Scaled Up Machine Learning--HW 1

Safyre Anderson

January 19, 2016

HW1.0.0. Define big data. Provide an example of a big data problem in your domain of expertise.

Big data are datasets that overwhelm a traditional single computer system, and typically characterized by the "3 V's": volume, velocity, and veracity. In other words, the extent of either one of these 3 V's (the size of the dataset, the speed of ingestion, and the variety of features) prevents Big Data from being analyzed on the limited hardware of a personal computer. Currently, I work in FinTech and one of the most important Big Data problems that exists is the near real-time detection of fraudulent transactions. There are over $1 billion people online that transact between 400 billion to 1 billion times per day (depending on what article you read), with variability in customers, transfers, purchases, payment methods, and many more kinds of recorded data--easily fulfilling the 3 V's.

HW1.0.1.In 500 words (English or pseudo code or a combination) describe how to estimate the bias, the variance, the irreducible error for a test dataset T when using polynomial regression models of degree 1, 2,3, 4,5 are considered. How would you select a model?

One of the recommended readings provided an excellent framework for this problem: https://theclevermachine.wordpress.com/2013/04/21/model-selection-underfitting-overfitting-and-the-bias-variance-tradeoff/ (https://theclevermachine.wordpress.com/2013/04/21/model-selection-underfitting-overfitting-and-the-bias-variance-tradeoff/).

Essentially, total error of a model is the sum of the squared error of bias, variance, and residual squared (noise). Thus on a high level, we will want to to determine the bias, variance, and total error of each model we test. From these three parameters, we can estimate the noise of each estimated true model by subtracting the mean squared error from the total test error:

Expected Prediction Error
$= E[(g(x^*) - E[g(x^*)]^2] + (E[g(x^*)] - f(x^*))^2 + E[(y^* - f(x))^2] = E[(g(x^*) - y^*)^2]$

In our case, $f(x)$ is unknown. However, we are given y^* and we can bootstrap training and test data to get $g(x^*)$ and E[g(x^*)]$ over all models (of the same degree).

In general:

Given a dataset:

```
for each polynomial of degree 1 through 5:

    for each bootstrapped sample of the dataset:
        subset $x%$ to training
        subset $1-x%$ to testing, where x is a fraction of the whole dataset

        fit a model on the training data
        predict $g(x^*)$ on the testing data

        calculate estimated predicted error (mean squared difference of observ
    ed $y^*_{test}$ from predicted y ($g(x^*)$).
        calculate variance (mean of the variance of $g(x^*)$ across all datase
    ts)
```

Here, we should already by able to select the best model by choosing the parameter(s) that achieved the lowest estimated predicted error. What about bias and noise? We still haven't figured those out yet.

Since noise is independent of the model, it is a constant term in the above equation. Thus, we can rethink this problem as such: $Expected\ Error - Variance = Bias + Constant$. To estimate the noise, we can take advantage of the bias variance tradeoff-- with the highest complexity model, we can assume bias approaches 0. Thus the equation becomes $Expected\ Error - Variance = Constant$. At polynomial N = 5, we can estimate the noise and finally solve for the bias:
$Bias = Expected\ Error - Variance - Constant$.

Part 2

In the remainder of this assignment you will produce a spam filter that is backed by a multinomial naive Bayes classifier b (see http://nlp.stanford.edu/IR-book/html/htmledition/properties-of-naive-bayes-1.html (http://nlp.stanford.edu/IR-book/html/htmledition/properties-of-naive-bayes-1.html)), which counts words in parallel via a unix, poor-man's map-reduce framework.

For the sake of this assignment we will focus on the basic construction of the parallelized classifier, and not consider its validation or calibration, and so you will have the classifier operate on its own training data (unlike a field application where one would use non-overlapping subsets for training, validation and testing).

The data you will use is a curated subset of the Enron email corpus (whose details you may find in the file enronemail_README.txt in the directory surrounding these instructions).

=====Instructions/Goals=====

In this directory you will also find starter code (pNaiveBayes.sh), (similar to the pGrepCount.sh code that was presented in this weeks lectures), which will be used as control script to a python mapper and reducer that you will supply at several stages. Doing some exploratory data analysis you will see (with this very small dataset) the following\:

```
> wc -l enronemail_1h.txt  #100 email records

      100 enronemail_1h.txt


> cut -f2 -d$'\t' enronemail_1h.txt|wc  #extract second field which is SPAM flag

      101     394    3999


JAMES-SHANAHANs-Desktop-Pro-2:HW1-Questions jshanahan\$ cut -f2 -d$'\t'
enronemail_1h.txt|head

0

0

0

0

0

0

0

0

1

1

> head -n 100 enronemail_1h.txt|tail -1|less #an example SPAM email record

018.2001-07-13.SA_and_HP       1

[ilug] we need your assistance to invest in your country        dear sir/madam,
i am well confident of your capability to assist me in  a transaction for mutual
benefit of both parties, ie  (me and you) i am also believing that you will not
expose or betray the trust and confidence i am about  to establish with you. i
have decided to contact you  with greatest delight and personal respect.  well,
i am victor sankoh, son to
mr. foday  sankoh  who was arrested by the ecomog peace keeping force  months
ago in my country sierra leone.
```

HW1.1. Read through the provided control script (pNaiveBayes.sh)

and all of its comments. When you are comfortable with their purpose and function, respond to the remaining homework questions below. A simple cell in the notebook with a print statmement with a "done" string will suffice here. (dont forget to include the Question Number and the quesition in the cell as a multiline comment!)

In [1]:
```
# just to run it to show i've read it. I know there aren't mappers and
reducers yet!
!chmod +x pNaiveBayes.sh
!./pNaiveBayes.sh
```

```
(standard_in) 1: parse error
split: enronemail_1h.txt: illegal line count
Traceback (most recent call last):
  File "./reducer.py", line 27, in <module>
    spam_total += int(spam_count)
NameError: name 'spam_count' is not defined
```

HW1.2. Provide a mapper/reducer pair that, when executed by pNaiveBayes.sh

will determine the number of occurrences of a single, user-specified word. Examine the word “assistance” and report your results.

To do so, make sure that

- mapper.py counts all occurrences of a single word, and
- reducer.py collates the counts of the single word.

CROSSCHECK: >grep assistance enronemail_1h.txt|cut -d$'\t' -f4| grep assistance|wc -l 8

In [ ]:
```
# download into current directory
!wget https://www.dropbox.com/sh/jylzkmauxkostck/AAC_6JZH7yqMcxfEGPc4-
_xJa/enronemail_1h.txt?dl=0 -O enronemail_1h.txt

# running wc -l enronemail_1h.txt outputs linecount = 0
# due to \r line break, need to change to \n line break (mac, unix, re
spectively)
!perl -pi -e 's/\r/\n/g' enronemail_1h.txt

!wc -l enronemail_1h.txt
```

In [2]:
```
%%writefile mapper.py
#!/usr/bin/python
import re
import sys
import os
import numpy as np

# store a regex expression into a pattern object
# that seeks words including underscores and single quotes
WORD_RE = re.compile(r"[\w']+")

# file input
filename = sys.argv[1]

# for this part, just assume word_list is length 1
word_list = sys.argv[2]
count = 0

with open(filename, 'rU') as f:
    for line in f.readlines():
        for word in word_list.split():
            counts = [1 if x == word else 0 for x in WORD_RE.findall(l
ine)]
            counts = np.array(counts)

            if counts.sum() > 0:
                print word + " " + str(1)
```

```
Overwriting mapper.py
```

Making the mapper.py an executable:

In [3]:
```
!chmod a+x mapper.py
```

In [9]:
```
%%writefile reducer.py
#!/usr/bin/python

import sys

files_list = sys.argv[1].split()

cur_word = ""

word_counts = {}

for f in files_list:
    with open(f, 'rU') as countfile:
        for line in countfile.readlines():
            new_word, count = line.split()
            if new_word != cur_word:
                # print the final count of the current word and start
over
                cur_word = new_word
                word_counts[cur_word] = word_counts.get(cur_word, 0) +
int(count)

            else:
                word_counts[new_word] += int(count)

for word in word_counts:
    print word + "\t" + str(word_counts[word])
```

```
Overwriting reducer.py
```

In [10]:
```
!chmod a+x reducer.py
```

In [11]:
```
%%writefile pNaiveBayes.sh
#!/bin/bash
## pNaiveBayes.sh
## Author: Jake Ryland Williams
## Usage: pNaiveBayes.sh m wordlist
## Input:
##       m = number of processes (maps), e.g., 4
##       wordlist = a space-separated list of words in quotes, e.g., "
the and of"
##
## Instructions: Read this script and its comments closely.
##               Do your best to understand the purpose of each comman
d,
##               and focus on how arguments are supplied to mapper.py/
reducer.py,
##               as this will determine how the python scripts take in
```

```
put.
##               When you are comfortable with the unix code below,
##               answer the questions on the LMS for HW1 about the sta
rter code.

## collect user input
m=$1 ## the number of parallel processes (maps) to run
wordlist=$2 ## if set to "*", then all words are used

## a test set data of 100 messages
data="enronemail_1h.txt"

## the full set of data (33746 messages)
# data="enronemail.txt"

## 'wc' determines the number of lines in the data
## 'perl -pe' regex strips the piped wc output to a number
linesindata=`wc -l $data | perl -pe 's/^.*?(\d+).*?$/$1/'`

## determine the lines per chunk for the desired number of processes
linesinchunk=`echo "$linesindata/$m+1" | bc`

## split the original file into chunks by line
split -l $linesinchunk $data $data.chunk.

## assign python mappers (mapper.py) to the chunks of data
## and emit their output to temporary files
for datachunk in $data.chunk.*; do
    ## feed word list to the python mapper here and redirect STDOUT to
a temporary file on disk
    ####
    ####
    ./mapper.py $datachunk "$wordlist" > $datachunk.counts &
    ####
    ####
done
## wait for the mappers to finish their work
wait

## 'ls' makes a list of the temporary count files
## 'perl -pe' regex replaces line breaks with spaces
countfiles=`\ls $data.chunk.*.counts | perl -pe 's/\n/ /'`

## feed the list of countfiles to the python reducer and redirect STDO
UT to disk
####
####
./reducer.py "$countfiles" > $data.output
####
####
```

```
## clean up the data chunks and temporary count files
\rm $data.chunk.*
```

```
Overwriting pNaiveBayes.sh
```

In [12]:
```
!chmod a+x pNaiveBayes.sh
```

Run for 1.2, first argument is number of mappers, second argument is a list of words. File list was printed only for debugging.

In [13]:
```
!./pNaiveBayes.sh 4 assistance
!head enronemail_1h.txt.output
```

```
assistance      8
```

Confirm answers with bash commands (note there are actually 10 instances of assistance, but only 8 lines):

In [14]:
```
!grep assistance enronemail_1h.txt|cut -d$'\t' -f4| grep assistance|wc
-l
```

```
       8
```

HW1.3. Provide a mapper/reducer pair that, when executed by pNaiveBayes.sh

will classify the email messages by a single, user-specified word using the Naive Bayes Formulation. Examine the word “assistance” and report your results. To do so, make sure that

- mapper.py and
- reducer.py

that performs a single word Naive Bayes classification.

In [31]:
```
%%writefile mapper.py
#!/usr/bin/python
import re
import sys
import os
import numpy as np

# store a regex expression into a pattern object
# that seeks words including underscores and single quotes
```

```
WORD_RE = re.compile(r"[\w']+")
TRUTH_RE = re.compile(r"\t(\d)\t")

# file input
filename = sys.argv[1]

# for this part, just assume word_list is length 1
word_list = sys.argv[2]

# Avoid KeyError if no data in chunk
#counts_dict = dict.fromkeys(['0', '1'], 0)
counts_dict = {}

spam_count = 0
ham_count  = 0

with open(filename, 'rU') as f:
    for line in f.readlines():
        # Remove punctuation
        line = re.sub(r'[^\w\s]','',line)
        truth = TRUTH_RE.findall(line)[0]
        if truth == '1':
            spam_count += 1
        else:
            ham_count += 1
        for category in ['0','1']:
            counts_dict[category] = {}

            for word in word_list.split():

                counts = [1 if x == word else 0 for x in WORD_RE.finda
ll(line)]
                counts = np.array(counts)

                if counts.sum() > 0:
                    count = 1
                    counts_dict[category][word] = counts_dict[category
].get(word, 0) + int(count)
                else:
                    counts_dict[category][word] = 0


for category, word_dictionary in counts_dict.iteritems():
    for words, count in counts_dict[category].iteritems():
        print category + "\t" + words + "\t" + str(count) + "\t" + str
(spam_count) +"\t" + str(ham_count)
```

```
Overwriting mapper.py
```

In [32]:
```
!chmod a+x mapper.py
```

In [37]:
```
%%writefile reducer.py
#!/usr/bin/python

import sys
import numpy as np
import re
from math import log

WORD_RE = re.compile(r"[\w']+")
TRUTH_RE = re.compile(r"\t(\d)\t")
files_list = sys.argv[1].split()

## training, gather all the counts and calculate corpus-wide priors, e
tc
## data come in as strings,
## TRUTH WORD COUNT SPAM_COUNT HAM_COUNT
counts_dict = {}
for category in ['0','1']:
    counts_dict[category] = {}

spam_total = 0
ham_total  = 0

for f in files_list:
    with open(f, 'rU') as countfile:
        for line in countfile.readlines():
            truth, word, count, spam_count, ham_count = line.split('\t
')
            counts_dict[truth][word] = counts_dict[truth].get(word, 0)
+ int(count)
        spam_total += int(spam_count)
        ham_total  += int(ham_count)

priors = {'0': float(ham_total)/(spam_total+ham_total),
          '1': float(spam_total)/(spam_total+ham_total)}

prior_counts = {'0': float(ham_total),
          '1': float(spam_total)}

print "Priors are: "
for category in priors:
    print category + " " + str(priors[category]) + "\n"

spam_vocab = counts_dict['1'].keys()
ham_vocab  = counts_dict['0'].keys()

spam_vocab_n = len(counts_dict['1'].keys())
```

```
ham_vocab_n  = len(counts_dict['0'].keys())

## although we are not implementing Laplace Transform
# probably don't need these next two lines
#vocab = union(spam_vocab, ham_vocab)
#vocab_n = len(vocab)

## Calculate conditional probabilities
## P(word | class)
posteriors = {}
for category in ['0', '1']:
    posteriors[category] = {}
    for word in counts_dict[category].keys():
        posteriors[category][word] = float(counts_dict[category][word]
)/prior_counts[category]

print "\nPosteriors are: "
for category in posteriors:
    for word in posteriors[category]:
        print word + " in class " + category + " " + str(posteriors[ca
tegory][word]) + "\n"

## Testing the classifer
## Without laplacian transform
print "DOC_ID | TRUTH | CLASS "
print "======================\n"

doc_id = 0
correct = 0
with open("enronemail_1h.txt", 'rU') as testdata:
    for line in testdata.readlines():
        score = [0,0]
        line = re.sub(r'[^\w\s]','',line)
        truth = TRUTH_RE.findall(line)[0]

        for category in ['0', '1']:
            idx = int(category)
            score[idx] = log(priors[category])
            #score[idx] =  priors[category]
            for word in posteriors[category]:
                if word in WORD_RE.findall(line):
                    score[idx] += log(float(posteriors[category][word]
)+1)
                    #score[idx] *= float(posteriors[category][word])
                #print "\n", idx, score[idx], category, word
        score = np.array(score)
        prediction = score.argmax()
        doc_id +=1

        if int(prediction) == int(truth):
```

```
            correct +=1
        print str(doc_id) + "\t" + truth + "\t" +str(prediction) +  "
" +str(score[0]) + " " + str(score[1])

accuracy = float(correct)/doc_id*100.0
print "Accuracy: ", accuracy
```

```
Overwriting reducer.py
```

In [38]:
```
!chmod a+x reducer.py
```

In [39]:
```
!./pNaiveBayes.sh 4 assistance
!cat enronemail_1h.txt.output
```

```
Priors are:
1 0.44

0 0.56


Posteriors are:
assistance in class 1 0.0227272727273

assistance in class 0 0.0178571428571

DOC_ID | TRUTH | CLASS
=======================

1       0       0 -0.579818495253 -0.82098055207
2       0       0 -0.579818495253 -0.82098055207
3       0       0 -0.579818495253 -0.82098055207
4       0       0 -0.579818495253 -0.82098055207
5       0       0 -0.579818495253 -0.82098055207
6       0       0 -0.579818495253 -0.82098055207
7       0       0 -0.579818495253 -0.82098055207
8       0       0 -0.579818495253 -0.82098055207
9       1       0 -0.579818495253 -0.82098055207
10      1       0 -0.579818495253 -0.82098055207
11      1       0 -0.562118918154 -0.798507696218
12      0       0 -0.579818495253 -0.82098055207
13      0       0 -0.579818495253 -0.82098055207
14      0       0 -0.579818495253 -0.82098055207
15      0       0 -0.579818495253 -0.82098055207
16      1       0 -0.579818495253 -0.82098055207
17      1       0 -0.579818495253 -0.82098055207
18      0       0 -0.562118918154 -0.798507696218
19      0       0 -0.579818495253 -0.82098055207
20      0       0 -0.579818495253 -0.82098055207
21      1       0 -0.579818495253 -0.82098055207
22      1       0 -0.579818495253 -0.82098055207
```

23	0	0	-0.562118918154	-0.798507696218
24	0	0	-0.579818495253	-0.82098055207
25	0	0	-0.579818495253	-0.82098055207
26	0	0	-0.579818495253	-0.82098055207
27	1	0	-0.579818495253	-0.82098055207
28	1	0	-0.579818495253	-0.82098055207
29	0	0	-0.579818495253	-0.82098055207
30	0	0	-0.579818495253	-0.82098055207
31	0	0	-0.579818495253	-0.82098055207
32	1	0	-0.579818495253	-0.82098055207
33	1	0	-0.579818495253	-0.82098055207
34	1	0	-0.579818495253	-0.82098055207
35	0	0	-0.579818495253	-0.82098055207
36	0	0	-0.579818495253	-0.82098055207
37	0	0	-0.579818495253	-0.82098055207
38	0	0	-0.579818495253	-0.82098055207
39	1	0	-0.579818495253	-0.82098055207
40	1	0	-0.579818495253	-0.82098055207
41	0	0	-0.579818495253	-0.82098055207
42	1	0	-0.579818495253	-0.82098055207
43	1	0	-0.579818495253	-0.82098055207
44	1	0	-0.579818495253	-0.82098055207
45	1	0	-0.579818495253	-0.82098055207
46	0	0	-0.579818495253	-0.82098055207
47	0	0	-0.579818495253	-0.82098055207
48	0	0	-0.579818495253	-0.82098055207
49	0	0	-0.579818495253	-0.82098055207
50	1	0	-0.579818495253	-0.82098055207
51	1	0	-0.579818495253	-0.82098055207
52	0	0	-0.579818495253	-0.82098055207
53	0	0	-0.579818495253	-0.82098055207
54	0	0	-0.579818495253	-0.82098055207
55	1	0	-0.562118918154	-0.798507696218
56	1	0	-0.579818495253	-0.82098055207
57	1	0	-0.579818495253	-0.82098055207
58	0	0	-0.579818495253	-0.82098055207
59	1	0	-0.562118918154	-0.798507696218
60	1	0	-0.579818495253	-0.82098055207
61	1	0	-0.579818495253	-0.82098055207
62	1	0	-0.579818495253	-0.82098055207
63	0	0	-0.579818495253	-0.82098055207
64	0	0	-0.579818495253	-0.82098055207
65	0	0	-0.579818495253	-0.82098055207
66	0	0	-0.579818495253	-0.82098055207
67	0	0	-0.579818495253	-0.82098055207
68	1	0	-0.579818495253	-0.82098055207
69	0	0	-0.579818495253	-0.82098055207
70	0	0	-0.579818495253	-0.82098055207
71	0	0	-0.579818495253	-0.82098055207
72	1	0	-0.579818495253	-0.82098055207

```
73      1       0 -0.562118918154 -0.798507696218
74      0       0 -0.579818495253 -0.82098055207
75      0       0 -0.579818495253 -0.82098055207
76      0       0 -0.579818495253 -0.82098055207
77      1       0 -0.579818495253 -0.82098055207
78      1       0 -0.579818495253 -0.82098055207
79      1       0 -0.579818495253 -0.82098055207
80      0       0 -0.579818495253 -0.82098055207
81      0       0 -0.579818495253 -0.82098055207
82      0       0 -0.579818495253 -0.82098055207
83      0       0 -0.579818495253 -0.82098055207
84      1       0 -0.579818495253 -0.82098055207
85      1       0 -0.579818495253 -0.82098055207
86      0       0 -0.579818495253 -0.82098055207
87      0       0 -0.579818495253 -0.82098055207
88      1       0 -0.579818495253 -0.82098055207
89      1       0 -0.579818495253 -0.82098055207
90      1       0 -0.579818495253 -0.82098055207
91      1       0 -0.579818495253 -0.82098055207
92      0       0 -0.579818495253 -0.82098055207
93      0       0 -0.579818495253 -0.82098055207
94      0       0 -0.579818495253 -0.82098055207
95      1       0 -0.579818495253 -0.82098055207
96      1       0 -0.579818495253 -0.82098055207
97      1       0 -0.579818495253 -0.82098055207
98      0       0 -0.579818495253 -0.82098055207
99      1       0 -0.562118918154 -0.798507696218
100     1       0 -0.562118918154 -0.798507696218
Accuracy:  56.0
```

HW1.4. Provide a mapper/reducer pair that, when executed by pNaiveBayes.sh

will classify the email messages by a list of one or more user-specified words. Examine the words “assistance”, “valium”, and “enlargementWithATypo” and report your results To do so, make sure that

- mapper.py counts all occurrences of a list of words, and
- reducer.py

  performs the multiple-word Naive Bayes classification via the chosen list.

In [40]:
```
# 1.4

!./pNaiveBayes.sh 3 "assistance valium enlargementWithATypo"
!cat enronemail_1h.txt.output
```

```
Priors are:
```

```
1 0.44

0 0.56


Posteriors are:
assistance in class 1 0.0227272727273

enlargementWithATypo in class 1 0.0

valium in class 1 0.0

assistance in class 0 0.0178571428571

enlargementWithATypo in class 0 0.0

valium in class 0 0.0

DOC_ID | TRUTH | CLASS
=======================

1       0       0 -0.579818495253 -0.82098055207
2       0       0 -0.579818495253 -0.82098055207
3       0       0 -0.579818495253 -0.82098055207
4       0       0 -0.579818495253 -0.82098055207
5       0       0 -0.579818495253 -0.82098055207
6       0       0 -0.579818495253 -0.82098055207
7       0       0 -0.579818495253 -0.82098055207
8       0       0 -0.579818495253 -0.82098055207
9       1       0 -0.579818495253 -0.82098055207
10      1       0 -0.579818495253 -0.82098055207
11      1       0 -0.562118918154 -0.798507696218
12      0       0 -0.579818495253 -0.82098055207
13      0       0 -0.579818495253 -0.82098055207
14      0       0 -0.579818495253 -0.82098055207
15      0       0 -0.579818495253 -0.82098055207
16      1       0 -0.579818495253 -0.82098055207
17      1       0 -0.579818495253 -0.82098055207
18      0       0 -0.562118918154 -0.798507696218
19      0       0 -0.579818495253 -0.82098055207
20      0       0 -0.579818495253 -0.82098055207
21      1       0 -0.579818495253 -0.82098055207
22      1       0 -0.579818495253 -0.82098055207
23      0       0 -0.562118918154 -0.798507696218
24      0       0 -0.579818495253 -0.82098055207
25      0       0 -0.579818495253 -0.82098055207
26      0       0 -0.579818495253 -0.82098055207
27      1       0 -0.579818495253 -0.82098055207
28      1       0 -0.579818495253 -0.82098055207
29      0       0 -0.579818495253 -0.82098055207
```

30	0	0	-0.579818495253	-0.82098055207
31	0	0	-0.579818495253	-0.82098055207
32	1	0	-0.579818495253	-0.82098055207
33	1	0	-0.579818495253	-0.82098055207
34	1	0	-0.579818495253	-0.82098055207
35	0	0	-0.579818495253	-0.82098055207
36	0	0	-0.579818495253	-0.82098055207
37	0	0	-0.579818495253	-0.82098055207
38	0	0	-0.579818495253	-0.82098055207
39	1	0	-0.579818495253	-0.82098055207
40	1	0	-0.579818495253	-0.82098055207
41	0	0	-0.579818495253	-0.82098055207
42	1	0	-0.579818495253	-0.82098055207
43	1	0	-0.579818495253	-0.82098055207
44	1	0	-0.579818495253	-0.82098055207
45	1	0	-0.579818495253	-0.82098055207
46	0	0	-0.579818495253	-0.82098055207
47	0	0	-0.579818495253	-0.82098055207
48	0	0	-0.579818495253	-0.82098055207
49	0	0	-0.579818495253	-0.82098055207
50	1	0	-0.579818495253	-0.82098055207
51	1	0	-0.579818495253	-0.82098055207
52	0	0	-0.579818495253	-0.82098055207
53	0	0	-0.579818495253	-0.82098055207
54	0	0	-0.579818495253	-0.82098055207
55	1	0	-0.562118918154	-0.798507696218
56	1	0	-0.579818495253	-0.82098055207
57	1	0	-0.579818495253	-0.82098055207
58	0	0	-0.579818495253	-0.82098055207
59	1	0	-0.562118918154	-0.798507696218
60	1	0	-0.579818495253	-0.82098055207
61	1	0	-0.579818495253	-0.82098055207
62	1	0	-0.579818495253	-0.82098055207
63	0	0	-0.579818495253	-0.82098055207
64	0	0	-0.579818495253	-0.82098055207
65	0	0	-0.579818495253	-0.82098055207
66	0	0	-0.579818495253	-0.82098055207
67	0	0	-0.579818495253	-0.82098055207
68	1	0	-0.579818495253	-0.82098055207
69	0	0	-0.579818495253	-0.82098055207
70	0	0	-0.579818495253	-0.82098055207
71	0	0	-0.579818495253	-0.82098055207
72	1	0	-0.579818495253	-0.82098055207
73	1	0	-0.562118918154	-0.798507696218
74	0	0	-0.579818495253	-0.82098055207
75	0	0	-0.579818495253	-0.82098055207
76	0	0	-0.579818495253	-0.82098055207
77	1	0	-0.579818495253	-0.82098055207
78	1	0	-0.579818495253	-0.82098055207
79	1	0	-0.579818495253	-0.82098055207

```
80      0       0 -0.579818495253 -0.82098055207
81      0       0 -0.579818495253 -0.82098055207
82      0       0 -0.579818495253 -0.82098055207
83      0       0 -0.579818495253 -0.82098055207
84      1       0 -0.579818495253 -0.82098055207
85      1       0 -0.579818495253 -0.82098055207
86      0       0 -0.579818495253 -0.82098055207
87      0       0 -0.579818495253 -0.82098055207
88      1       0 -0.579818495253 -0.82098055207
89      1       0 -0.579818495253 -0.82098055207
90      1       0 -0.579818495253 -0.82098055207
91      1       0 -0.579818495253 -0.82098055207
92      0       0 -0.579818495253 -0.82098055207
93      0       0 -0.579818495253 -0.82098055207
94      0       0 -0.579818495253 -0.82098055207
95      1       0 -0.579818495253 -0.82098055207
96      1       0 -0.579818495253 -0.82098055207
97      1       0 -0.579818495253 -0.82098055207
98      0       0 -0.579818495253 -0.82098055207
99      1       0 -0.562118918154 -0.798507696218
100     1       0 -0.562118918154 -0.798507696218
Accuracy:  56.0
```

After looking through the predictions for 1.3 and 1.4 more closely, it appears I wasn't able to make any positive predictions of spam. It is very likely that laplacian smoothing would greatly improve the accuracy as our vocabulary is extremely sparse compared to the entire vocabulary of the corpus. Furthermore, it appears that for reasons I wasn't able to figure out, I was unable to accurately count the lines that contained "valium". This is probably why the prediction for 1.4 is practically identical to 1.3.

In [ ]: