperature given three variables: "Coffee Consumption", "Ice cream consumption" and "Season".

Table 10-1. Weather Data for Seattle

Average Temperature	Matt's Coffee Consumption	Ice cream consumption	Month
47F	4	2	Jan
50F	4	2	Feb
54F	4	3	Mar
58F	4	3	Apr
65F	4	3	May
70F	4	3	Jun
76F	4	4	Jul
76F	4	4	Aug
71F	4	4	Sep
60F	4	3	Oct
51F	4	2	Nov
46F	4	2	Dec

Obviously you can see that I generally drink about 4 cups of coffee a day. I tend to eat more ice cream in the summer time and it's generally hotter around that time.

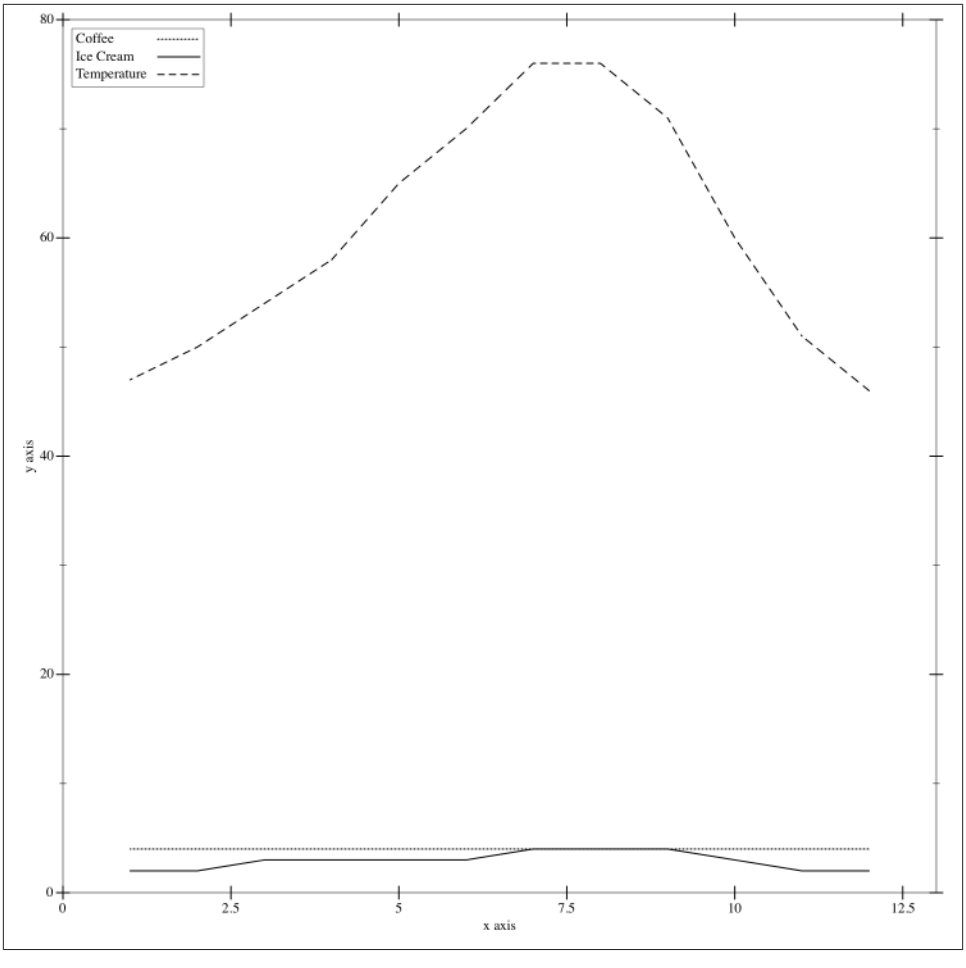


Figure 10-3. A graph showing my ice cream consumption

Now we know inherently that the one thing that causes temperature changes is the season. It is generally hotter during the summer, and cooler during the winter. That is because the sun is in a different place in the sky. Ice cream consumption doesn't matter and neither does coffee consumption. Although ice cream consumption correlates with the hotter months.

The thing to realize here though is that we have one feature that is irrelevant and will just skew data towards it, on the other hand we have a composed variable which is really a combination of other variables outside of our model and finally we have a signal which is the month.

Say for ins tance we wanted to take a random subset of variables and test whether that improved our data. We can do that. This is called random feature selection and actually

ends up working a lot of the time. The reason usually has to do with data being better behaved in smaller dimensions. So even though we're randomly reducing dimensions we are improving our data.

The basic idea is to take a random subset of data and to run your model against it then. Random feature selection is probably one of the simplest model improvement tools out there.

But there is one big downside to doing this and that is slowness. Unfortunately wrapping a model with some feature selection will take a long time. That is because you need to take data randomly select a subset and run your model, then test the fitness of that. As you could expect this will take a long time to achieve. But in the case of K-Nearest Neighbors or other fast algorithms this might be good enough. But what if you were to give a neural network better data or something that might take some time? For that we can filter our data even before it gets sent to the algorithm.

Feature Transformation

To understand feature transformation let's think about the case of keeping a food journal. A lot of us decide on dieting and tracking calories. Let's just say you want to track when you are hungry vs when you are not. So you keep a log of what hour of the day it is and how hungry you are.

The only problem is that you have spent time in Los Angeles and Hawaii when you were writing down your hunger for the hour. The data you have collected is:

Table 10-2. Hunger Log

Hungry?	Hour of Day	Timezone Offset
Yes	7	-8
No	8	-10
No	9	-8
Yes	9	-10
Yes	12	-8
No	14	-10
No	15	-8
No	16	-8
No	18	-10
No	19	-10
Yes	18	-8
Yes	20	-10

Looking at this data would yield a noisy look into when you get hungry:

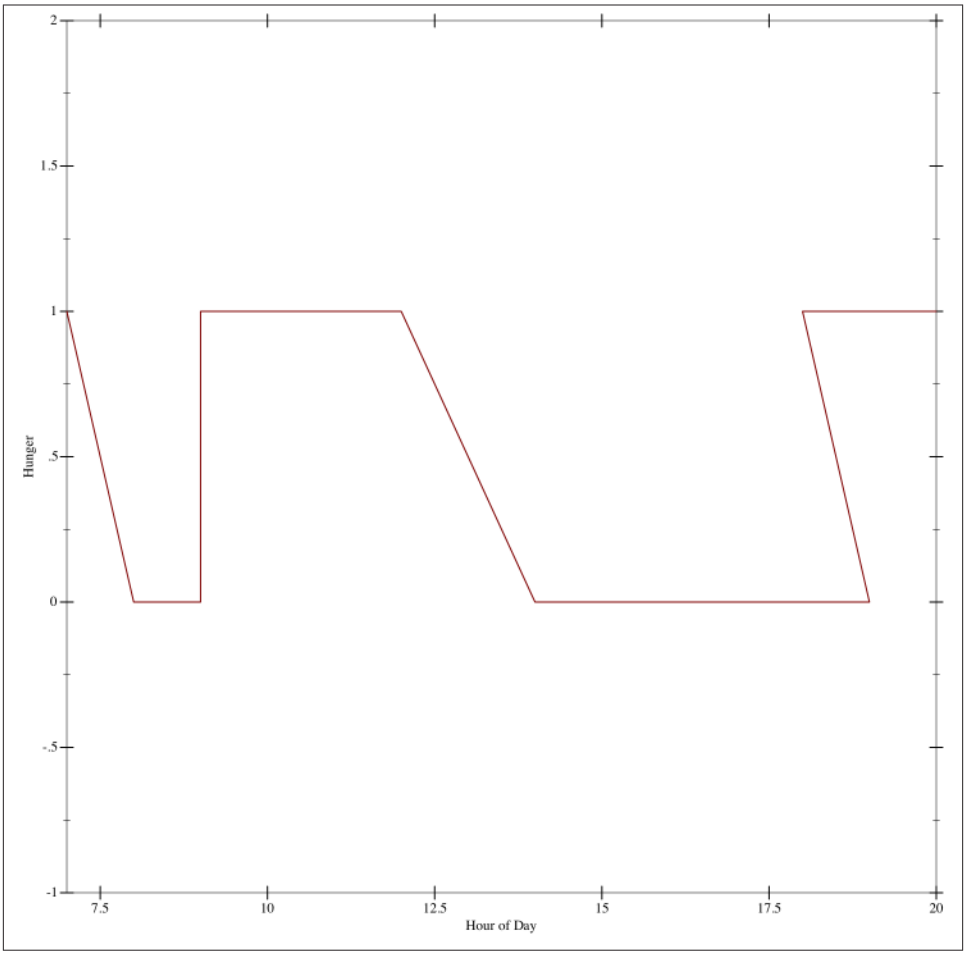


Figure 10-4. Not a very definite trend

You can see that there doesn't seem to be any sort of pattern to eating patterns. It just kind of is either hungry or not.

But instead if you combine hour of day and timezone offset together in a linear combination then you'll notice that in fact you get hungry three times a day. At 7, 12, and 6pm at night. You'd get something completely different and that'd look like this.

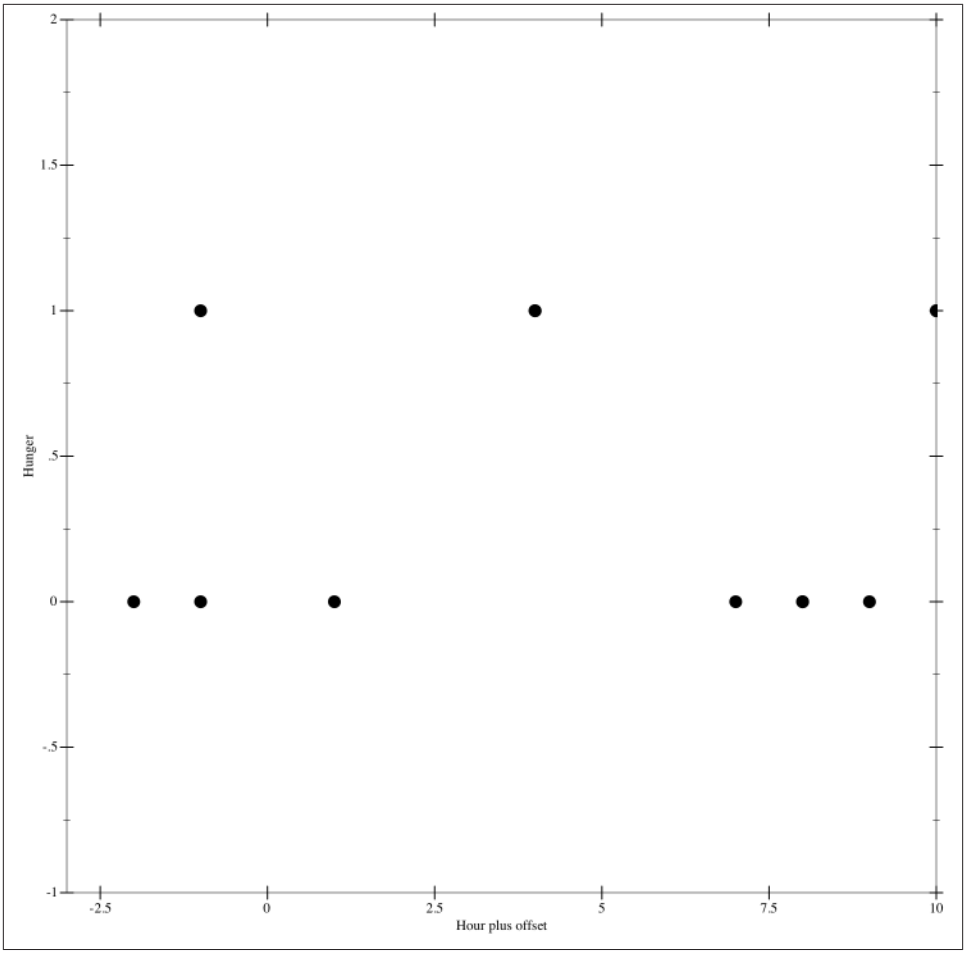


Figure 10-5. Transformed to sum offset and hour of day

There is a better way of doing this and that is through feature transformation algorithms. There are a bunch of different types but we're going to focus on PCA and ICA.

Principal Component Analysis (PCA)

Principal component analysis has been around for a long time. This algorithm simply looks at the direction with the most variance and then determines that as the first principal component. This is very similar to how regression works in that it determines the best direction to map data to. Graphically imagine you have a noisy data set that looks like this:

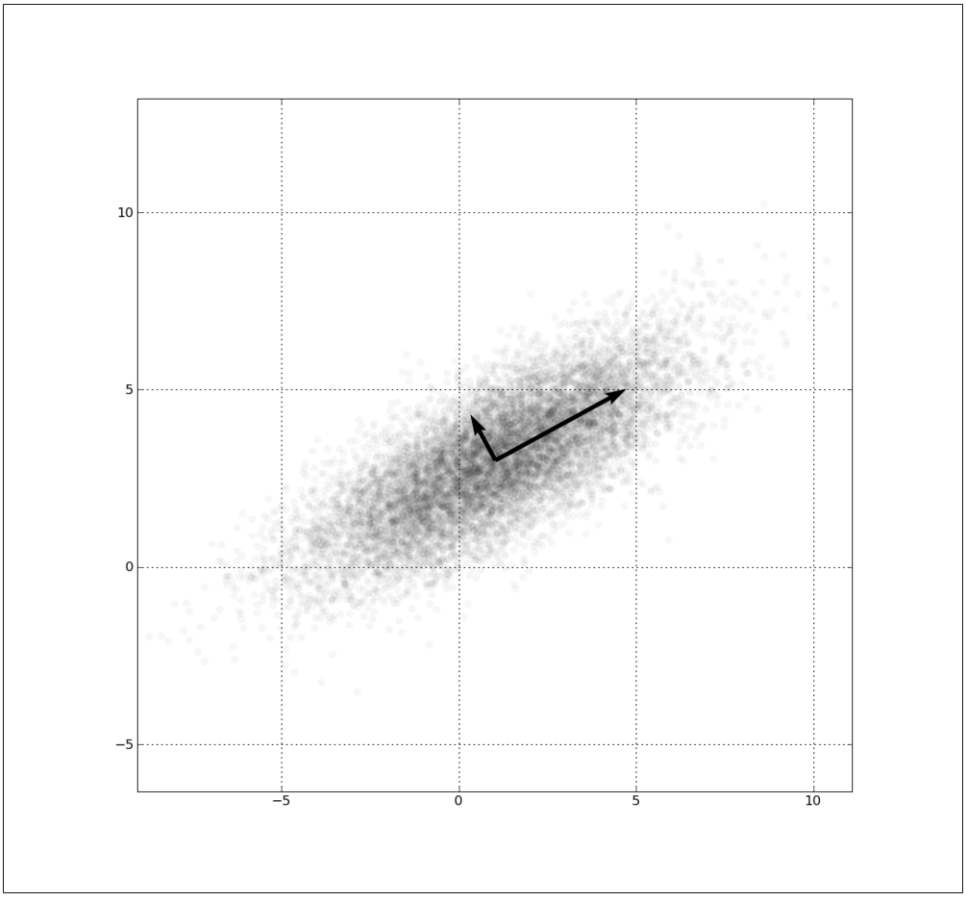


Figure 10-6. This shows principal components in the scatter plot

As you can see the data has a definite direction up and to the right. If we were to determine the principal component it would be that direction because the data is in maximal variance in that way. The second principal component would end up being orthogonal to that and then over iterations we would reduce our dimensions by transforming dimensions into these principal directions.

Another way of thinking about PCA is how it relates to faces. When applying PCA to a set of faces an odd result happens which is called the Eigenfaces.

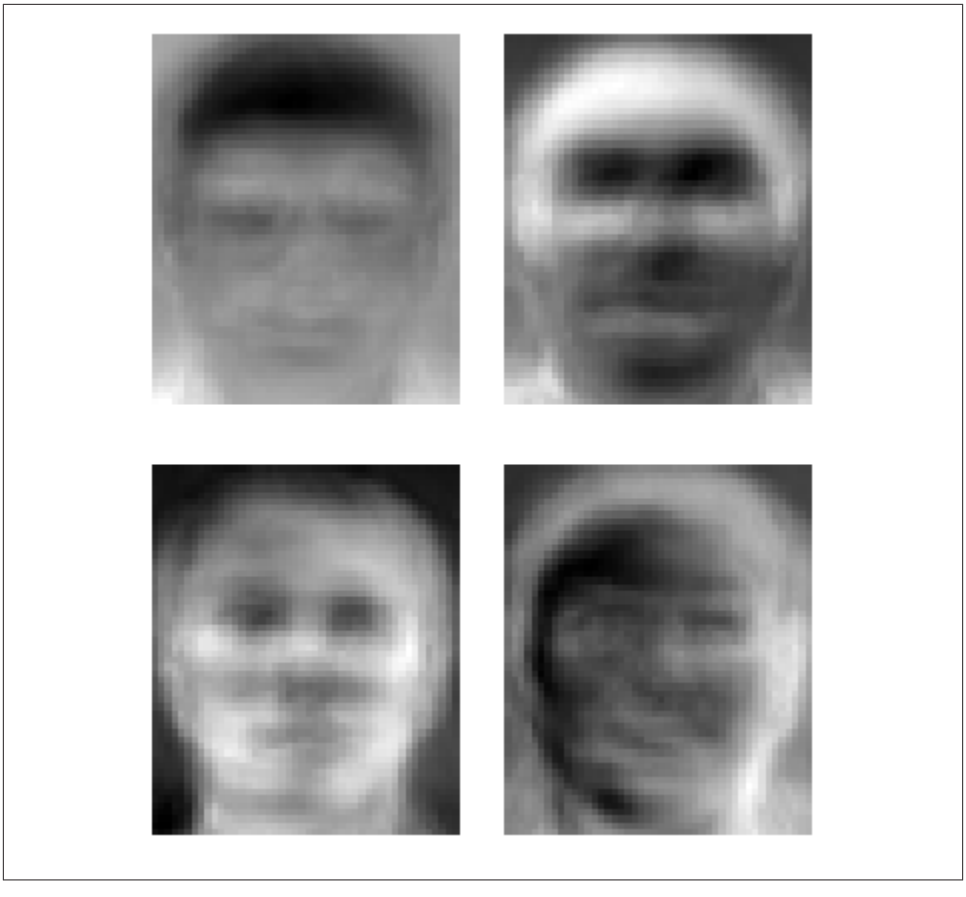


Figure 10-7. Eigenfaces from AT&T Laboratories Cambridge

While these look quite odd it is fascinating that what comes out is really an average face summed up over all of the training data. Instead of implementing PCA I'm going to wait until the next section where we implement ICA which actually relies on PCA as well.

Independent Component Analysis (ICA)

Imagine you are at a party and your friend is coming over to talk to you. Near you is a loud mouth that you hate who won't shut up, and on the other side of the room is a washing machine that keeps making noise.

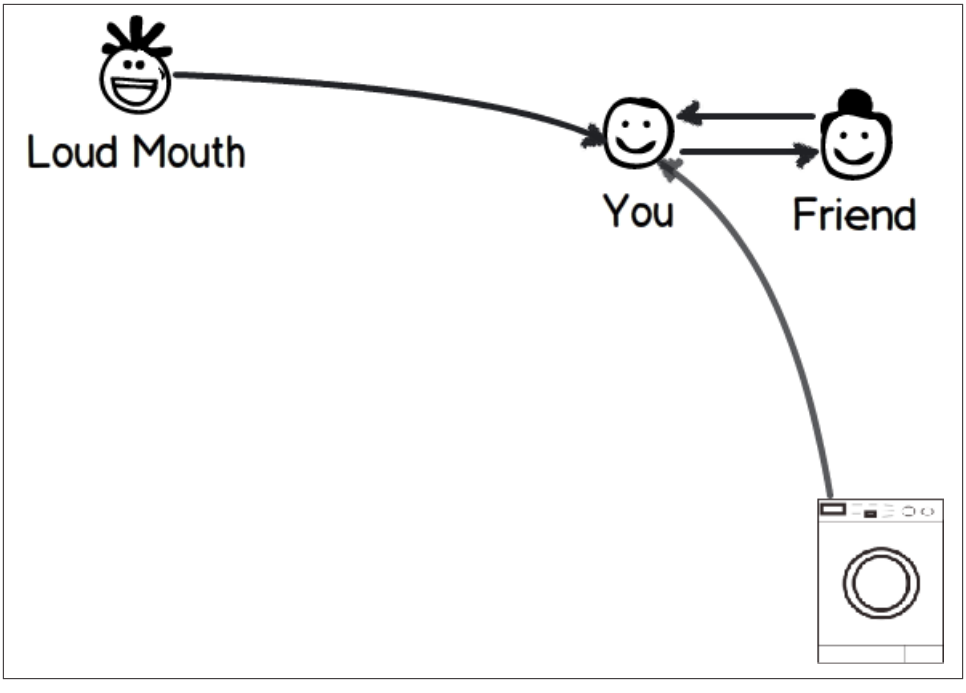


Figure 10-8. Cocktail party example

You want to listen to what your friend has been up so you listen to her closly to figure out what she's been up to. Being human you are superb at separating out the sounds like a washing machine and that loudmouth you hate. But how could we do that with data?

Let's say that instead of listening to your friend you only had a recording and wanted to filter out all of the noise in the background. How would you do something like that? There actually is an algorithm called Independent Component Analysis which does that.

Technically it does that by minimizing mutual information, or the information shared between the two variables. Intuitively this makes sense find me the signals in the aggregate that are different.

Compared to our face recognition above what does ICA extract? Well unlike eigenfaces it extracts pieces of a face, like noses, eyes, hair and other pieces.

Both algorithms are useful for transforming data and can analyze information even more.

Unfortunately just like PCA there is no Ruby Gem for ICA. Instead we can rely on the "R in Ruby" gem to call into R. This is definitely cheating for a book on using machine learning in Ruby but sometimes it is perfectly useful to call out to other languages.

Another possibility would be to use JRuby and utilize FastICA which is actually what gets used in R as well.

Setup Notes

To get these examples running check out *https://github.com/thoughtfulml/examples/tree/master/9-improving-models-and-data-extraction*

You will have to install R to get this to work properly.

The first step we need to achieve is to install fastICA in R we can do that by typing the following

```
# example_1.rb

require 'rinruby'

R.eval(<<-R)
  install.packages("fastICA")
  library(fastICA)
  S <- matrix(runif(10000), 5000, 2)
  A <- matrix(c(1, 1, -1, 3), 2, 2, byrow = TRUE)
  X <- S %*% A
  a <- fastICA(X, 2, alg.typ = "parallel", fun = "logcosh", alpha = 1,
  method = "C", row.norm = FALSE, maxit = 200,
  tol = 0.0001, verbose = TRUE)
  par(mfrow = c(1, 3))
  plot(a$X, main = "Pre-processed data")
  plot(a$X %*% a$K, main = "PCA components")
  plot(a$S, main = "ICA components")
R
```

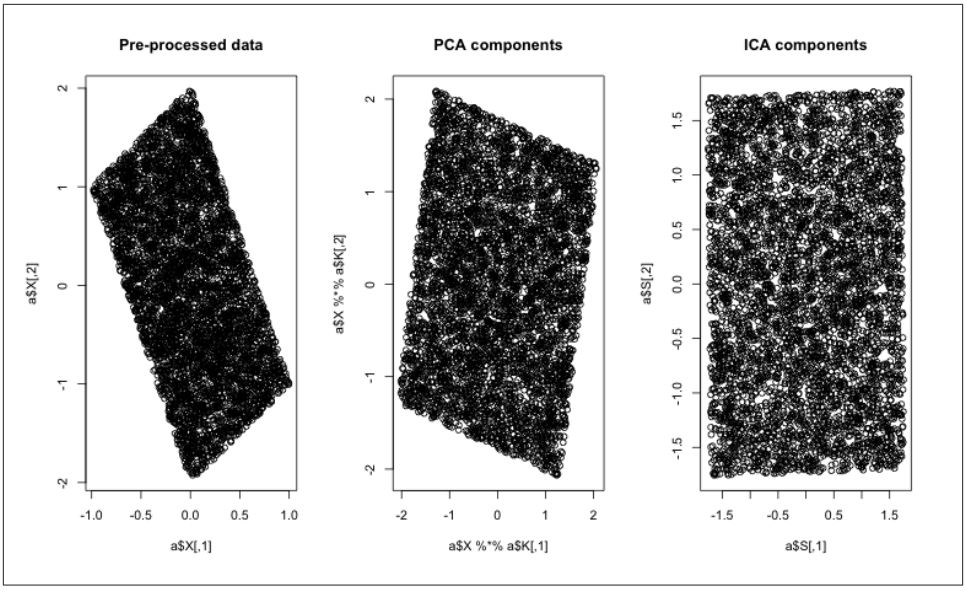


Figure 10-9. Random data that is mapped using PCA and ICA

Now that we know about feature transformation, feature selection and other topics let's get into the last part of this chapter which is monitoring performance of machine learning in a production environment.

Monitoring Machine Learning Algorithms

Throughout the book we've talked a lot about writing tests, cross validation and Ockham's Razor but little about monitoring of the code. If production code seems fine then you aren't measuring enough. When it comes to machine learning things are no different, just because these algorithms can seem magical at times doesn't mean that we can just deploy without continually testing our code.

The tools we can use are generally measuring precision recall and alerting when anomalies happen. We will also discuss an online way to measure mean squared error.

Precision and Recall Example: Spam Filter

Remember our Spam Filter? In general we want to mark email as either Spam or Ham. Since e-mail is important we talked about the notion that we would rather have less precision, or have spam messages float into our inbox, than to miss an important message where someone sounded like spam.

Lets say that we have found that the following is true

Table 10-3. Spam vs Ham based on experience

	Predicted Ham	Predicted Spam
Ham	10000	100
Spam	40	60

Looking at this, we are correctly calculating that 10000 Ham messages are in fact Ham. Unfortunately 100 of the actual Ham messages classified as Spam.

This is useful information and when we cross validated we used this to optimize our model. But what if we wanted to monitor this? That is where Precision, Accuracy, and Recall comes in.

Precision is the amount of correctly classified spam divided by the predicted spam. That would be 60 / 160 or 3/8. This is bad. What this means is that the algorithm gives us more irrelevant examples for Spam. So it thinks that Ham is actually Spam when it isn't.

Accuracy is the correctly classified examples divided by the total amount of examples used. So in this case it would be (10000 + 60) / (10000 + 100 + 40 + 60) or roughtly 98.6%. This isn't too bad, this means that 98.6% of the examples are correct. We have used this throughout the book extensively measuring error rates.

Lastly we have Recall which is 60 / (60 + 40) or 3/5. This means that we got more than half of the Spam examples which is good.

These metrics are useful inside of cross validations but I find them much better suited for monitoring. In a lot of ways it's important to measure all of these as users interact and feed more data into the algorithm.

So for instance let's say that our spam filter really doesn't work well and people keep marking new things as Spam when it was Ham. What would that do to our metrics?

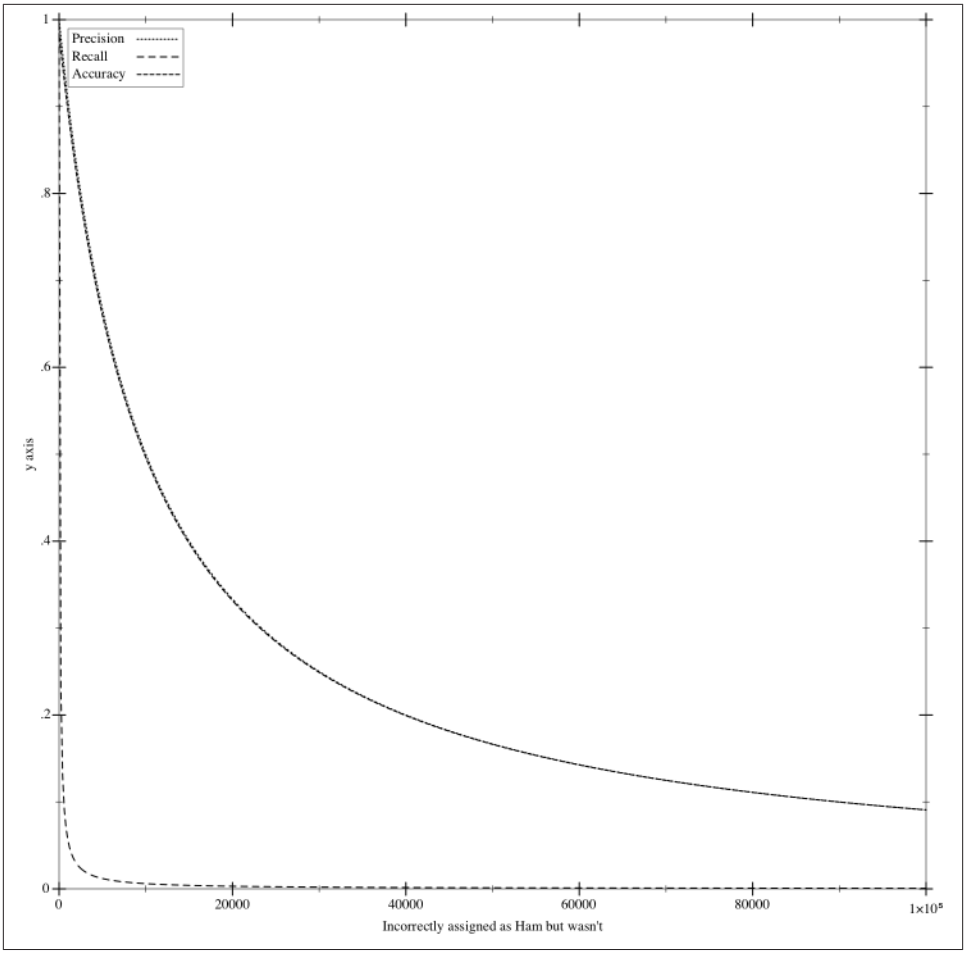


Figure 10-10. The decay of precision, recall, and accuracy

Table 10-4. Data supporting Graph

Predicted as Ham but isn't	Ham Pecision	Recall	Accuracy
0	100%	100%	99%
10	99%	85%	98.9%
20	99%	75%	98%
30	99.7%	66%	98%
..	..	..	..
60	99.4%	50%	98%
..	..	..	..
100	99%	37.5%	98%

As you can see as false negatives will have little impact on a problem with lots of training in terms of accuracy but when it comes to recall it will. When recall dips below 50% is when the model becomes less viable and monitoring alerts should be happening.

The Confusion Matrix

We've been talking about false and positive predictions but there is a more general term for this and it's called the confusion matrix. This relates to correctly classified instances. So for example let's say we classify into three beer categories "Pilsner", "Stout", and "Hefeweizen". And given our classification algorithm we found the following data

Table 10-5. Confusion Matrix

	Pilsner	Stout	Hefeweizen
Pilsner	20	1	3
Stout	1	30	1
Hefeweizen	5	1	10

These are different beers so our algorithm found a decent classification of the examples. From this confusion matrix you can calculate all Recalls, Precision and overall accuracy of the model.

For instance the precision of Stout is 30 / 36 or around 86%. The recall of Stout would be 30 / 32 or 93.75%.

Unfortunately Confusion matrices have one downside which is that they only work with discrete classification problems, what about something like a regression or an algorithm that returns a continuous variable? For that we have to rely on mean squared error.

Mean Squared Error

When it comes to measuring an algorithm like predictions of a continuous variable like a review we need to take a different approach which is to measure the mean squared error of the model. But unfortunately that is difficult to do in a production context because you'd have to keep all classifications around in the past and errors. Actually not so! There is a workaround for this.

So our goal is to measure mean squared error over time as new information comes in. Let's say we have a model $\hat{y} = f(x)$ where y is some real number. Assuming we can get some information from our users like what they'd actually rate whatever it is we could then determine an error $\epsilon = (\hat{y} - y)^2$. We square the error mainly to make it positive.

This is where most people would think that you would have to hold on to all errors and calculate $\sum_{i=0}^{n} \epsilon_i$. But instead we can calculate an incremental squared error instead. First let's rewrite this as $\bar{\epsilon}_n = \frac{\epsilon_1 + \epsilon_2 + \cdots + \epsilon_n}{n}$. Likewise we can say the next average will equal

$\overline{\epsilon_{n+1}} = \frac{\epsilon_1 + \epsilon_2 + \cdots + \epsilon_n + \epsilon_{n+1}}{n+1}$. Now if we multiply the first equation by n we can actually just input that into the fraction. So $\overline{\epsilon_{n+1}} = \frac{n * \epsilon_n + \epsilon_{n+1}}{n+1}$. This means that the next average is simply the previous average multiplies by the amount of used instances divided by the new instances.

So let's say that our current mean squared error is 2 calculated over 10 iterations. We find out in iteration 11 that the squared error was 100. That means that the new average should equal (2 * 10 + 100) / 11 which equals around 10.9.

This means that we can easily build a program that monitors mean squared error in production.

```
# incremental_meaner.rb

class IncrementalMeaner
  attr_reader :current_mean, :n
  def initialize
    @current_mean = 0
    @n = 0
    @mutex = Mutex.new
  end

  def add(error)
    @mutex.synchronize {
      @current_mean = (((@n * @current_mean) + error) / (@n + 1.0)
      @n += 1
      @current_mean
    }
  end
end
```

The Wilds of Production Environments

As they say anything that can go wrong probably will. This is more especially true of production environments. We can cross validate, test our seams and determine that our model is good but when it is in production it can also break down. User input is one thing that people can't optimize for because people will do the funniest things. Instead it's important to build monitoring around mean squared error as well as precision recall metrics to have more confidence in how the algorithm is performing.

This also has an architectural component which is that there needs to be a feedback loop with algorithms. There needs to be something to test against. Otherwise the code will not work and you'll end up with the problem that websites have with poorly trained information that becomes stagnant and useless. In the end user experience is what we are trying to optimize with machine learning algorithms.

Conclusion

This chapter was kind of a smorgasbord of different tastes of ways to improve a model you have. Sometimes you can just select features that work better, sometimes you need to transform but most especially the important part is to make sure that we are measuring our results against a baseline and monitoring success in production or the success with our users.

CHAPTER 11
Putting it all together

Well here we are! The end of the book. While you probably don't have the same depth of understanding as a Ph.D in Machine Learning but I hope you have learned something. But more especially a thought process to approaching problems that machine learning works so well at. I firmly believe that using tests is the only way that we can effectively use the scientific method. It is the reason the modern world exists and helps us become much better at writing code.

Of course you can't write a test for everything but it's the mindset that matters. And hopefully you have learned a bit about how you can apply that mindset to machine learning. In this chapter we will discuss what we covered in a high level and we can post some further reading for you if you'd like to delve further into machine learning research.

Machine learning algorithms revisited

While we touched on this in the book, machine learning splits into three main categories. Supervised, Unsupervised and Reinforcement Learning. This book blatantly skips reinforcement learning but I highly suggest you read into it now that you have a better understanding you can delve into reinforcement learning, I'll put a source that you can read into in the further reading section.

Table 11-1. Machine Learning Categories

Category	Description
Supervised	Supervised learning is the most common machine learning category. This is functional approximation. We are trying to map some data points to some fuzzy function. Optimization wise we are trying to fit a function that best approximates the data to use in the future. It is called supervised because it has a learning set given to it.
Unsupervised	Unsupervised learning is just analyzing data without any sort of Y to map to. It is called unsupervised because the algorithm doesn't know what the output should be and instead has to come up with it itself.

Category	Description
Reinforcement	Similar to supervised learning with a reward that emits from each step. For instance this is like a mouse looking for cheese in a maze. The mouse wants to find the cheese and in most cases will not be rewarded until the end when it finally finds it.

On top of these categories there are generally two types of biases for each. One is the restriction bias and the other is preference. Restriction bias is basically what limits the algorihtm while preference is what sort of problems it prefers.

All of this information helps us determine whether we should use each algorithm or not.

Table 11-2. Machine Learning Algorithm Matrix

Algorithm	Type	Class	Restriction Bias	Preference Bias
KNN	Supervised Learning	Instance Based	Generally speaking KNN is good for measuring distance based approximations, it suffers from the curse of dimensionality	Prefers problems that are distance based
Naïve Bayes	Supervised Learning	Probabilistic	Works on problems where the inputs are independent from each other	Prefers problems where the probability will always be greater than zero for each class
SVM	Supervised Learning	Decision Boundary	Works where there is a definite distinction between two classifications	Prefers binary classification problems
Neural Networks	Supervised Learning	Non linear functional approximation	Little restriction bias	Prefers binary inputs
(Kernel) Ridge Regression	Supervised	Regression	Low restriction on problems it can solve	prefers continuous variables
Hidden Markov Models	Supervised / Unsupervised	Markovian	Generally works well for system information where the markov assumption holds	Prefers timeseries data and memoryless information
Clustering	Unsupervised	Clustering	No restriction	Prefers data that is in groupings given some form of distance (Euclidean, Manhattan, or others).
Filtering	Unsupervised	Feature Transformation	No restriction	Prefer data to have lots of variables to filter on

How to use this information for solving problems

Using the above matrix we can figure out when given a problem how to approach it. For instance if we are trying to solve a problem like determining what neighborhood

someone lives in KNN is a pretty good choice, whereas Naive Bayes makes absolutely no sense. But Naive Bayes could determine sentiment or some other type of probability. Support vector machines work well for problems that are looking at finding a hard split between two pieces of data and it doesn't suffer from the curse of dimensionality nearly as much. So SVM's tend to be good for word problems where there's a lot of features. Neural Networks can solve problems ranging from classifications to driving a car. Ridge Regression is really just a simple trick to add onto a linear regression toolbelt and can find the mean of a curve. Hidden Markov models can follow musical scores, tag parts of speech and other system like applications.

Clustering is good at grouping data together without any sort of goal. This can be useful for analysis, or just to build a library and store data effectively. Filtering is well suited for overcoming the curse of dimensionality. We saw it used predominantly in the KNN chapter by reducing extracted pixels to features.

The thing that we didn't touch on in the book is that these algorithms are just a starting point. The important part to realize is that it doesn't matter what you pick it is what you are trying to solve that matters. That is why we cross-validate, we measure precision, recall and accuracy. Testing every step of the way guarantees that we at least approach better answers and check our work.

I encourage you to read more about machine learning and to think about applying tests to them. Most algorithms have them baked into them as well which is good, but for us mere humans to write code that learns over time we need to be checking our own work as well.

What's next for you?

This is just the beginning in your journey. Machine Learning is a field that is rapidly growing every single year. We are learning about how to build robotic self-driving cars using deep learning networks, and classify many things like health problems using Restricted Boltzmann Machines. The future is bright for Machine Learning, and now that you've read this book you are better suited to learn more about deeper subtopics like Reinforcement Learning, Deep Learning, Artificial Intelligence in general and more complicated Machine Learning algorithms.

There is a plethora of information out there for you to read more into. Great books, and video lectures.

A short list of books I would recommend are:

1. Sutton, Richard and Andrew Barto - Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)
2. Mitchell, Tom - Machine Learning
3. Russell, Stuart and Peter Norvig - Artificial Intelligence: A Modern Approach

4. Flach, Peter - Machine Learning: The Art and Science that Make Sense of Data
5. Segaran, Toby - Programming Collective Intelligence: Building Smart Web 2.0 Applications
6. Mackay, David - Information Theory, Inference, and Learning Algorithms

On top of that there are a massive amount of videos to check out online either through online courses, or YouTube. Along with that watching lectures on deep learning is rewarding. Geoffrey Hinton's lectures are a great place to start or anything done by Andrew Ng including his Coursera course.

Now that you know a bit more about Machine Learning you can go out and solve problems that are not black or white but instead involve a lot more shades of grey. Using a test driven approach as we have throughout the book will equip you to think about these problems from a scientific lens and to attempt to solve problems not by being true or false but instead having a level of accuracy. Machine Learning is a fascinating field because it allows you to take two divergent ideas like computer science which is theoretically sound and data which is practically noisy and zip them together in one beautiful relationship.