O'REILLY®

Thoughtful Machine Learning with Python

A TEST-DRIVEN APPROACH

Matthew Kirk

Thoughtful Machine Learning with Python

A Test-Driven Approach

Matthew Kirk

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Thoughtful Machine Learning with Python
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://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editor: Shannon Cutt
Production Editor: FILL IN PRODUCTION EDITOR
Copyeditor: FILL IN COPYEDITOR
Proofreader: FILL IN PROOFREADER

Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

November 2016: First Edition

Revision History for the First Edition
2016-11-14 First Early Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491924068* for release details.

978-1-491-92406-8

[FILL IN]

Table of Contents

Preface

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at https://github.com/oreillymedia/title_title.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Safari

Safari (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit *http://oreilly.com/safari*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at http://www.oreilly.com/catalog/<catalog page>.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at http://www.oreilly.com.

Find us on Facebook: http://facebook.com/oreilly

Follow us on Twitter: http://twitter.com/oreillymedia

Watch us on YouTube: http://www.youtube.com/oreillymedia

Acknowledgments

CHAPTER 1

Probably Approximately Correct Software

If you've ever flown on an airplane you have participated in one of the safest forms of travel in the world. The odds of being killed in an airplane are 1 in 29.4 Million [*http://www.statisticbrain.com/airplane-crash-statistics/*] , meaning that you could decide to become an airline pilot and throughout a 40 year career never once be in a crash. Those odds are staggering considering just how complex airplanes really are. But it didn't always used to be that way.

2014 was a bad year for aviation. The number of deaths in 2014 was 824 [*http://www.nola.com/business/index.ssf/2014/12/2014_was_actually_a_good_year.html*], including the Malaysia Air plane that went missing. In 1929 there were 257 casualties [*http://www.baaa-acro.com/result-histogram-fatalities-page/?from=1918&to=1929&type=8*]. This seems like we've become worse at aviation until you realize that in the US alone there are over 10 Million flights per year where as in 1929 there were substantially less flights. An estimate would be in the 50-100 thousand flight range. This means that the overall probability of being killed in a plane wreck from 1929 to 2014 has plummeted from 0.25% to 0.00824%.

Plane rides have changed over the years and so has software development. While in 1929 software development as we know it didn't exist over the course of 85 years we have built and failed many software projects.

Recent examples include software projects like healthcare.gov which is a disaster in terms of fiscal spending. Some estimate it costing around $634 Million dollars [*http://news.slashdot.org/story/13/10/10/132221/cost-of-healthcaregov-634-million-so-far*]. Even worse are when software projects have other disasterous bugs associated with them. In 2013 the NASDAQ shutdown due to a software glitch and were fined $10 million USD [*http://www.reuters.com/article/2013/08/22/us-nasdaq-halt-tapec-idUSBRE97L0V420130822*]. 2014 saw the heartbleed bug happen that made many sites using SSL vulnerable. CloudFlare as a result revoked over one hundred thousand

SSL certificates which they have said will cost them millions [*http://www.theguardian.com/technology/2014/apr/18/heartbleed-bug-will-cost-millions*].

Software and airplanes share one common thread: they're both complex and when they fail they fail catastrophically and publically. Airlines have been able to ensure safe travels and decrease the probability of airline disasters by over 96%. Unfortunately with the complexity of software only becoming higher as time goes on we cannot say the same of software. Catastrophic bugs are happening with regularity, and costing us billions of dollars in wasted money.

Why is it that airlines have become so safe and software so buggy?

Writing Software Right

Between 1929 and 2014 many things have changed with airplanes: they have become more complex, bigger, and faster. But with that growth also came more regulation from the FAA and international bodies as well as a culture of checklists with pilots.

While hardware has rapidly changed with computers and technology the software that runs it hasn't changed as much. We still use mostly procedural and object oriented code that doesn't take advantage of parallel computation as well as it could. But there have been good strides with coming up with guidelines around writing software as well as a culture of testing. These have led to the adoption of SOLID, and TDD. SOLID is a set of principles that guide us to write better code. And TDD is either Test Driven Design or Test Driven Development. We will talk about these two mental models as it relates to writing the right software as well as talk about refactoring which seems to be software centric.

SOLID

In the same ways that the FAA defines what an airline or airplane *should* do, SOLID tells us how software *should* be created. Violations happen occasionally with the FAA and they can range from disastorous to minute. The same is true with SOLID. These principles sometimes make a huge difference but most of the time are just guidelines. SOLID is a framework that helps design better object oriented code. This was introduced by Robert Martin under the name the Five Principles. The impetus was to write better code that is maintainable, understandable, and stable. Michael Feathers came up with the mnemonic device SOLID to remember them.

SOLID stands for:

- Single Responsibility Principle (SRP)
- Open / Closed Principle (OCP)
- Liskov Substitution Principle (LSP)

- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP).

Single Responsibility Principle

Single Responsibility Principle, or SRP, has become one of the most prevalent parts of writing good object oriented code. The reason is that single responsibility defines simple classes or objects. The same mentality can be applied to functional programming with pure functions. But the idea is all about simplicity. Have a piece of software do one thing and only one thing. A good example of a SRP violation is multi-tools. They do just about everything but unfortunately are only useful in a pinch.

Figure 1-1. A Multitool like this has too many responsibilities

Open / Closed Principle

Open / Closed Principle, or OCP, sometimes also called encapsulation is the principle that objects should be open for extending but not for modification. This can be shown in the case of a Counter object which has an internal count associated with it. The object has the methods increment and decrement. This object should not allow anybody to change the internal count unless it follows the defined api but it can be extended for instance to notify someone of a count change by an object like Notifier.

Liskov Substitution Principle

Liskov Substitution Principle, or LSP, states that any subtype should be easily substituted out from underneath a object tree without side effect. For instance a model car could be substituted for a real car.

Interface Segregation Principle

Interface Segregation Principle, or ISP, is the principle that having many client-specific interfaces is better than a general interface for all clients. This principle is about simplifying the interchange of data between entities. A good example would be separating garbage, compost, and recycling. Instead of having one big garbage can it is specific to the garbage type.

Dependency Inversion Principle

Dependency Inversion Principle, or DIP, is a principle that guides us to depend on abstractions not concretions. What this is saying is that we should build a layer or inheritance tree of objects. The example Robert Martin explains in his original paper is that we should have a KeyboardReader inherit from a general Reader object instead of being everything in one class. This also aligns well with what Arthur Riel said in *Object Oriented Design Heuristics* about avoiding *god classes*. While you could solder a wire directly from a guitar to an amplifier it most likely would be inefficient and not very good.

SOLID the framework has stood the test of time and has shown up in many books by Martin, Feathers as well as showing up in Sandi Metz's book *Practical Object-Oriented Design in Ruby* [*http://poodr.info*]. This framework is meant to be guideline but also remind us of the simple things so that when we're writing code we write the best we can. These guidelines help write software right architectually.

Testing or TDD

In the early days of aviation pilots didn't use checklists to test whether their airplane was ready for takeoff. In the book "The Right Stuff" by Tom Wolfe most of the original test pilots like Chuck Yaeger would go by feel and their own ability to manage the complexities of the craft. This also led to a quarter of test pilots being killed in action [cite].

Today, things are different. Before taking off, pilots go through a set of checks. Some of these checks can seem arduous like introducing yourself by name to the other crew people. But imagine if you find yourself in a tailspin and need to notify someone of a problem immediately. If you didn't know their name it'd be hard to communicate.

The same is true for good software. Having a set of systematic checks, that run regularly, to test whether our software is running properly or not is what makes software run and not break.

In the early days of software, most tests were done after writing the original software [cite]. This worked well with the style of project management common then. Similar to how airplanes are still built, software used to be designed first, written according to specs, and then tested before delivering to the customer. But as technology has a short shelf life this method of testing could take months or even years. This led to the Agile Manifesto as well as the culture of testing and TDD. This was spearheaded by many people including Kent Beck, Ward Cunningham, and others.

The idea of test driven development is simple, write a test to write down what you want to achieve, test to make sure the test fails first, write the code to fix the test, and then after it passes fix your code to fit in with the SOLID guidelines above. While many people argue that this adds time onto the development cycle it drastically reduces bug deficiencies in code and improves stability of code as it operates in production [cite microsoft research].

Airplanes with their low tolerance for failure operate in many ways the same way. Before a pilot flies the Boeing 787 they have spent x amount of hours in a flight simulator understanding and testing their knowledge of the plane. Before planes take off they are tested, and during the flight they are tested again. Modern software development is very much the same way. We test our knowledge by writing tests before as well as when something is deployed (by monitoring).

But this still leaves one problem which is the reality that not everything stays the same, writing a test doesn't make good code. David Heinemer Hanson in his viral presentation about test driven damage [cite] has pointed out some very good points that following TDD and SOLID blindly will yield complicated code. Most of his points have to do with needless complication due to extracting out every piece of code into different classes, or writing code to be testable and not readable. But I would argue that this is where the last factor in writing software right comes in: Refactoring.

Refactoring

Refactoring is one of the hardest to explain practices that programmers do, to people who are not programmers. Non programmers don't get to see what is underneath the surface. When you fly on a plane you are seeing 20% of what makes the plane fly. Underneath all of the pieces of aluminum and titanium are intricate electrical systems that power emergency lighting in case anything fails during flight, plumbing, engineered trusses to be light and also sturdy, plus too much to list here. In many ways explaining what goes into airplane is like explaining to someone that there's pipes under the sink of that beautiful faucet.

http://www.istockphoto.com/photo/modern-bathroom-sink-43303312?st=e997ef5 http://www.istockphoto.com/photo/plumbing-leak-45480444?st=4e40ced

Refactoring is taking the existing structure and making it better. It's taking a messy circuit breaker and cleaning it up so that when you look at it you know exactly what is going on. While airplanes are rigidly designed software is not. Things change rapidly in software. Many companies are continuously deploying software to a production environment. All of that feature development can sometimes cause a certain amount of *technical debt*.

Technical debt, also known as design debt or code debt, is a metaphor about poor system design that happens over time with software projects. The debilitating problem of technical debt is that it accrues interest and eventually becomes a blocker to any future feature development.

If you've been on a project long enough you will know the feeling of having fast releases in the beginning only to come to a stalwart towards the end. Technical debt in many cases is found through not writing tests, or not following the SOLID principles listed above.

Having technical debt isn't a bad thing, sometimes projects need to be pushed out earlier so business can increase but not paying down debt will eventually acrue enough interest to destroy a project. The way we get over this is by refactoring our code.

Refactoring is moving our code closer to SOLID, and a TDD codebase. It's cleaning up the existing code and making it easy for new developers to come in and work on the code that exists.

1. Follow the SOLID guidelines
    a. Single Responsibility Principle
    b. Open / Closed Principle
    c. Liskov Substitution Principle
    d. Interface Segregation Principle
    e. Dependency Inversion Principle
2. Implement TDD (Test Driven Development / Design)
3. Refactor your code to avoid a buildup of technical debt

The real question now is — what makes the software right?

Writing the Right Software

Writing the right software is much more tricky than writing software right. In his book *Specification by Example* Gojko Adzic determines the best approach is by writing specifications first and working with consumers directly. Then only after having a specification does one write the code to fit that spec. But this suffers from the problem of practice, sometimes the world isn't what we think it is. Our initial model of what we think is true many times isn't.

Webvan, for instance, failed miserably at building an online grocery business. They had almost $400 million invested in them and rapidly built infrastructure to support their most likely booming business. Unfortunately they were a flop because of the cost of shipping food as well as the mispredicted market of online grocery buying. By many means they were a success at writing software and building a business but the market just wasn't ready for them and they went bankrupt quickly. To this day a lot of the infrastructure that they built is used by Amazon.com for AmazonFresh.

> In theory, theory and practice are the same. In practice they are not
>
> —Albert Einstein

We are now at the point where theoretically we can write software correctly and it'll work but writing the right software is a much fuzzier problem. This is where Machine Learning really comes in.

Writing the right software with Machine Learning

In The Knowledge-Creating Company by Nonaka and Takeuchi, the authors outline what makes 1980s japanese companies so successful. Instead of a top down approach of solving the problem, they would learn over time. Their example of kneading bread and turning that into a breadmaker is a perfect example of iteration and is easily applied to software development.

But we can go further with machine learning.

What exactly is Machine Learning?

According to most definitions Machine Learning is a collection of algorithms, techniques, and tricks of the trade that allow machines to learn from data. Data being something represented in numerical format (matricies, vectors, etc).

To understand Machine Learning better though let's look at how it came into existence.

In the 1950's there was extensive research done on playing checkers. A lot of these models focused on playing the game better and coming up with optimal strategies. You could probably come up with a simple enough program to play checkers today

just by working backwards from a win and mapping out a decision tree and optimizing that way.

This was a very narrow and deductive way of reasoning. Effectively the agent had to be programmed. In most of these early programs there was no context or irrational behavior programmed in.

About 30 years later Machine Learning started to take off. Many of the same minds started working on problems involving spam filtering, classification, and general data analysis.

The shift here that is important is a move away from computerized deduction to computerized induction.

Deduction much like Sherlock Holmes involved using complex logic models to come to a conclusion. Induction involves taking data as being true and trying to fit a model to that data. This shift has created many great advances around finding good enough solutions to common problems.

The issue with inductive problems though is that you can only feed the algorithm data that **you** know about. Quantifying some things is exceptionally difficult. For instance how could you quantify how cuddly a kitten looks in an image?

In the last 10 years we have been witnessing a renaissance around Deep Learning which alleviates that problem. Instead of relying on data coded by humans algorithms like autoencoders have been able to find data points we couldn't quantify before.

This all sounds amazing but with all this power comes an exceptionally high cost and responsibility.

The High Interest Credit Card Debt of Machine Learning

Recently in a paper published by Google entitled Machine Learning: The High Interest Credit Card of Technical Debt Sculley et.al explained that Machine Learning projects suffer from the same technical debt issues outlined above plus more.

They listed that machine learning projects are inherently complex, have vague boundaries, rely heavily on data dependencies, suffer from system level spaghetti code, and can radically change due to changes in the outside world. Their argument is that these are specifically related to machine learning projects and for the most part they are.

Instead of going through these one by one I thought it would be more interesting to tie back to our original discussion above about SOLID and TDD as well as Refactoring and how it relates to Machine Learning code.

Table 1-1. High Interest Credit Card Debt Of Machine Learning

Machine Learning Problem	Manifests as	SOLID Volation
Entanglement	Changing one factor changes everything	SRP
Hidden Feedback Loops	Having built in hidden features in model	OCP
Undeclared Consumers / Visibility Debt		ISP
Unstable Data Dependencies	Volatile data	ISP
Underutilized Data Dependencies	Unused dimensions	LSP
Correction Cascade		*
Glue Code	Writing code that does everything	SRP
Pipeline Jungles	Sending data through complex workflow	DIP
Experimental Paths	Dead paths that go nowhere	DIP
Configuration Debt	Using old configurations for new data	*
Fixed Thresholds in a Dynamic World	Not being flexible to changes in correlations	Correlations Change

SOLID applied to Machine Learning

SOLID as you remember is just a guideline reminding us to follow certain goals when writing object oriented code. Many machine learning algorithms are inherently not object oriented. They are functional, mathematical, and use lots of statistics but that doesn't have to be the case. Instead of thinking of things in purely functional terms we can strive to use objects around each row vector and matrix of data.

Single Responsibility Principle

In machine learning code one of the biggest challenges for people to realize is that the code and the data are dependent on each other. Without the data the machine learning algorithm is worthless, and without the machine learning algorithm we wouldn't know what to do with the data. So by definition they are tightly intertwined and coupled. This is probably one of the biggest reasons that Machine Learning projects fail is this tightly coupled dependency.

This is manifested as a couple problems in Machine Learning code: entanglement and glue code.

Entanglement is sometimes called the principle of Changing Anything Changes Everything or CACE. The easiest example to think of with entanglement is probabilities. If you remove one probability out of a distribution then all the rest have to adjust. This is a violation of SRP.

A few possible mitigation strategies include: isolating models, analyzing dimensional dependencies [H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, D. Golovin, S. Chikkerur, D. Liu, M. Wattenberg, A. M. Hrafnkelsson, T. Boulos, and J. Kubica. Ad click prediction: a view from the trenches.

In The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013, 2013.] , and regularization techniques [A. Lavoie, M. E. Otey, N. Ratliff, and D. Sculley. History dependent domain adaptation. In Domain Adaptation Workshop at NIPS '11, 2011.]

We will return to this problem when we look at bayesian models more, or probability models.

Whether machine learning researchers want to admit it or not, many times the actual machine learning algorithms themselves are quite simple. A lot of the code that surrounds that is what makes up the girth of the project. Depending on what library you use, whether it be graphlab, matlab, scikit-learn, R, they all have their own implementation of vectors and matrices which is what most of machine learning comes down to.

Open / Closed Principle

OCP as you remember is about opening classes for extension but not modification. A few ways this manifests in machine learning code are the problem of Changing Anything Changes Everything. This can be manifested in any software project but in Machine Learning projects they are seen a lot as Hidden Feedback Loops.

A good example of a hidden feedback loop is *predictive policing*. Over the last few years there are many researchers showing that machine learning algorithms can be applied to determine where crimes will occur. Preliminary results have shown that these algorithms work exceptionally well. But unfortunately there is a dark side to them as well.

While these algorithms can show where crimes will happen, what will naturally occur is the police will start policing those areas more and finding more crimes there and as a result will self reinforce the algorithm. This could also be called confirmation bias, or the bias of confirming our pre-conceived notion. This of course also has the downside of enforcing systematic descrimination against certain demographics or neighborhoods.

While hidden feedback loops are hard to detect they should be watched for with a keen eye, and taken out.

Liskov Substitution Principle

Not a lot of people talk about the Liskov Substitution Principle anymore cause many programmers are advocating for composition over inheritance these days. But in the machine learning world the Liskov Substitution Principle is violated a lot. Many times we are given datasets that we don't have all the answers for yet. Sometimes these data sets are thousands of dimensions wide.

Running algorithms against those datasets can actually cause a violation of the Liskov Substitution Principle. One common manifestation in machine learning code is underutilized data dependencies.

Many times we are given datasets that include thousands of dimensions which can sometimes yield pertinent information and something not. Sometimes our models will take all dimensions and use one infrequently. So for instance in classifying mushrooms to be either poisonous or edible information like odor can be a big indicator while ring number isn't.

The ring number has low granularity and can only be zero, one, or two and really doesn't add much to our model of classifying mushrooms. So that information could be trimmed out of our model and wouldn't cause too much degradation in performance.

You might be thinking why this is related with Liskov Substitution Principle and the reason is if we can use only the smallest set of datapoints (or features) we have built the best model possible. This also aligns well with Ockham's Razor which states that the simplest solution is the best one.

Interface Segregation Principle

Interface Segregation Principle or ISP is the notion that a client-specific interface is better than a general purpose one. In machine learning projects many times this can be hard to enforce because of the tight coupling of data to the code. In machine learning code, ISP is usually violated by two types of problems: 1) *visibility debt*, and 2) *unstable data*.

Take for instance the case where a company has a reporting database that is used to collect information about sales, shipping data, and other pieces of crucial information. This is all managed through some sort of project which gets the data into this database. Off of that database the consumer that is defined is a machine learning project which takes previous sales data to predict the sales for the future. Then one day as cleanup someone renames a table that used to be called something very confusing to something much more useful.

All hell breaks loose and people are wondering what happened. What ended up happening is that the machine learning project wasn't the only consumer of the data, there were 6 Access databases that were attached to this as well. The fact that there were that many undeclared consumers is in itself a piece of debt for a machine learning project.

This type of debt is called Visibility Debt and while for the most part it doesn't affect a project's stability at a certain point in time sometimes as features are built it will hold everything back.

Data as we know is dependent on the code that finds induction off of it so building a stable project requires having stable data. Many times this just isn't the case. Take for instance the stock market, daily the stock market is fluctuating and moving around. In the morning a momentum type stock might be valuable while in the evening a reversal might happen.

This ends up violating Interface Segregation because we are looking at the general data stream instead of a specific one to the client.

This can make portfolio trading algorithms very difficult to build. One common trick is to build some sort of exponentially weighting scheme around data but more importantly to version data streams.

This versioned scheme serves as a way of limiting the volatility of a model's predictions and can be a viable way of mitigating the effects of a vacillating model.

Dependency Inversion Principle

The Dependency Inversion Principle is about limiting our concretions of data and making code more flexible for future changes. In a machine learning project we see concretions happen in two specific ways: Pipeline Jungles, and Experimental Paths.

Pipeline Jungles are common in data driven projects and are almost a form of glue code. This is the amalgamation of data being prepared and moved around. In cases this code is tying everything together so the model can work with the prepared data. Unfortunately though over time these start to grow complicated and unusable.

As you know machine learning code requires the software to run it, and also the data. They are intertwined and inseparable. That means that sometimes we have to test things out in what is production. Sometimes tests on our machines give us false hope and we need to try something out. Those experimentations add up over time and end up polluting our workspace. The best way of reducing the debt associated with that is to introduce Tombstoning which is an old technique from C.

Tombstones are a method of marking something as ready to be deleted. If the method is called in production it will log an event to a logfile that can be utilized to sweep the codebase at some point in time.

For those of you who have studied about garbage collection you most likely have heard of this method as Mark and Sweep. Basically you mark an object as ready to be deleted and occasionally you sweep the marked objects.

Machine Learning code is complex but not impossible

At times, machine learning code is difficult to write and understand, but it is far from impossible. Remember the flight analogy we began with, and utilze the SOLID guide-

lines as your "pre-flight" checklist for writing successful machine learning code — while complex, it doesn't have to be complicated.

In the same vein, you can compare machine learning code to flying a spaceship — it's certainly been done before, but it's still bleeding edge. With the SOLID checklist model, we can launch our code effectively using TDD and refactoring. IN essence, writing successful machine learning code comes down to be disciplined enough to follow the principles of design we've laid out in this chapter, and writing tests to support your code-based hypotheses. Another critical element in writing effective code is being flexible and adapting to the changes it will encounter in the real world.

TDD: Scientific Method 2.0

Every true scientist is a dreamer and a skeptic. Daring to put a man on the moon is audacious but through systematic research and development we have accomplished that and much more. The same is true with Machine Learning code. Some of the applications are fascinating but also hard to pull off.

The secret to pulling it off is to have the checklist of SOLID for ML above and using the tools of TDD and Refactoring to get us there.

Test Driven Development is more of a style of problem solving not a mandate from above. What testing gives us is a feedback loop which we can use to work through tough problems. As scientists would assert that they need to first hypothesize, test, and theorize, we can assert that as a TDD practitioner red (the tests fail), green (the tests pass), refactor is just as viable.

This book will delve heavily into Test Driven Development applied to machine learning but also applying SOLID principles to machine learning as well. The goal of course being able to refactor our way to building a stable, scalable, and easy to use model.

Refactoring our way to knowledge

Refactoring is the ability to edit one's work and to rethink what was once stated. Throughout the book we will talk about refactorings of common machine learning pitfalls as it applies to algorithms.

The Plan for the book

This book will cover a lot of ground with machine learning but by the end you should have a better grasp of how to write machine learning code as well. How to deploy to a production environment and operate at scale. Machine Learning is a fascinating field that can achieve much but without discipline, checklists, and guidelines many machine learning projects are doomed to fail.

Throughout the book we will tie back to the original principles in this chapter by talking about SOLID principles, testing our code (using various testing means), and refactoring, as way to continually learn from and improve the performance of our code.

Every chapter will explain the python packages we utilize as well as a general testing plan for the chapter. While Machine Learning code isn't testable in a 1-to-1 case, it ends up being something that we can write tests to help our knowledge of the problem.

CHAPTER 2

A Quick Introduction to Machine Learning

You've picked up this book because you're interested in machine learning. While you probably have an idea of what machine learning is, it's a subject that is often defined in a somewhat vague way. In this quick introduction, we'll go over what exactly machine learning is, as well as a general framework for thinking about machine learning algorithms.

What Is Machine Learning?

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

Machine learning is a type of artificial intelligence whereby an algorithm or method will extract patterns out of data. Generally speaking, there are a few problems machine learning tackles; these are listed in Table 2-1 and described in the subsections that follow.

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
Maximizing rewards over time	Reinforcement learnig

Supervised Learning

Supervised learning, or function approximation, is simply fitting data to a function of any variety. For instance, given the noisy data shown in Figure 2-1, you can fit a line that generally approximates it.

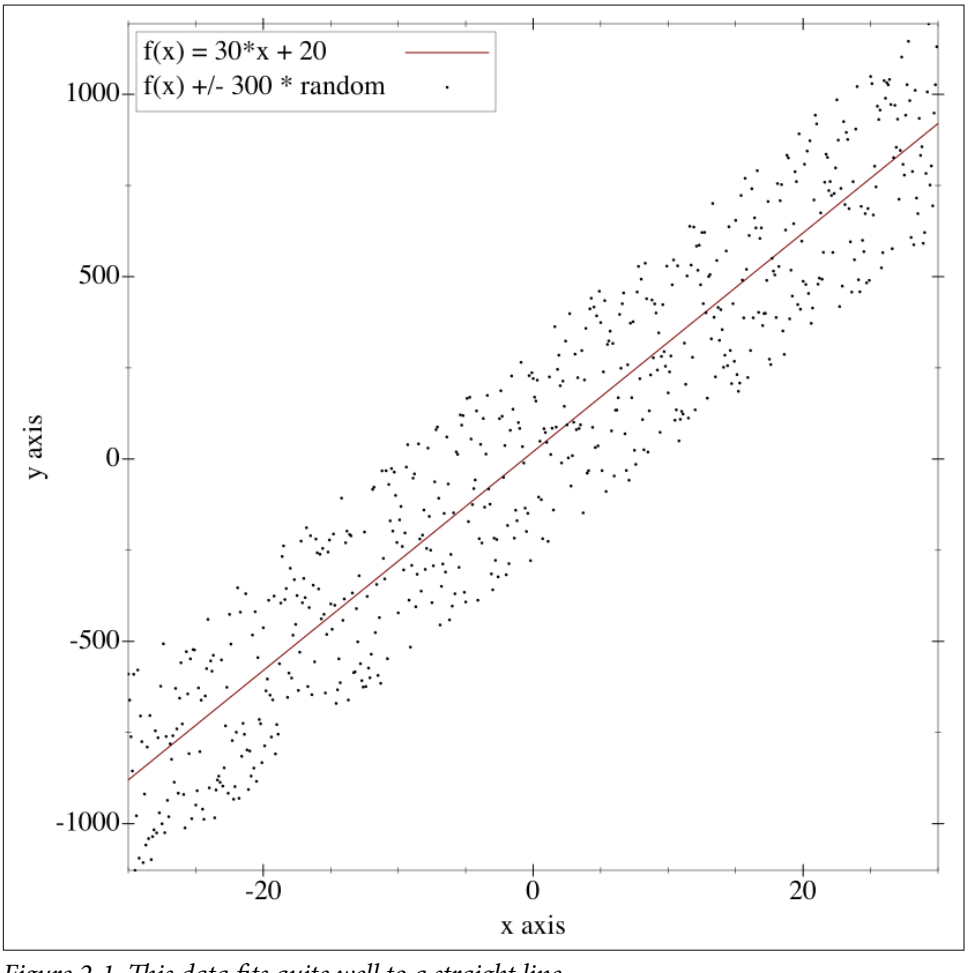

Figure 2-1. This data fits quite well to a straight line

Unsupervised Learning

Unsupervised learning involves figuring out what makes the data special. For instance, if we were given many data points, we could group them by similarity (Figure 2-2), or perhaps determine which variables are better than others.

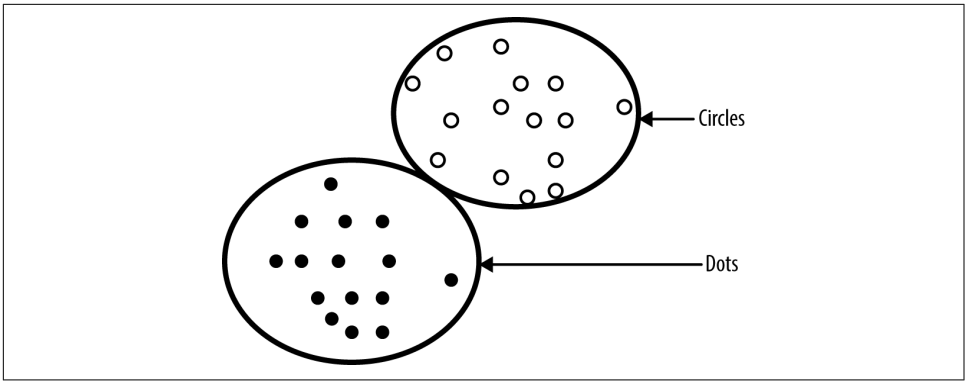

Figure 2-2. Two clusters grouped by similarity

Reinforcement Learning

Reinforcement learning involves figuring out how to play a multistage game with rewards and payoffs. Think of it as the algorithms that optimize the 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 final 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?

What makes machine learning unique is its ability to optimally figure things out. But each machine learning algorithm has quirks and trade-offs. Some do better than others. This book covers quite a few algorithms, so Table 2-2 provides a matrix to help you navigate them and determine how useful each will be to you.

Table 2-2. Machine learning algorithm matrix

Algorithm	Type	Class	Restriction bias	Preference bias
K Nearest Neighbors	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
Naive 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

Decision Trees / Random Forests	Supervised learning	Tree	Becomes less useful on problems with low covariance	Prefers problems with categorical data
Support Vector Machines	Supervised Learning	Decision Boundary	Works where there is a definite distinction between two classifications	Prefers binary classification problems
Neural Networks	Supervised Learning	Nonlinear functional approximation	Little restriction bias	Prefers binary inputs
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)
Feature Selection	Unsupervised	Matrix Factorization	No restrictions	Depending on algorithm can prefer data with high mutual information
Feature Transformation	Unsupervised	Matrix Factorization	Must be a non-degenerate matrix	Will work much better on matricies that don't have inversion issues
Bagging	Meta-Heuristic	Meta-Heuristic	Will work on just about anything	Prefers data that isn't highly variable

Refer to this matrix throughout the book to understand how these algorithms relate to one another.

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

Before we get started, you will need to install Python, which you can do at *https://www.python.org/downloads/*. This book was tested using Python 2.7.12, but most likely it will work with Python 3.x as well. All of those changes will be annotated in the coding resources, which are available on GitHub.

Mathematical Notation Used Throughout the Book

This book uses mathematics to solve problems, but all of the examples are programmer-centric. Throughout the book, I'll use the mathematical notations shown in Table 2-3.

Table 2-3. Mathematical notations used in this book's examples

Symbol	How do you say it?	What does it do?
$\sum_{i=0}^{n} x_i$	The sum of all x's from x_0 to x_n	This is the same thing as $x_0 + x_1 + \cdots + x_n$
$\|x\|$	The absolute value of x	This takes any value of x and makes it positive. So $\|-x\| = \|x\|$

Symbol	How do you say it?	What does it do?
$\sqrt{4}$	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 plane 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 occurrences.
$P(A\|B)\$]$	Probability of A given B	This is the probability of A and B divided by the probability of B.
{1, 2, 3} ∩ {1}	The intersection of set one and two	This turns into a set {1}.
{1, 2, 3} ∪ {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 $m\dot{a} = b$.
min $f(x)$	Minimize $f(x)$	This is an objective function to minimize the function $f(x)$.
X^T	Transpose of the matrix X	Take all elements of the matrix and switch the row with the column. Conclusion ~~~~ This isn't an exhaustive introduction to machine learning, but that's OK. There's always going to be a lot for us all to learn when it comes to this complex subject, but for the remainder of this book, this should serve us well in approaching these problems.

CHAPTER 3

K-Nearest Neighbors

Have you ever bought a house before? If you're like a lot of people around the world, the joy of owning your own home is exciting. But the process of finding, and buying a house can be stressful. Whether we're in a economic boom or recession everybody wants to get the best house for most reasonable price.

But how would you go about buying a house? How do you appraise a house? How does a company like Zillow come up with their Zestimates? We'll spend most of this chapter answering questions related to this fundamental concept, distance based approximations.

First we'll talk about how we can estimate a house value. Then we'll talk about how to classify houses into categories such as "Buy", "Hold", "Sell". At that point we'll talk about a general algorithm K-Nearest Neighbors and how it can be used to solve problems such as this. We'll break it down into a few subheadings of what makes something near, as well as what a neighborhood really is (what is the optimal K for something).

How do you determine whether you want to buy a house?

This question has plagued many of us for a long time. If you are going out to buy a house, or calculate whether it's better to rent, you are most likely trying to answer this question implicitly. Home appraisals are a tricky subject, and notorious for drift with calculations. For instance on Zillow's website they explain that their famous Zestimate(tm) is flawed [cite]. They state that based on where you are looking the value might drift by a localized amount.

Localization is really key with houses. Seattle might have a different demand curve than San Francisco. Matter of fact that makes complete sense if you know housing!

The question as to whether to buy or not, comes down to value amortized over the course of how long you're living there. But how do you come up with a value?

How valuable is that house?

"Things are worth as much as someone is willing to pay" - Old Saying

Valuing a house is tough business. Even if we were able to come up with a model with many endogenous variables that make a huge difference it doesn't cover up the fact that buying a house is subjective and sometimes is victim of a bidding war. These are almost impossible to predict. You're more than welcome to use this to value houses but there will be errors that take years of experience to overcome.

A house is worth as much as it'll sell for. The answer to how valuable a house is, at it's core is simple but difficult to estimate. Due to inelastic supply, or that houses are all fairly unique, prices have a tendency to be irratic with home sales. Sometimes you just love your home and will pay a premium for it.

But let's just say that the house is worth what someone will pay for it. And that is a function based on a bag of attributes associated with houses. We might determine that a good approach to estimating house values would be:

Equation 3-1. House Value

$$HouseValue = f(Space, LandSize, Rooms, Bathrooms, \cdots)$$

This model could be found through regression (which we'll cover in Chapter [FIXME: Dont know what chapter regression will be]) or other approximation algorithms, but this is missing a major component of real estate: "Location, Location, Location!". To overcome this we can come up with something called a Hedonic regression.

Hedonic Regression

Hedonic Regression in real life. You probably already know of a highly used hedonic regression that is published: the CPI index. This is used as a way of decomposing baskets of items that individuals buy on a general basis to come up with an index for inflation.

Economics is a dismal science because we're trying to approximate rational behaviors. Unfortunately we are predictably irrational (Call to Dan Airely). But a good algorithm for valuing houses which is similar to what home appraisers use is called Hedonic regression.

The general idea with hard to value items like houses which don't have a highly liquid market and suffer from subjectivity is that there are externalities that we can't really estimate directly. For instance how would you estimate pollution, noise, neighbors who are jerks?

To overcome this hedonic regression takes a different approach than general regression. Instead of focusing on fitting a curve to a bag of attributes it focuses on the components of a house. For instance the hedonic method allows you to find out how much a bedroom costs (on average).

Take a look at the following housing data related with number of bedrooms. From here we can fit a naive approximation of value to bedroom number, to come up with an estimate of cost per bedroom.

Table 3-1. House Price by Bedrooms

Price (in $1,000)	Bedrooms
$899	4
$399	3
$749	3
$649	3

This is extremely useful for valuing houses because as consumers we can utilize this to focus on what matters to us and decompose houses into whether it's overpriced cause of bedroom numbers or the fact that it's right next to a park.

This gets us to the next improvement which is location. Even with hedonic regression, we suffer from the problem of location. A bedroom in SoHo in London, England is probably more expensive than a bedroom in Mumbai, India. So for that we need to focus on the neighborhood.

For additional reading on Hedonic regression I suggest you take a look at the Consumer Price Index which is a Hedonic regression. [*http://www.bls.gov/cpi/*]

What is a neighborhood?

The value of a house is in many times caused by it's neighborhood. For instance in Seattle, an apartment in Capitol Hill is more expensive than one in Lake City. Generally speaking the cost of commuting is worth half of your hourly wage plus maintenance and gas [Van Ommeren, Jos, Van den Berg, Gerard, and Gorter, C. "Estimating

the Marginal Willingness to Pay for Commuting." Journal of Regional Science 40 (2000): 541–563.], so a neighborhood closer to the economic center is more valuable.

But how would we focus only on the neighborhood?

Theoretically we could come up with an elegant solution using something like an exponential decay function that weights closer houses higher and further houses lower. Or we could come up with something static that works exceptionally well: K-Nearest Neighbors.

K-Nearest Neighbors

What if we were to come up with a solution that is inelegant but works just as well? Say we were to assert that we only look at an arbitrary amount of houses near to a similar house we're looking at. Would that work just as well?

Surprisingly yes. This is the K-Nearest Neighbor (KNN) solution that performs exceptionally well. It takes two forms either as a regression, where we want a value, or a classification.

To apply KNN to our problem of house values we would just have to find the nearest K neighbors.

The K-Nearest Neighbor algorithm was originally introduced by Drs. Evelyn Fix, and J.L. Hodges Jr, PhD, in an unpublished technical report written for the U.S. Air Force School of Aviation Medicine. Fix and Hodges' original research focused on splitting up classification problems into a few subproblems:

Subproblems

- Distributions F and G are completely known.
- Distributions F and G are completely known except for a few parameters.
- F and G are unknown, except possibly for the existence of densities.

Fix and Hodges pointed out that if you know the distributions of two classifications or you know the distribution minus some parameters, you can easily back out useful solutions. Therefore, they focused their work on the more difficult case of finding classifications among distributions that are unknown. What they came up with laid the groundwork for the KNN algorithm.

Though now we get to a few more questions:

- What are neighbors, what makes them near?
- How do we pick the arbitrary number of neighbors: K?
- What do we do with the neighbors afterwards?

Mr K's Nearest Neighborhood

We all implicitly know what a neighborhood is. Whether you live in the woods or a row of brownstones, we all live in a neighborhood of sorts. A neighborhood for lack of a better term could just be called a cluster of houses (we'll get to clustering later).

A cluster at this point could be just thought of a tight grouping of houses or items in n-dimensions. But what denotes a "tight grouping"? Since you've most likely taken a geometry class sometime in your life you're probably thinking of the pythagorean theorem or something similar but things aren't quite that simple. Distances are a class of functions that can be much more complex.

Distances

> As the crow flies
>
> —Old Saying

Geometry class taught us that if you sum the square of two sides of a triangle and take it's square root you'll have the side of the hypotenuse or the third side. This as we all know is the pythagorean theorem, but distances can be much more complicated. Distances can take many different forms but generally there are geometrical, computational, and statistical distances which we'll discuss in this section.

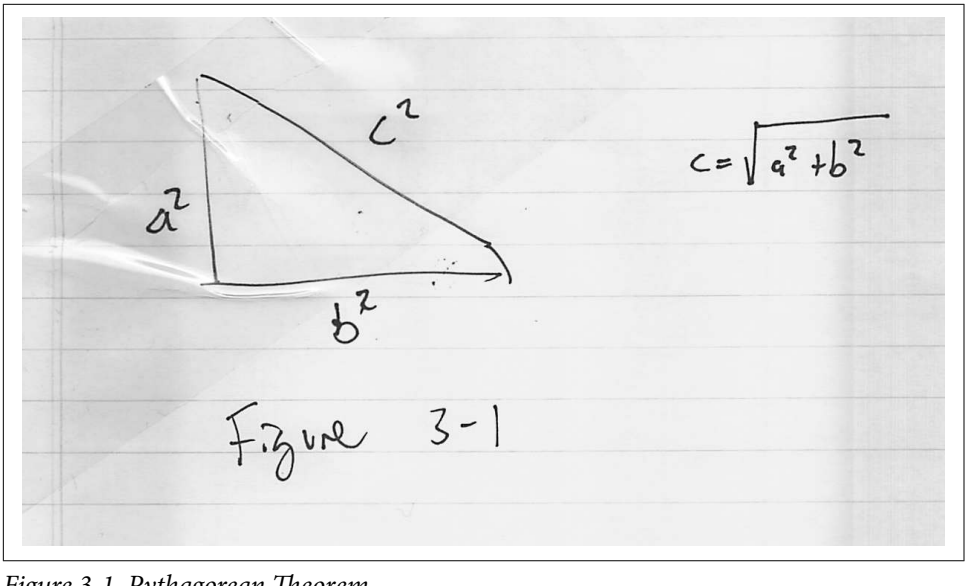

Figure 3-1. Pythagorean Theorem

Triangle Inequality

Thinking about that triangle above something is interesting which is that the distance on the hypotenuse is always less than the distance of each side added up individually.

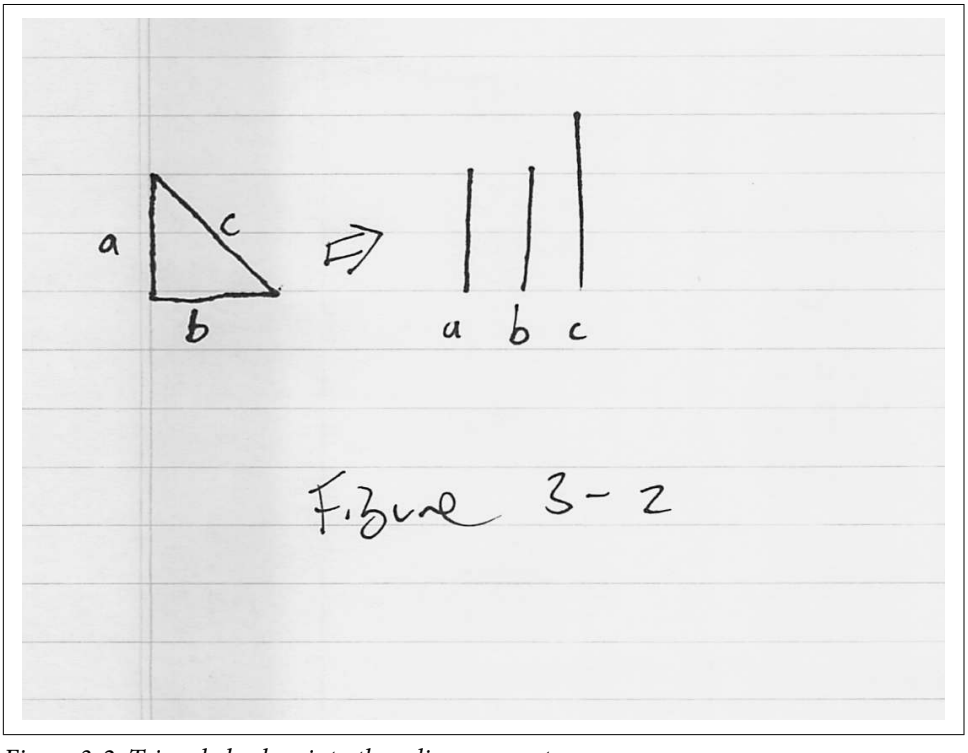

Figure 3-2. Triangle broken into three line segments

Stated mathematically: $||x|| + ||y|| \leq ||x|| + ||y||$. This inequality is important for finding a distance function. For if the triangle inequality didn't hold then what would happen is distances would become slightly distorted and as you measure distance between points in a euclidean space.

Geometrical Distance

The most intuitive distance functions are geometrical. Intuitively we can measure how far something is from one point to another. We already know about the pythagorean theorem but there are a countably infinite amount of possibilities, that satisfy the triangle inequality.

Stated mathematically we can take the pythagorean theorem and build what is called the euclidean distance which is denoted as: $d(x, y) = \sqrt{\Sigma_{i=0}^{n}(x_i - y_i)^2}$.

As you can see this is similar to the pythagorean theorem except it includes a sum. Though mathematics gives us even greater ability to build distances by using something called a Minkowski Distance: $d_p(x, y) = (\Sigma_{i=0}^{n} \left| x_i - y_i \right|^p)^{\frac{1}{p}}$. This p can be any integer and still satisfy the triangle inequality.

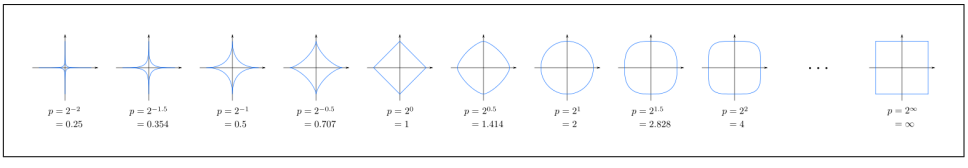

Figure 3-3. Minkowski distances as n increases (Wikimedia image)

Cosine Similarity

One last geometrical distance is called Cosine Similarity or Cosine Distance. The beauty of this distance is it's sheer speed at calculating distances between sparse vectors. For instance if we had 1000 attributes collected about houses and 300 of these were mutually exclusive (meaning that one house had them but the others don't) then we would only need to include 700 dimensions in the calculation.

Visually this measures the inner product space between two vectors and presents us with cosine as a measure. It's function is $d(x, y) = \frac{x\dot{y}}{||x||||y||}$ where $||x||$ denotes the euclidean distance we discussed above.

Geometrical distances are generally what we want. When we talk about houses we want a geometrical distance. But there are other spaces which are just as valuable: computational or discrete, as well as stastical distances.

Computational Distances

Imagine you want to measure how far it is from one part of the city to another. One way of doing this would be to utilize coordinates (longitude, latitude) and calculate a euclidean distance. Let's say you're at St Edwards Park in Kenmore, WA (47.7329290, -122.2571466) and want to meet someone at Vivace Espresso on Capitol Hill, Seattle, WA (47.6216650, -122.3213002).

Using the euclidean distance we would calculate $\sqrt{(47.73 - 47.62)^2 + (-122.26 + 122.32)^2} \approx 0.13$

This is obviously a small result as it's in degrees of latitude and longitude. To convert this into miles would multiply it by 69.055 which yields approximately 8.9 miles (14.32 kilometers). Unfortunately this is way off! The actual distance is 14.2 miles (22.9 kilometers). Why are things so far off?

Note that 69.055 is actually an approximation of latitude degrees to miles. Earth is an ellipsoid and therefore it actually depends on where you are in the world to calculate distances. But for such a short distance it's good enough

If I had the ability to lift off from St Edwards Park and fly to Vivace then yes it'd be shorter but if I were to walk or drive I'd have to drive around Lake Washington.

This gets us to the motivation behind computational distances. If you were to drive from St Edwards Park to Vivace then you'd have to follow the constraints of a road.

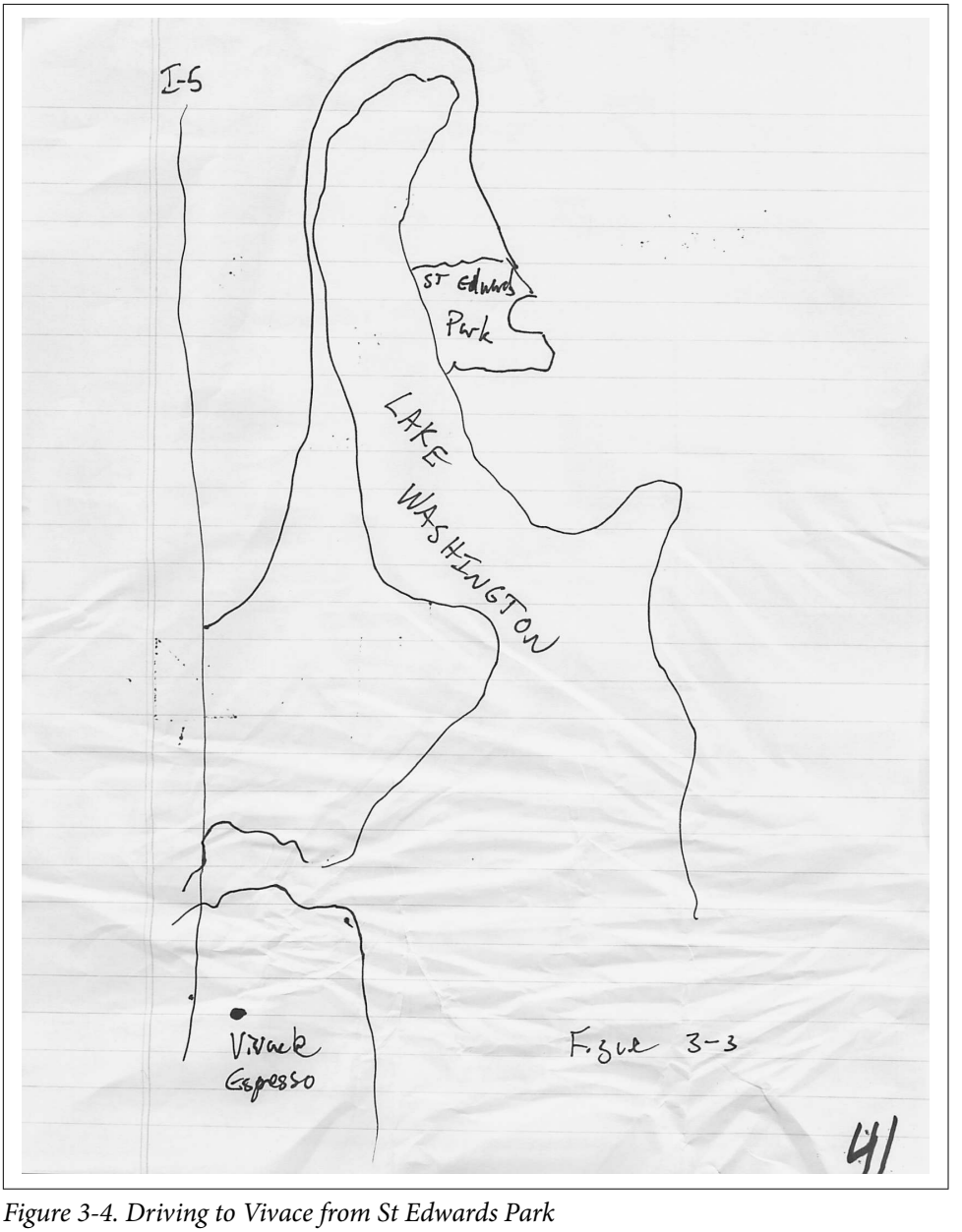

Figure 3-4. Driving to Vivace from St Edwards Park

Manhattan Distance

This gets us into what is called the Taxi-Cab distance or Manhattan Distance.

Equation 3-2. Manhattan Distance

$$\Sigma i = 0^{n} |x_i - y_i|$$

Note that there is no ability to travel outside of bounds. So imagine that your metric space is a grid of graphing paper and you are only allowed to draw on boxes.

The Manhattan Distance can be used for problems such as traversal of a graph, as well as discrete optimization problems where you are constrained by edges. With our housing example most likely you would want to measure the value of houses that are close by driving not by flying. Otherwise you might include houses in your search that are across a barrier like a lake, or a mountain!

Levenshtein Distance

Another distance that is commonly used in natural language processing is the Levenshtein distance. An analogy of how Levenshtein distance works is by changing one neighborhood to make an exact copy of another. The number of steps to make that happen is the distance. Usually this is applied with strings of characters to determine how many deletions, additions, or substitutions the strings require to be equal.

This can be quite useful for determining how similar neighborhoods are as well as strings.

The formula for this is a bit more complicated as it is a recursive function, so instead of looking at the math we'll just write python for this:

```
def lev(a, b):
  if not a: return len(b)
  if not b: return len(a)
  return min(lev(a[1:], b[1:])+(a[0] != b[0]), lev(a[1:], b)+1, lev(a, b[1:])+1)
```

This is an extremely slow algorithm and I'm only putting it here for understanding not to actually implement this. If you'd like to implement Levenshtein then you will need to use Dynamic Programming to have good performance.

Statistical Distances

Lastly there's a third class of distances which I call statistical distances. In statistics we're taught that to measure volatility or variance we take pairs of datapoints and measure the squared difference. This gives us an idea of how dispersed the popula-

tion is. This can actually be used inside of a distance as well which is called the Mahalanobis Distance.

Imagine for a minute that you want to measure distance in an affluent neighborhood that is right on the water. People love living on the water and the closer you are to the source the higher the value. But with our distances above whether computational or geometrical we would have a bad approximation of this particular neighborhood because things are primarily spherical in nature.

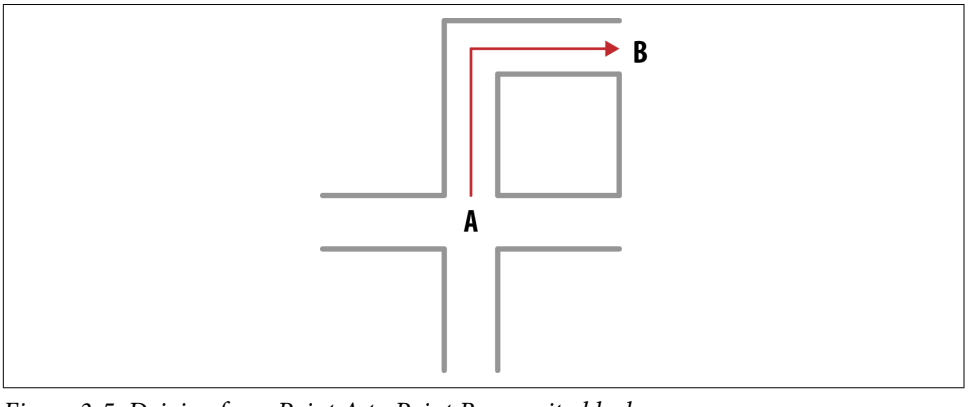

Figure 3-5. Driving from Point A to Point B on a city block

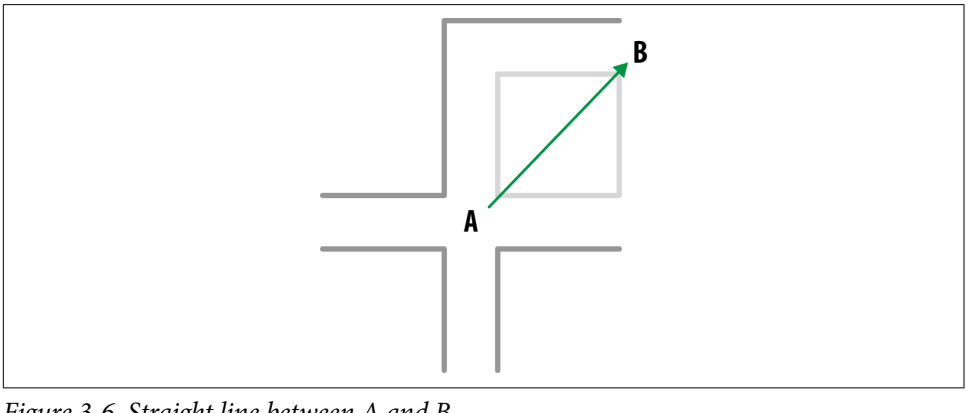

Figure 3-6. Straight line between A and B

This seems like a bad approach for this neighborhood because it is not spherical in nature. If we were to use euclidean distances we'd be measure values of houses not on the beach. If we were to use manhattan distances we'd only look at houses close by the road.

Mahalanobis Distance

Another approach is using the Mahalanobis Distance. This takes into consideration some other factors which are statistical: $d(x, y) = \sqrt{\sum_{i=1}^{n} \frac{(x_i - y_i)^2}{s_i^2}}$

What this effectively does is give more stretch to the grouping of items:

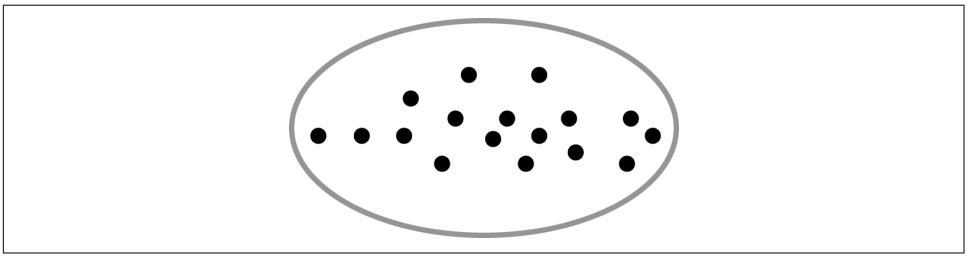

Figure 3-7. Mahalanobis Distance

Jaccard Distance

Yet another distance metric is called the Jaccard distance. And this takes into consideration the population of overlap. For instance if the number of attributes for a house match another then they would be overlapping and therefore close in distance where as if the houses had diverging attributes they wouldn't match. This is primarily used as a way of determining quickly how similar text is by counting up the frequencies of letters in a string and then counting the characters that are not the same across both. It's forumla is $J(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}$.

This finishes up a primer on distances. Now that we know how to measure what is close and what is far how do we go about building a grouping or neighborhood. How many houses should be in the neighborhood?

Curse of Dimensionality

Before we continue on there's a serious concern with using distances for anything and that is called the Curse of Dimensionality. Effectively when we model high dimension spaces our approximations of distance become less reliable. In practice it is important to realize that finding features of datasets is essential to making a resilient model. We will talk about feature engineering in a different chapter but for now be cognizant of the problem.

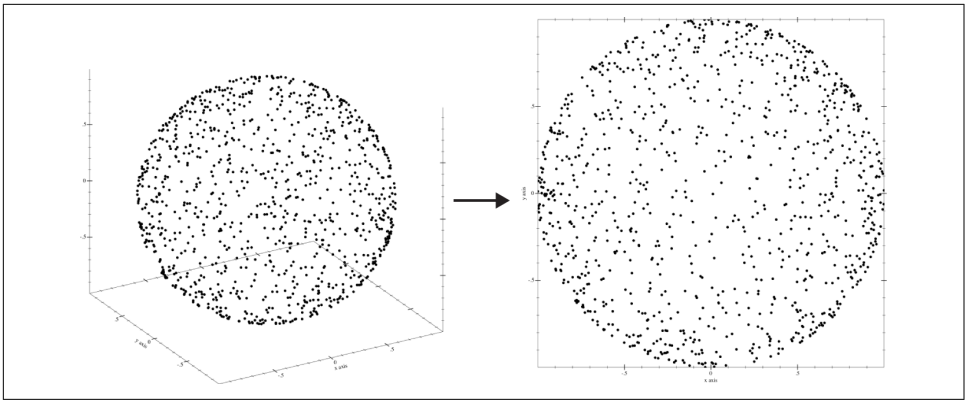

Figure 3-8. Curse of dimensionality

A visual way of thinking about it is put above. When we put random dots on a unit sphere and measure the distance from the origin (0,0,0) we find that the distance is always 1. But if we were to project that onto a 2d space the distance would be less than or equal to 1. This same truth holds when we expand the dimensions. For instance if we expanded our set from 3 dimensions to 4 then it would be greater than or equal to 1. This inability to center in on a consistent distance is what breaks distance based models because all of the data points become chaotic and move away from each other.

How do we pick K?

Picking the number of houses to put into this model is a NP problem, meaning it's easy to verify but hard to calculate before hand. At this point we know how we want to group things, but just don't know how many items to put into our neighborhood. There are a few approaches to determining an optimal K, each with their own set of downsides.

Approaches

- Guessing
- Using a heuristic
- Optimizing using an algorithm

Guessing K

Guessing is always a good solution. Many times when we are approaching a problem we have domain knowledge of the problem. Whether we are an expert or not we know about the problem enough to know what is a neighborhood. My neighborhood

where I live for instance is roughly 12 houses. If I wanted to expand I could set my K to 30 for a more flattened out approximation.

Heuristics for Picking K

1. Use coprime class and K combinations
2. Choose a K that is greater or equal to the number of classes plus one.
3. Choose a K that is low enough to avoid noise.

Use coprime class and K combinations

Picking coprime numbers of classes and K will ensure fewer ties. Coprime numbers are simply two numbers that don't share any common divisors except for 1. So, for instance, 4 and 9 are coprime while 3 and 9 are not. Imagine you have two classes, good and bad. If we were to pick a K of 6, which is even, then we might end up having ties. Graphically it looks like Figure 3-9.

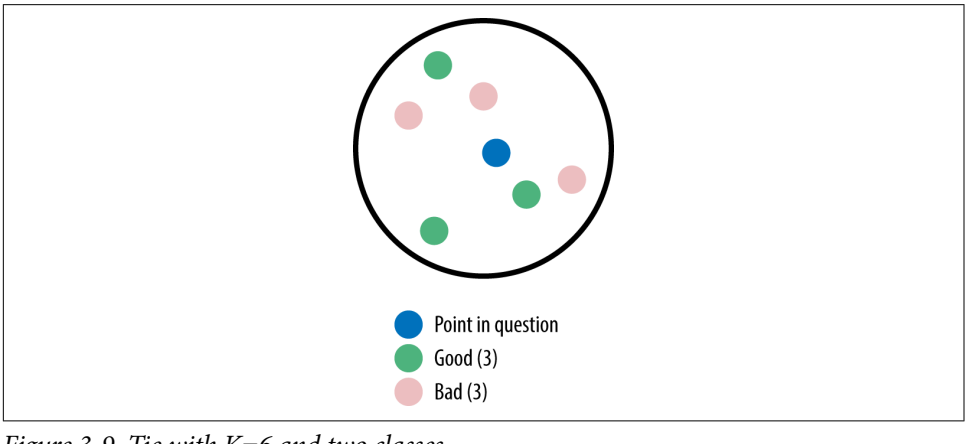

Figure 3-9. Tie with K=6 and two classes

If you picked a K of 5 instead (Figure 3-10), there wouldn't be a tie.

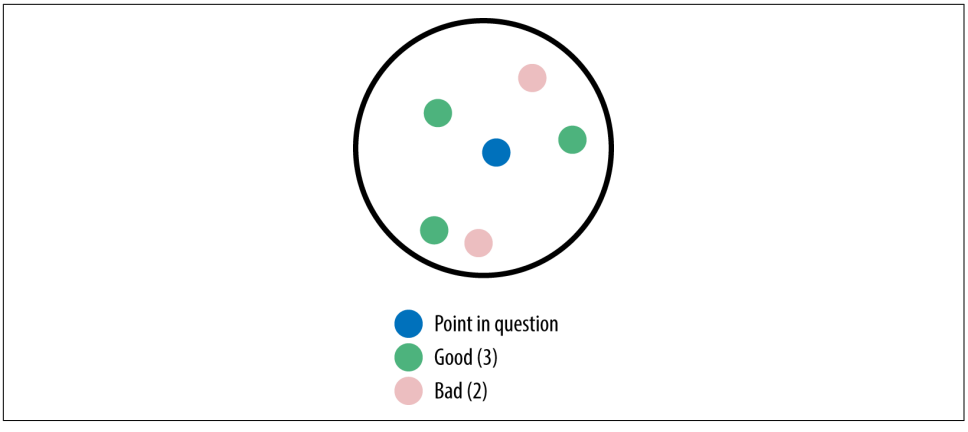

Figure 3-10. K=5 with two classes and no tie

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

Imagine there are three classes: lawful, chaotic, and neutral. A good heuristic is to pick a K of at least 3 because anything less will mean that there is no chance that each class will be represented. To illustrate, Figure 3-11 shows the case of K=2.

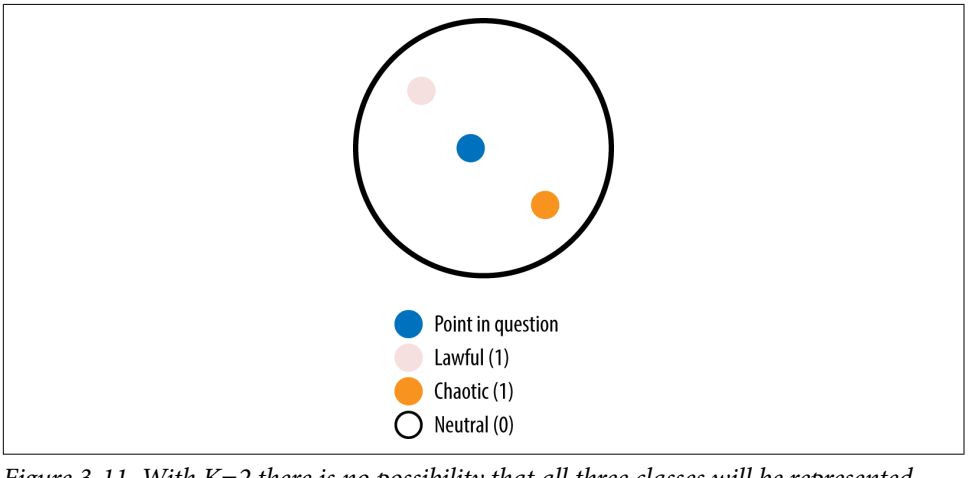

Figure 3-11. With K=2 there is no possibility that all three classes will be represented

Note how there are only two classes that get the chance to be used. Again, this is why we need to use at least K=3. But based on what we found in the first heuristic, ties are not a good thing. So, really, instead of K=3, we should use K=4 (as shown in Figure 3-12).

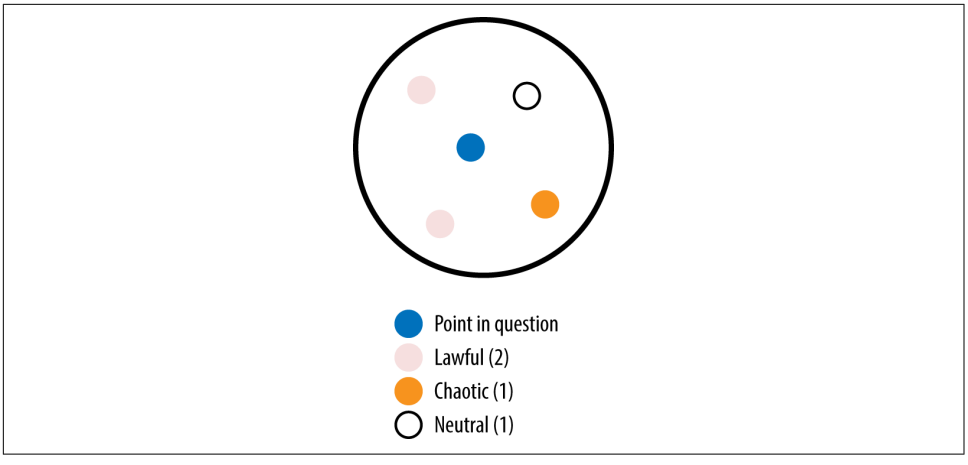

Figure 3-12. With K set greater than the number of classes, there is a chance for all classes to be represented

Choose a K that is low enough to avoid noise

As K increases, you eventually approach the size of the entire data set. If you were to pick the entire data set, you would select the most common class. To return to our brand affinity example, say you have 100 orders, as shown in Table 3-2.

Table 3-2. Brand to Count

Brand	Count
Widget Inc.	30
Bozo Group	23
Robots and Rockets	12
Ion 5	35
Total	100

If we were to set K=100, our answer will always be Ion 5 because Ion 5 is the distribution (the most common class) of the order history. That is not really what we want; instead, we want to determine the most recent order affinity. 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, like K=3 orK=11.

Algorithms for Picking K

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

grid searches. Many people assert that you should determine K based on domain knowledge that you have as the implementor. For instance, if you know that 5 is good enough, you can pick that. This problem where you are trying to minimize error based on an arbitrary K is known as a hill climbing problem. The idea is to iterate through a couple of possible Ks 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 performance. In other words, 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 in Florian Nigsch et al.'s Journal of Chemical Information and Modeling article, "Melting Point Prediction Employing k-Nearest Neighbor Algorithms and Genetic Parameter Optimization". Personally, I think iterating twice through 1% of the population size is good enough. You should have a decent idea of what works and what doesn't just by experimenting with different Ks.

Valuing houses in Seattle

Valuing houses in Seattle is a tough gamble. According to Zillow their Zestimate(tm) is consistently off in Seattle. But regardless how would we go about building something that tells us how valuable the houses are in Seattle? This section will walk through a simple example so that you can figure out with a reasonable rate what a house is worth based on freely available data from the King County Asessor.

If you'd like to follow along in the code examples checkout *https://github.com/thoughtfulml/examples-in-python/tree/master/k-nearest-neighbors*

About the Data

While the data is freely available, it wasn't easy to put together. I did a bit of cajoling to get the data well formed. There's a lot of features ranging from whether the house has a view of Mount Rainier or not. I felt that while that was an interesting exercise it's not really important to discuss here. In addition to the data they gave us geolocation has been added to all of the datapoints such that we can come up with a location distance much easier.

General Strategy

Our general Strategy for finding the values of houses in Seattle is to come up with something we're trying to minimize / maximize such that we know how good the model is. Since we will be looking at house values explicitly we can't calculate an "Accuracy" rate because every value will be different. So instead we will utilize a different metric called mean absolute error.

So with all models our goal is to minimize or maximize something in this case we're goign to minimize the mean absolute error. This is definied as the average of the absolute errors. The reason we'll use absolute error over any other common metrics (like mean squared error) is that it's useful. When it comes to house values it's hard to get intuition around the average squared error, when using absolute error we can instead say that our model is off by 70,000 or similar on average.

As for unit testing and functional testing we will approach this in a random fashion by stratifying the data into multiple chunks so that we can sample the mean absolute errors. This is mainly so that we don't find just one weird case where the mean absolute error was exceptionally low. We will not be talking extensively in this chapter about unit testing because this is an early chapter and I feel that it's important to focus on the overall testing of the model through mean absolute error.

Coding and Testing Design

The basic design of the code for this chapter is going to center around a Regressor class. This class will take in King County housing data that comes in via a flat file and calculate both an error rate and do the regression. We will not be doing any unit testing in this chapter but instead will visually test the code using the plot_error function we will build inside of the Regressor class.

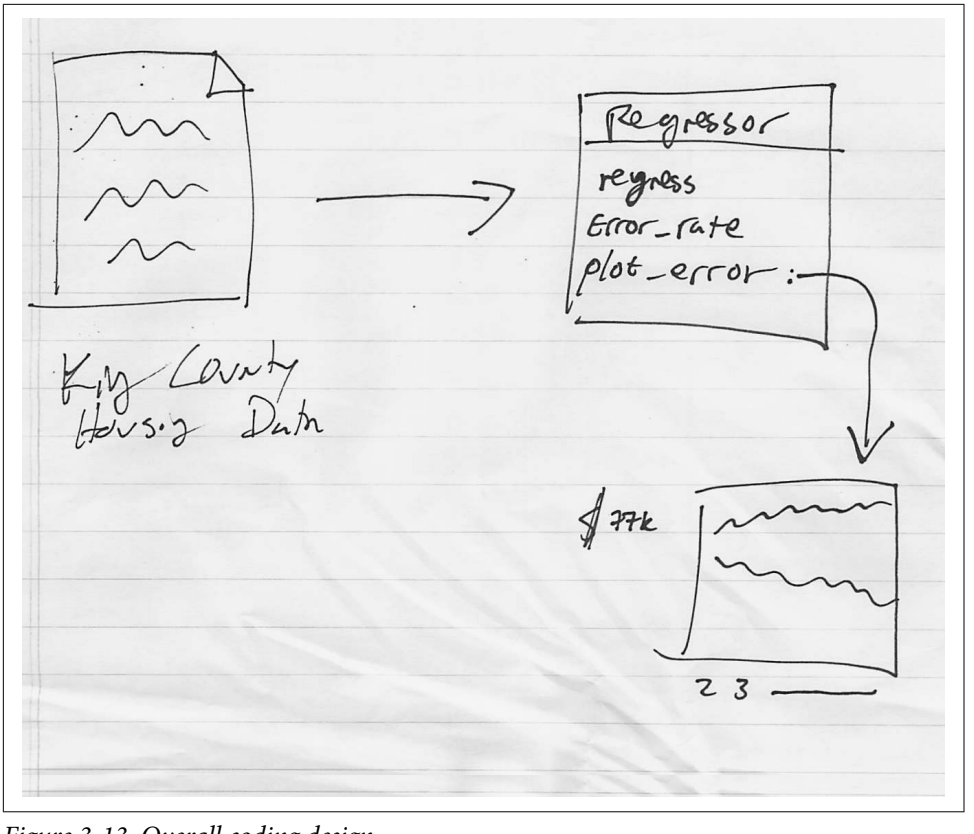

Figure 3-13. Overall coding design

For this chapter we will determine success by looking at the nuances of how our regressor works as we increase folds.

KNN Regressor Construction

To construct our KNN Regression we will utilize something called a KDTree. It's not essential that you know how these work but the idea is that inside of the KDTree it will store data in a easily queriable fashion based on distance. The distance metric we will use is the Euclidean Distance since it's easy to compute and will suit us just fine. You could try many other metrics to see whether the error rate was better or worse.

A note on packages

You'll note that we're using quite a few packages. Python has excellent tools available to do anything data science related such as Numpy, Pandas, Scikit-Learn, Scipy, and others.

Pandas and Numpy work together to build what is at it's core a multi dimensional array but operates similar to how a sql database allows you to query it. Pandas is the query interface and Numpy is the numerical processing underneath. You will also find other useful tools inside of the Numpy library.

Scikit-Learn is a collection of machine learning tools available for common algorithms (that we will be talking about in this book).

SciPy is a scientific computing library that allows us to do things like using a KDTree.

As we progress in the book we will rely heavily on these libraries.

```
from pandas import Series, DataFrame
import pandas as pd
import numpy as np
import numpy.random as npr
import random
from scipy.spatial import KDTree
from sklearn.metrics import mean_absolute_error
import sys

sys.setrecursionlimit(10000)

class Regression:
  def __init__(self, csv_file = None, data = None, values = None):
    if (data is None and csv_file is not None):
      df = pd.read_csv(csv_file)
      self.values = df['AppraisedValue']
      df = df.drop('AppraisedValue', 1)
      df = (df - df.mean()) / (df.max() - df.min())
      self.df = df
      self.df = self.df[['lat', 'long', 'SqFtLot']

    elif (data is not None and values is not None):
      self.df = data
      self.values = values
    else:
      raise ValueError("Must have either csv_file or data set")

    self.n = len(self.df)
    self.kdtree = KDTree(self.df)
    self.metric = np.mean
    self.k = 5
```

Do note that we had to set the recursion limit higher since KDTree will recurse and throw an error otherwise. There's a few things we're doing here I thought we should discuss.

One of them is the idea of normalizing data. This is a great trick to make all of the data similar. Otherwise what will happen is that we find something close that really shouldn't be. Or the bigger numered dimensions will skew results.

On top of that we're only selecting latitude longitude and SqFtLot. This is just because this is a proof of concept.

```
class Regression:
  # __init__
  def regress(self, query_point):
    distances, indexes = self.kdtree.query(query_point, self.k)
    m = self.metric(self.values.iloc[indexes])
    if np.isnan(m):
      zomg
    else:
      return m
```

Here we are querying the KDTree to find the closest K houses. We then use the metric, in this case mean, to calculate a regression value.

Though at this point we need to focus on the fact that all of this is great but needs to have some sort of test to make sure our data is working properly.

KNN Testing

Up until this point we've written a perfectly reasonable K-Nearest-Neighbors regression tool to tell us house prices in King County. But how well does it actually perform? To do that we use something called cross validation which involves the following generalized algorithm:

. * Take a training set and split it into two categories: testing and training * Use the training data to train the model * Use the testing data to test how well the model performs.

We can do that below with the following code.

```
class Regression:
  # __init__
  # regress
  def error_rate(self, folds):
    holdout = 1 / float(folds)
    errors = []
    for fold in range(folds):
      y_hat, y_true = self.__validation_data(holdout)
      errors.append(mean_absolute_error(y_true, y_hat))

    return errors

  def __validation_data(self, holdout):
    test_rows = random.sample(self.df.index, int(round(len(self.df) * holdout)))
    train_rows = set(range(len(self.df))) - set(test_rows)
```

```
    df_test = self.df.ix[test_rows]
    df_train = self.df.drop(test_rows)
    test_values = self.values.ix[test_rows]
    train_values = self.values.ix[train_rows]
    kd = Regression(data=df_train, values=train_values)

    y_hat = []
    y_actual = []

    for idx, row in df_test.iterrows():
      y_hat.append(kd.regress(row))
      y_actual.append(self.values[idx])

    return (y_hat, y_actual)
```

When it comes to terminology Folds are generally how many times you wish to split the data. So for instance if we had 3 folds we would hold 2/3 of the data for training and 1/3 for testing and iterate through the problem set 3 times.

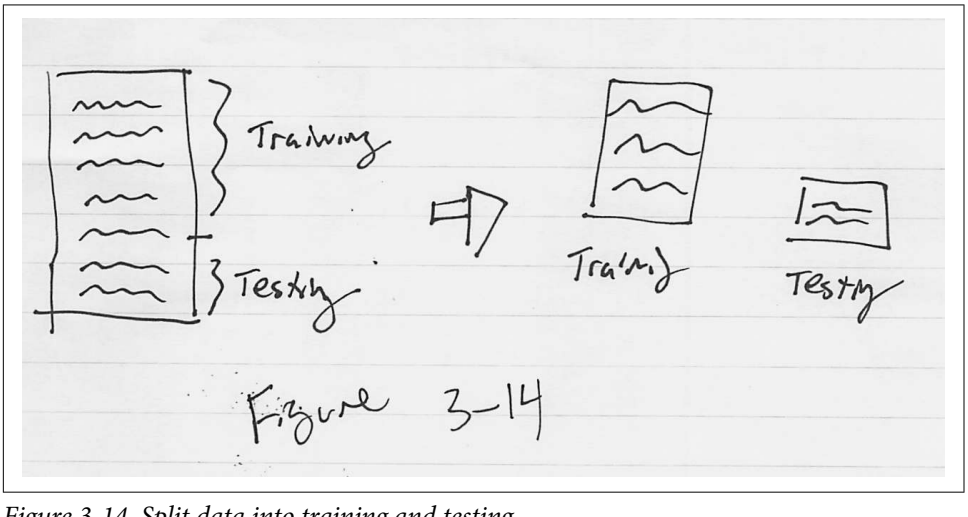

Figure 3-14. Split data into training and testing

Now these datapoints are interesting but how well does our model perform? Do to that let's take a visual approach and write the following which utilizes Pandas' graphics and matplotlib.

```
class Regression:
  # __init__
  # regress
  # error_rate
  # __validation_data
  def plot_error_rates(self):
    folds = range(2, 11)
    errors = pd.DataFrame({'max': 0, 'min': 0}, index=folds)
```

```
for f in folds:
  error_rates = r.error_rate(f)
  errors['max'][f] = max(error_rates)
  errors['min'][f] = min(error_rates)
errors.plot(title='Mean Absolute Error of KNN over different folds')
plt.show()
```

Running this yields the following graphic:

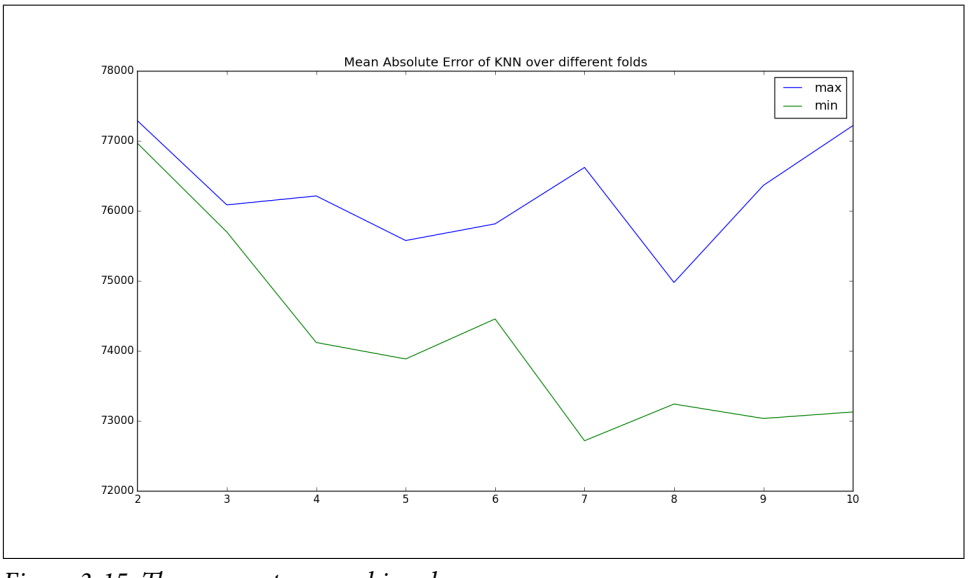

Figure 3-15. The error rates we achieved

As you can see starting with folds of 2 we start with a fairly tight absolute deviation of about $77,000 dollars. As we increase the folds and as a result reduce the testing sample that increases to a range of $73,000-$77,000. For a very simplistic model that is containing all information from waterfront property to condos this actually does quite well!

Conclusion

While K-Nearest Neighbors is a simple algorithm it yields quite good results. We have seen that for distance based problems we can utilize KNN to great effect. We also learned about how you can use this algorithm for either a classification or regression problem. We then analyzed the regression we built using a graphic representing the error.

We also showed that KNN has a downside which is inherent in any distance based metric, which is the curse of dimensionality. This curse is something we can overcome using feature transformations or selections.

Overall it's a great algorithm and stands the test of time.

CHAPTER 4

Naive Bayesian Classification

Remember email several years ago? You probably recall your inbox being 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.

Nowadays, we spend a lot less time filtering spam than we used to, thanks to Gmail and tools like SpamAssassin. Using a method called a Naive Bayesian Classifier, such tools have been able to mitigate the influx of spam to our inboxes. This chapter will explore that topic as well as:

Topics

- Bayes' theorem
- What a Naive Bayesian Classifier is and why it's called "naive"
- How to build a spam filter using a Naive Bayesian Classifier

As noted in Chapter 2, a Naive Bayes Classifier is a supervised and probabilistic learning method. It does well with data in which the inputs are independent from one another. It also prefers problems where the probability of any attribute is greater than zero.

Using Bayes' Theorem to Find Fraudulent Orders

Imagine you're running an online store and lately you've been overrun with fraudulent orders. You estimate that about 10% of all orders coming in are fraudulent. In other words, in 10% of orders, people are stealing from you. Now of course you want to mitigate this by reducing the fraudulent orders, but you are facing a conundrum.

Every month you receive at least 1,000 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 fraudulent or not, and a customer service representative costs around $15 per hour to hire, that totals 200 hours and $3,000 per year.

Another way of approaching this problem would be to construct a probability that an order is over 50% fraudulent. In this case, we'd expect the number of orders we'd have to look at to be much lower. But this is where things become difficult, because the only thing we can determine is the probability that 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 that fraudulent orders often use gift cards and multiple promotional codes. Using this knowledge, how would we determine what is fraudulent or not-namely, how would we calculate the probability of fraud given that the purchaser used a gift card?

To answer for that, we first have to talk about conditional probabilities.

Conditional Probabilities

Most people understand what we mean by the probability of something happening. For instance, the probability of an order being fraudulent is 10%. That's pretty straightforward. But what about the probability of an order being fraudulent given that it used a gift card? To handle that more complicated case, we need something called a conditional probability, which is defined as follows:

Equation 4-1. Conditional Probability

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

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. Here we'll cover how you'd notate each of these scenarios:

$A \cap B$ is called the intersection function but could also be thought of as the boolean operation AND. For instance, in Python it looks like this:

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

```
set(a) & set(b) #=> [1]
```

$A \cup B$ could be called the or function, as it is both A and B. For instance, in Python it looks like the following:

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

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

Finally, the probability of A given B looks as follows in Python:

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

total = 6.0

p_a_cap_b = len(a & b) / total
p_b = len(b) / total

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

This definition basically says 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 Figure 4-1.

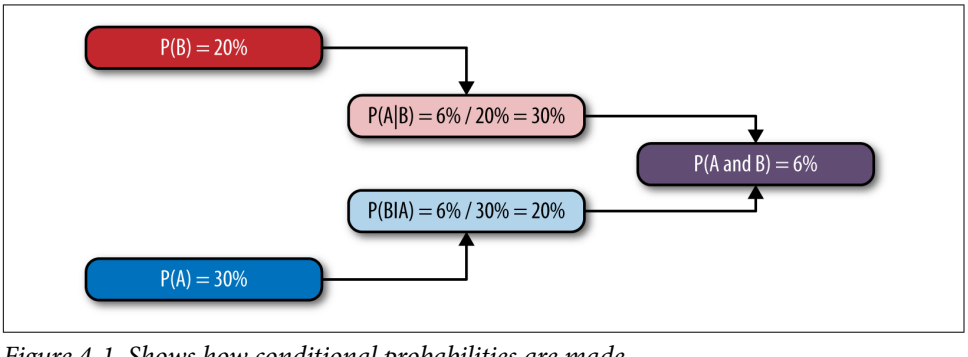

Figure 4-1. Shows how conditional probabilities are made

This shows how $P(A \mid B)$ sits between $P(A \text{ and } B)$ and $P(B)$

In our fraud example, let's say we want to measure the probability of fraud given that an order used 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 the problem that we cannot calculate P(Fraud|Giftcard) because that is hard to separate out. To solve this problem, we need to use a trick introduced by Bayes.

Inverse Conditional Probability (aka Bayes' Theorem)

In the 1700s, Reverend Thomas Bayes came up with the original research that would become Bayes' theorem. Pierre-Simon Laplace extended Bayes' research to produce the beautiful result we know today. Bayes' theorem is as follows:

Equation 4-2. Bayes Theorem

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

This is because of the following:

Equation 4-3. Bayes Theorem expanded

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

This is useful in our fraud example because we can effectively back out our result using other information. Using 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 gift card use is 10%, and based on our research the probability of gift card use in a fraudulent order is 60%. So what is the probability that an order is fraudulent given that it uses a gift card?

$$P(Fraud \mid Giftcard) = \frac{60\,\% \dot{1}0\,\%}{10\,\%} = 60\,\%$$

The beauty of this is that your work on measuring fraudulent orders is drastically reduced because all you have to look for is the orders with gift cards. Because the total number of orders is 1,000, and 100 of those are fraudulent, we will look at 60 of those fraudulent orders. Out of the remaining 900, 90 used gift cards, which brings the total we need to look at to 150!

At this point, you'll notice we reduced the orders needing fraud review from 1,000 to 40 (i.e., 4% of the 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 a gift card was used, but what about the problem of fraudulent orders given the fact that they

have gift cards, or 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 think back to probability class, you might recall that the probability of A and B happening is the probability of B given A times the probability of A. Mathematically, this looks like $P(A \cap B) = P(B \mid A)P(A)$. This is assuming these events are not mutually exclusive. Using something called a joint probability, this smaller result transforms into the 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:

Equation 4-4. Chain Rule

$$P(A_1, A_2, \cdots, A_n) = P(A_1)P(A_2 \mid A_1)P(A_3 \mid A_1, A_2)\cdots P(A_n \mid A_1, A_2, \cdots, A_{n-1})$$

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

Naievty in bayesian reasoning

The chain rule 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, as it doesn't depend on whether the order is fraudulent or not. At this point, we need to focus on finding the calculation for $P(Giftcard, Promos \mid Fraud)P(Fraud)$. If we apply the chain rule, this is equivalent to $P(Fraud, Giftcard, Promos)$.

You can see this by the following:

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

Now at this point we have a conundrum: how do you measure the probability of a promo code given fraud and gift cards? While this is the correct probability, it really

can be difficult to measure-especially with more features coming in. What if we were to be a tad naive and assume that we can get away with independence and just say that we don't care about the interaction between promo codes and gift cards, just the interaction of each independently with fraud?

In that case, our math would be much simpler:

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

This would be proportional to our numerator. And, to simplify things even more, we can assert that we'll normalize later with some magical Z, which is 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 simply determine which input-fraud or not fraud-yields the highest probability. See Table 4-1.

Table 4-1. Probability of gift cards versus promos

	Fraud	Not Fraud
Gift card present	60%	30%
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 based purely on whether it has a gift card present and whether it used multiple promos. The probability that an order is fraudulent given the use of gift cards and multiple promos is 62.5%. While we can't exactly figure out how much savings this gives you in terms of the number of orders you must review, we know that we're using better information and making a better judgment.

There is one problem, though: what happens when the probability of using multiple promos given a fraudulent order is zero? A zero result can happen for several reasons, including that there just isn't enough of a sample size. The way we solve this is by using something called a pseudocount.

Pseudocount

There is one big challenge with a Naive Bayesian Classifier, and that is the introduction of new information. For instance, let's say we have a bunch of emails that are classified as spam or ham. We build our probabilities using all of this data, but then something bad happens: a new spammy word, fuzzbolt. Nowhere in our data did we see the word fuzzbolt, and so when we 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 toward the data we have.

Because 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, the email subject "Fuzzbolt: Prince of Nigeria." Assuming we strip off of, we have the data shown in Table 4-2.

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

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 because fuzzbolt isn't present. At that point, because 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: pseudocount. When we go about calculating the probability, we add one to the count of the word. So, in other words, 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 one to each count to ensure that it is never zero.

So in our preceding example, let's say we have 3,000 words. We would give fuzzbolt a score of $\frac{1}{3000}$. The other scores would change slightly, but this avoids the zeroing-out problem.

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 you have learned before, Naive Bayesian Classifiers can be easily calculated, and operate well under strongly independent conditions. In this example, we will cover the following:

- What the classes look like interacting with each other
- A good data source
- A tokenization model

- An objective to minimize our error
- A way to improve over time

Setup Notes

Python is constantly changing and I have tried to keep the examples working under both 2.7.x and 3.0.x python. That being said things might change as python changes. For more comprehensive information check out the github repo at *https://github.com/thoughtfulml/examples-in-python/tree/master/naive-bayes*

Coding and Testing Design

In our example, each email has an object that takes an .eml type text file that then tokenizes it into something the SpamTrainer can utilize for incoming email messages. See Figure 4-2 for the class diagram.

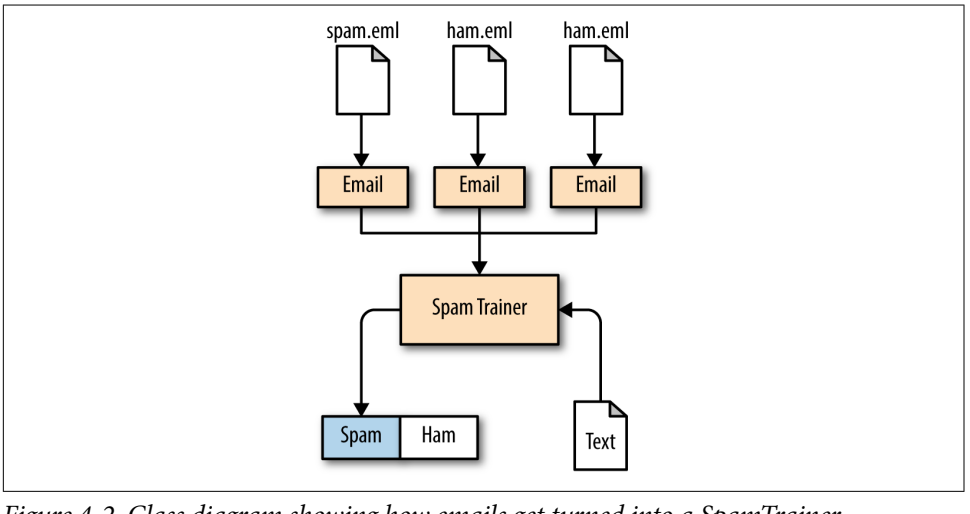

Figure 4-2. Class diagram showing how emails get turned into a SpamTrainer

When it comes to testing we will focus on the tradeoff between false positives, and false negatives. With spam detection it becomes important to realize that a false positive (classifying an email as spam when it isn't) could actually be very bad for business. We will focus on minimizing the false positive rate but similar results could be applied to minimizing false negatives or having them equal each other.

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.

This data set has 4,327 total messages, of which 2,949 are ham and 1,378 are spam. For our proof of concept, this should work well enough. Email Class ^^^^^

The Email class has one responsibility, which is to parse an incoming email message according to the RFC for emails. To handle this, we use the mail gem because 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 go through this step by step.

Starting with the simple plaintext case, we'll 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 email and will work for our purposes. Note that the split between a message and header in an email 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 our test case, which is:

Unit testing in Python

Up until this point we haven't introduced the unittest package in Python. It's main objective is to define unit tests for us to run on our code. Like similar unit testing frameworks in other languages like Ruby, we build a class that is prefixed with "Test" and then implement specific methods.

Methods to implement:

- Any method that is prefixed with test_ will be treated as a test to be run
- setUp(self) is a special method that gets run before any test gets run. Think of this like a block of code that gets run before all tests.

Table 4-3. Inside of our tests there are many assertions we can run on those tests like

Method	Checks
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False

Method	Checks
assertIs(a,b)	a is b
assertIsNot(a,b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a,b)	a in b
assertNotIn(a,b)	a not in b
assertIsInstance(a,b)	isinstance(a,b)
assertNotIsInstance(a,b)	not isinstance(a,b)

Do note that we will not use all of these methods but they are listed here for future reference.

```
import unittest
import io
import re
from naive_bayes.email_object import EmailObject

class TestPlaintextEmailObject(unittest.TestCase):
  CLRF = "\n\n"
  def setUp(self):
    self.plain_file = './tests/fixtures/plain.eml'
    self.plaintext = io.open(self.plain_file, 'r')
    self.text = self.plaintext.read()
    self.plaintext.seek(0)
    self.plain_email = EmailObject(self.plaintext)

  def test_parse_plain_body(self):
    body = self.CLRF.join(self.text.split(self.CLRF)[1:])
    self.assertEqual(self.plain_email.body(), body)

  def test_parses_the_subject(self):
    subject = re.search("Subject: (.*)", self.text).group(1)
    self.assertEqual(self.plain_email.subject(), subject)
```

Now instead of relying purely on regular expressions, we want to utilize a gem. We'll use the stdlib of python, which will handle all of the nitty-gritty details. Making email work for this particular case, we have:

BeautifulSoup is a library that parses HTML and XML

```
import email
from BeautifulSoup import BeautifulSoup

class EmailObject:
  def __init__(self, filepath, category = None):
    self.filepath = filepath
    self.category = category
    self.mail = email.message_from_file(self.filepath)

  def subject(self):
    return self.mail.get('Subject')

  def body(self):
    return self.mail.get_payload(decode=True)
```

Now that we have captured the case of plaintext, we need to solve the case of HTML. For that, we want to capture only the inner_text. But first we need a test case, which looks something like this:

```
import unittest
import io
import re
from BeautifulSoup import BeautifulSoup
from naive_bayes.email_object import EmailObject

class TestHTMLEmail(unittest.TestCase):
  def setUp(self):
    self.html_file = io.open('./tests/fixtures/html.eml', 'rb')
    self.html = self.html_file.read()
    self.html_file.seek(0)
    self.html_email = EmailObject(self.html_file)

  def test_parses_stores_inner_text_html(self):
    body = "\n\n".join(self.html.split("\n\n")[1:])
    expected = BeautifulSoup(body).text
    self.assertEqual(self.html_email.body(), expected)

  def test_stores_subject(self):
    subject = re.search("Subject: (.*)", self.html).group(1)
    self.assertEqual(self.html_email.subject(), subject)
```

As mentioned, we're using BeautifulSoup to calculate the inner_text, and we'll have to use it inside of the Email class as well. Now the problem is that we also need to detect the content_type. So we'll add that in:

```
import email
from BeautifulSoup import BeautifulSoup

class EmailObject:
  def __init__(self, filepath, category = None):
    self.filepath = filepath
    self.category = category
    self.mail = email.message_from_file(self.filepath)
```

```
def subject(self):
  return self.mail.get('Subject')

def body(self):
  content_type = part.get_content_type()
  body = part.get_payload(decode=True)

  if content_type == 'text/html':
   return BeautifulSoup(body).text
  elif content_type == 'text/plain':
   return body
  else:
   return ''
```

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 mentioned earlier in the chapter, you can see the multipart version.

Now we have a working email parser, but we still have to deal with tokenization, or what to extract from the body and subject.

Tokenization and Context

As Figure 4-3 shows, there are numerous ways to tokenize text, such as by stems, word frequencies, and words. In the case of spam, we are up against a tough problem because things are more contextual. The phrase Buy now sounds spammy, whereas Buy and now do not. Because we are building a Naive Bayesian Classifier, we are assuming that each individual token is contributing to the spamminess of the email.

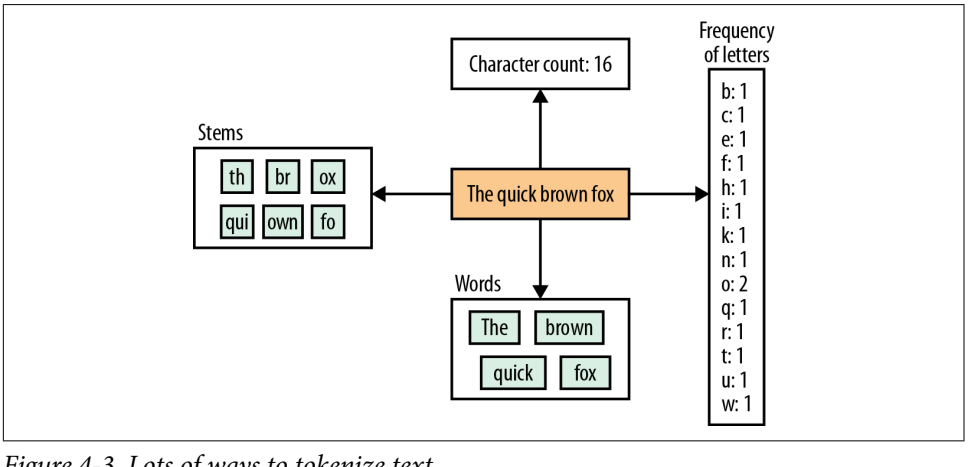

Figure 4-3. Lots of ways to tokenize text

The goal of the tokenizer we'll build is to extract 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. Our tokenizer should also downcase all strings to keep them similar:

```
import unittest
from naive_bayes.tokenizer import Tokenizer

class TestTokenizer(unittest.TestCase):
  def setUp(self):
    self.string = "this is a test of the emergency broadcasting system"

  def test_downcasing(self):
    expectation = ["this", "is", "all", "caps"]

    actual = Tokenizer.tokenize("THIS IS ALL CAPS")
    self.assertEqual(actual, expectation)

  def test_ngrams(self):
    expectation = [
      [u'\u0000', "quick"],
      ["quick", "brown"],
      ["brown", "fox"],
    ]

    actual = Tokenizer.ngram("quick brown fox", 2)
    self.assertEqual(actual, expectation)
```

As promised, we do two things in this tokenizer code. First, we lowercase all words. Second, instead of returning an array, we use a block. This is to mitigate memory constraints, as there is no need to build an array and return it. This makes it lazier. To make the subsequent tests work, though, we will have to fill in the skeleton for our tokenizer module like so:

```
import re

class Tokenizer:
  NULL = u'\u0000'

  @staticmethod
  def tokenize(string):
    return re.findall("\w+", string.lower())

  @staticmethod
  def ngram(string, ngram):
    tokens = Tokenizer.tokenize(string)

    ngrams = []

    for i in range(len(tokens)):
      shift = i-ngram+1
      padding = max(-shift,0)
      first_idx = max(shift, 0)
      last_idx = first_idx + ngram - padding
```

```
      ngrams.append(Tokenizer.pad(tokens[first_idx:last_idx], padding))

    return ngrams

  @staticmethod
  def pad(tokens, padding):
    padded_tokens = []

    for i in range(padding):
      padded_tokens.append(Tokenizer.NULL)

    return padded_tokens + tokens
```

Now that we have a way of parsing and tokenizing emails, we can move on to the Bayesian portion: the SpamTrainer.

SpamTrainer

We now need to build the SpamTrainer, which will accomplish three things:

- Storing training data.
- Building a Bayesian classifier.
- Minimizing the false positive rate by testing.

Storing training data

The first step we need to tackle is to store training data from a given set of email messages. In a production environment, you would pick something that has persistence. In our case, we will go with storing everything in one big dictionary.

Remember that most machine learning algorithms have two steps: training and then computation. Our training step will consist of these substeps:

- Storing a set of all categories
- Storing unique word counts for each category
- Storing the totals for each category

So first we need to capture all of the category names; that test would look something like this:

A set is a unique collection of data

```
import unittest
import io
import sets
from naive_bayes.email_object import EmailObject
from naive_bayes.spam_trainer import SpamTrainer

class TestSpamTrainer(unittest.TestCase):
  def setUp(self):
    self.training = [['spam', './tests/fixtures/plain.eml'], ['ham', './tests/fixtures/small.eml']
    self.trainer = SpamTrainer(self.training)
    file = io.open('./tests/fixtures/plain.eml', 'r')
    self.email = EmailObject(file)

  def test_multiple_categories(self):
    categories = self.trainer.categories
    expected = sets.Set([k for k,v in self.training])
    self.assertEqual(categories, expected)
```

The solution is in the following code:

```
from sets import Set
import io
from tokenizer import Tokenizer
from email_object import EmailObject
from collections import defaultdict

class SpamTrainer:
  def __init__(self, training_files):
    self.categories = Set()

    for category, file in training_files:
      self.categories.add(category)

    self.totals = defaultdict(float)

    self.training = {c: defaultdict(float) for c in self.categories}

    self.to_train = training_files

  def total_for(self, category):
    return self.totals[category]
```

You'll notice we're just using a set to capture this for now, as it'll hold on to 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:

```
class TestSpamTrainer(unittest.TestCase):
  # setUp
  # test_multiple_categories

  def test_counts_all_at_zero(self):
    for cat in ['_all', 'spam', 'ham', 'scram']:
      self.assertEqual(self.trainer.total_for(cat), 0)
```

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

```
class SpamTrainer:
  # __init__
  # total_for

  def train(self):
    for category, file in self.to_train:
      email = EmailObject(io.open(file, 'rb'))

      self.categories.add(category)

      for token in Tokenizer.unique_tokenizer(email.body()):
        self.training[category][token] += 1
        self.totals['_all'] += 1
        self.totals[category] += 1

    self.to_train = {}
```

Now 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

To refresh your memory, Bayes' theorem is:

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

But because we're being naive about this, we've distilled it into something much simpler:

$$Score(Spam, W_1, W_2, \cdots, W_n) = P(Spam)P(W_1 \mid Spam)P(W_2 \mid Spam) \cdots P(W_n \mid Spam)$$

which is then divided by some normalizing constant, Z.

Our goal now is to build the methods score, normalized_score, and classify. The score method will just be the raw score from the preceding calculation, while normalized_score will fit the range from 0 to 1 (we get this by dividing by the total sum, Z).

The score method's test is as follows:

```
class TestSpamTrainer(unittest.TestCase):
  # setUp
  # test_multiple_categories
  # test_counts_all_at_zero
```

```
  def test_probability_being_1_over_n(self):
    trainer = self.trainer
    scores = trainer.score(self.email).values()

    self.assertAlmostEqual(scores[0], scores[-1])

    for i in range(len(scores)-1):
      self.assertAlmostEqual(scores[i], scores[i+1])
```

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

```
class SpamTrainer:
  # __init__
  # total_for
  # train

  def score(self, email):
    self.train()

    cat_totals = self.totals

    aggregates = {cat: cat_totals[cat]/cat_totals['_all'] for cat in self.categories}

    for token in Tokenizer.unique_tokenizer(email.body()):
      for cat in self.categories:
        value = self.training[cat][token]
        r = (value+1)/(cat_totals[cat]+1)
        aggregates[cat] *= r

    return aggregates
```

This test does the following:

Trains the model if it's not already trained (the train! method handles this).

For each token of the blob of an email we iterate through all categories and calculate the probability of that token being within that category. This calculates the Naive Bayesian score of each without dividing by Z.

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

```
class TestSpamTrainer(unittest.TestCase):
  # setUp
  # test_multiple_categories
  # test_counts_all_at_zero
  # test_probability_being_1_over_n

  def test_adds_up_to_one(self):
    trainer = self.trainer
    scores = trainer.normalized_score(self.email).values()
```

```
        self.assertAlmostEqual(sum(scores), 1)
        self.assertAlmostEqual(scores[0], 1/2.0)
```

and subsequently on the SpamTrainer class we have:

```
class SpamTrainer:
  # __init__
  # total_for
  # train
  # score

  def normalized_score(self, email):
    score = self.score(email)
    scoresum = sum(score.values())

    normalized = {cat: (aggregate/scoresum) for cat, aggregate in score.iteritems()}
    return normalized
```

Calculating a classification

Because 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 evenly split? Probably the best course of action is to go with which is more popular, whether it be turkey or tofu. What about the case where the probability is the same? In that case, we can just go with alphabetical order.

When testing for this, we need to introduce a preference order-that is the occurrence of each category. A test for this would be:

```
class TestSpamTrainer(unittest.TestCase):
  # setUp
  # test_multiple_categories
  # test_counts_all_at_zero
  # test_probability_being_1_over_n
  # test_adds_up_to_one

  def test_preference_category(self):
    trainer = self.trainer
    expected = sorted(trainer.categories, key=lambda cat: trainer.total_for(cat))

    self.assertEqual(trainer.preference(), expected)
```

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

```
class SpamTrainer:
  # __init__
  # total_for
  # train
  # score
```

```
  # normalized_score

  def preference(self):
    return sorted(self.categories, key=lambda cat: self.total_for(cat))
```

Now that we have preference set up, we can test for our classification being correct. The code to do that is as follows:

```
class TestSpamTrainer(unittest.TestCase):
  # setUp
  # test_multiple_categories
  # test_counts_all_at_zero
  # test_probability_being_1_over_n
  # test_adds_up_to_one
  # test_preference_category

  def test_give_preference_to_whatever_has_the_most(self):
    trainer = self.trainer
    score = trainer.score(self.email)

    preference = trainer.preference()[-1]
    preference_score = score[preference]

    expected = SpamTrainer.Classification(preference, preference_score)
    self.assertEqual(trainer.classify(self.email), expected)
```

Getting this to work in code again is simple:

```
class SpamTrainer:
  # __init__
  # total_for
  # train
  # score
  # normalized_score
  # preference

  class Classification:
    def __init__(self, guess, score):
      self.guess = guess
      self.score = score
    def __eq__(self, other):
      return self.guess == other.guess and self.score == other.score

  def classify(self, email):
    score = self.score(email)

    max_score = 0.0
    preference = self.preference()
    max_key = preference[-1]

    for k,v in score.iteritems():
      if v > max_score:
        max_key = k
```

```
            max_score = v
        elif v == max_score and preference.index(k) > preference.index(max_key):
            max_key = k
            max_score = v
    return self.Classification(max_key, max_score)
```

Error Minimization Through Cross-Validation

At this point, we need to measure how well our model works. To do so, we need to take the data that we downloaded earlier 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.

Minimizing 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 what we're optimizing for. Instead, we want to minimize false positives. 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 the email isn't, then the user will lose her emails. We want our spam filter to have as few 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, but they are less important because we are trying to reduce spam that enters someone's mailbox, not eliminate it.

To accomplish this, we first need to take some information from our data set, which we'll cover next.

Building the two folds

Inside the spam email training data is a file called keyfile.label. It contains information about whether the file is spam or ham. So using that we can build a cross validation script. First let's start with setup which involves importing the packages we've worked on and some io and regular expression libraries.

```
from spam_trainer import SpamTrainer
from email_object import EmailObject
import io
import re

print "Cross Validation"
```

```
  correct = 0
  false_positives = 0.0
  false_negatives = 0.0
  confidence = 0.0
```

This doesn't do much yet except start with a zeroed counter for correct, false positives, false negatives and confidence being 0. To set up the test we need to load the label data and turn that into a SpamTrainer object. We can do that using the following.

```
def label_to_training_data(fold_file):
  training_data = []

  for line in io.open(fold_file, 'rb'):
    label_file = line.rstrip().split(' ')
    training_data.append(label_file)

  print training_data
  return SpamTrainer(training_data)

trainer = label_to_training_data('./tests/fixtures/fold1.label')
```

This instantiates a trainer object by calling the label_to_training_data function. Next we parse the emails we have in fold number 2.

```
def parse_emails(keyfile):
  emails = []
  print "Parsing emails for " + keyfile

  for line in io.open(keyfile, 'rb'):
    label, file = line.rstrip().split(' ')

    emails.append(EmailObject(io.open(file, 'rb'), category=label))

  print "Done parsing files for " + keyfile
  return emails

emails = parse_emails('./tests/fixtures/fold2.label')
```

Now we have a trainer object and emails parsed. All we need to do now is calculate the accuracy and validation metrics.

```
def validate(trainer, set_of_emails):
  correct = 0
  false_positives = 0.0
  false_negatives = 0.0
  confidence = 0.0

  for email in set_of_emails:
    classification = trainer.classify(email)
    confidence += classification.score
```

```
      if classification.guess == 'spam' and email.category == 'ham':
        false_positives += 1
      elif classification.guess == 'ham' and email.category == 'spam':
        false_negatives += 1
      else:
        correct += 1

    total = false_positives + false_negatives + correct

    false_positive_rate = false_positives/total
    false_negative_rate = false_negatives/total
    accuracy = (false_positives + false_negatives) / total
    message = """
    False Positives: {0}
    False Negatives: {1}
    Accuracy: {2}
    """.format(false_positive_rate, false_negative_rate, accuracy)
    print message

validate(trainer, emails)
```

Finally we can analyze the other direction of the cross validation i.e. validating fold1 against a fold2 trained model

```
trainer = label_to_training_data('./tests/fixtures/fold2.label')
emails = parse_emails('./tests/fixtures/fold1.label')
validate(trainer, emails)
```

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:

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

        False Positives: 0.0036985668053629217
        False Negatives: 0.16458622283865001
        Error Rate: 0.16828478964401294

Cross Validation::Fold2 unigram model
  validates fold2 against fold1 with a unigram model

        False Positives: 0.005545286506469501
        False Negatives: 0.17375231053604437
        Error Rate: 0.17929759704251386
```

Table 4-4. Spam vs Ham

Category	Email Count	Word Count	Probability of Email	Probability of word
Spam	1,378	231,472	31.8%	36.3%

Category	Email Count	Word Count	Probability of Email	Probability of word
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 default to that and more often than not we'll 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

In this chapter, we have delved into building and understanding a Naive Bayesian Classifier. As you have learned it, this algorithm is well suited for data that can be asserted to be independent. Being a probablistic model, it works well for classifying data into multiple directions given the underlying score. This supervised learning method is useful for fraud detection, spam filtering, and any other problem that has these types of features.