Homework #1

Student: Alex Smith
Course: W261 - Machine Learning at Scale
Professor: Jimi Shanahan
Due Date: May 17

Useful resources

When completing this notebook, I found the following resources particularly useful:

- 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/)
- Bayes' Theorem: statement of theorem (https://en.wikipedia.org/wiki/Bayes%27_theorem#Statement_of_theorem)
- An introduction to information retrieval, chapter 13 (http://nlp.stanford.edu/IR-book/pdf/irbookonlinereading.pdf)
- SKLearn's Explanation for Naive Bayes (http://scikit-learn.org/stable/modules/naive_bayes.html)

Libraries

The following libraries must be installed before running the below code. They can all be installed through Pip (https://github.com/pypa/pip).

- Scikit Learn (http://scikit-learn.org/stable/)
- Numpy (http://www.numpy.org/)
- Regular Expression (https://docs.python.org/2/library/re.html)
- Pretty Table (https://pypi.python.org/pypi/PrettyTable)

HW0.0

Prepare your bio and include it in this HW submission. Please limit to 100 words. Count the words in your bio and print the length of your bio (in terms of words) in a separate cell.

After completing my undergrad in Middle Eastern Studies, I moved to Atlanta to be part of a management program at an industrial supply company, McMaster-Carr. I'm currently the Billings and control manager for the branch. I had never really analyzed data before starting work at McMaster. At McMaster, I got to do some really fun data analysis and build some really nifty models in Excel (more business intelligence than data science). I realized that I find playing around with data to be pretty fun. I decided that I want to explore this more and found this program.

Function to count the number of words in my bio

In [1]:
```
def bioWordCount(text,limit=100):
    """a word count function that counts the
    number of words in the inputed text and
    compares it a limit, by default set to 100"""

    # calculate the length of my bio by
    # splitting the words using python's native split
    # function which splits by spaces
    text_length = len(text.split())

    # if the text length is greater than the
    # number of words allowed, warn the user
    # and print out the length of the text
    if text_length > limit:

        return "Uh-oh, my bio has more words than allowed. \
        My bio has " + str(text_length) + " words, but \
        I am only allowed " + str(limit) + " words."

    # else if the text length is less than
    # or equal to the limit, tell the user how
    # many they words they used
    else:

        return "The length of my bio is " + \
    str(text_length) + " which is " + \
    str(limit-text_length) + \
        " less than the bio limit of " + str(limit)
```

In [2]:
```
# write my bio
bio_my = """After completing my undergrad in Middle Eastern Studie
s,
I moved to Atlanta to be part of a management program at an
industrial supply company, McMaster-Carr. I'm currently the Billing
s
and control manager for the branch. I had never really analyzed
data before starting work at McMaster. At McMaster, I got to
do some really fun data analysis and build some really nifty
models in Excel (more business intelligence than data
science). I realized that I find playing around with data
to be pretty fun. I decided that I want to explore this
more and found this program."""

# print out my word count using the function defined in the cell ab
ove
print bioWordCount(bio_my)
```

```
The length of my bio is 97 which is 3 less than the bio limit of 1
00
```

HW1.0.0.

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

A useful framework to think about big data is the 3 V's: volume, velocity, and variety. In layman's terms, this means data is huge, comes very quickly, and is composed of different types. To define these V's more specifically, it is useful to think of a standard laptop with 1 TB hard drive and 8-16 GB of memory. Big data would be data for which it would simply be impractical to use a single, standard laptop.

1. **Volume**: We cannnot hold more than 1 TB of data on the drive, and so analyzing anything larger than this becomes an impossibility. Rather than thinking about terabytes, we think of big data in zettabytes.
2. **Velocity**: To read a whole terabyte of data on our standard laptop would require three hours. This makes it impractical to analyze data generated in real time. Rather than thinking about data as batches or dumps, big data requires us to think of streaming data that must be analyzed in real time.
3. **Variety**: Previous to concept of big data, we might think of data as a table. All of the data would be homogeneous and standardized. Big data instead requires us to think about data as a compliation of different types of information. For example, we might combine texts of tweets with geolocation with pictures posted to facebook at that same location.

As a side note, IBM also mentions a fourth V of big data: **veracity**. Big data analysis must sometimes deal with data of suspect quality. A big data analyst must make decisions on how to handle data of poor quality (e.g. missing records, inaccuracies).

HW1.0.1. Bias Variance

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?

Any model is going to two types of error, **reducible** and **irreducible** error, when attempting to estimate some true funtion. We can think of the **irreducible error** as the error that arise from noise that surrounds that surrounds the true function. As its name implies, we cannot minimize or remove this error because the noise varies randomly from the true function. In the image below, we can see the true function as the solid black line and the points generated with random noise as the white circles close to the line.

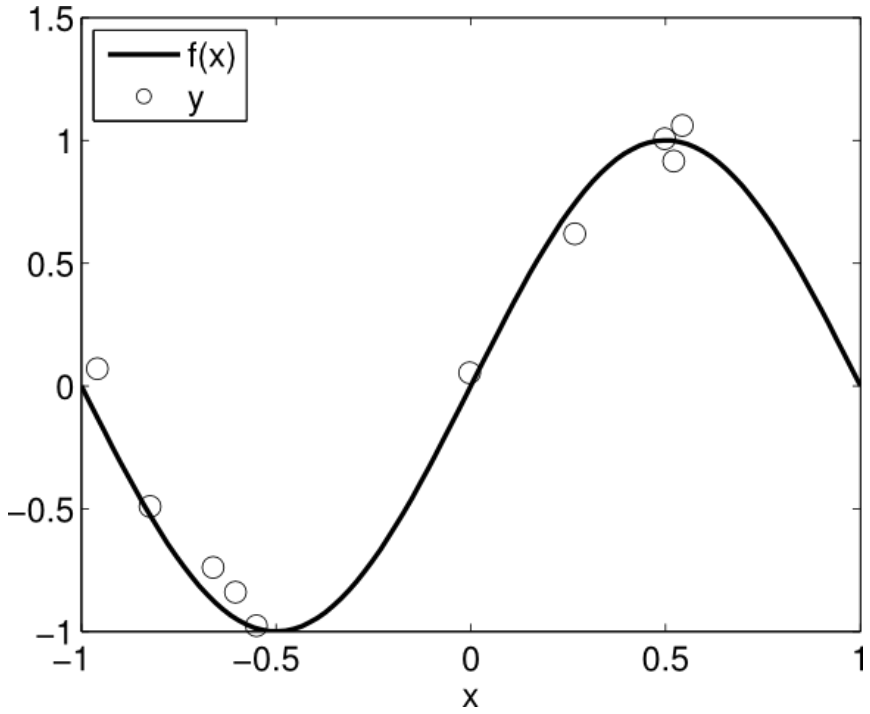

Image source: dustinstansbury. "Model Selection: Underfitting, Overfitting, and the Bias-Variance Tradeoff." The Clever Machine. (21 April 2013)

On the other hand, we have **reducible error**. This type of error we want to minimize. Reducible error is further broken up into bias and variance. We can think of **bias** as the difference between the estimate of a model and the true value. For example, a model will have higher bias if the estimates it produces are further from the true values than another model. We measure the bias as the *error due to squared bias* which is the difference between the model's predicted value and the true value over the training data. In the image below, we can see the model as the red lines and the true function as the black line. For most points, these models have very high bias (the red lines are not close to the true values demarcted by the black line).

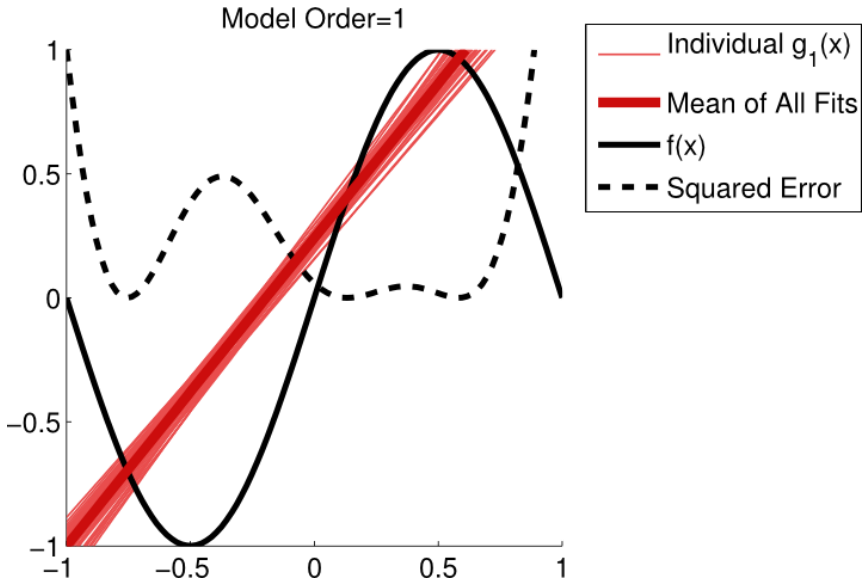

Image source: dustinstansbury, 2013.

The second part of the reducible error is the **variance**. We can think of variance as the difference that results from running multiple models from samples of the same data. A model would have higher variance if multiple iterations of the model over different samples of the training data are widely different. We measure the variance as the *error due to variance* which is amount by which the prediction from one training sets differs from the predictions of all other training sets. In the image below, we can see the model as purple lines and the true function as the black line. The lines are very inconsistent and appear to wildly fluctuate to match the specific sample of training data that was pulled. This is a clear case of *overfitting* the model to a particular training set.

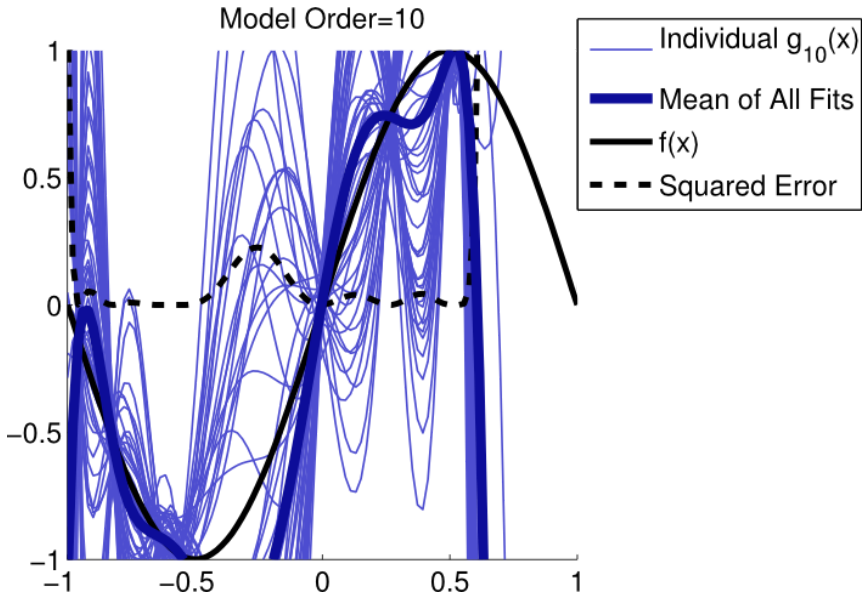

Image source: dustinstansbury, 2013.

When using polynomial models of increasing degrees, I would divide up my data in multiple training sets and a test set. I would then run multiple iterations of each polynomial degree model on the training sets. I would test each model on the test sets and calculate the error due to squared bias and the error due to variance as defined in the preceding paragraphs. I would expect to find that increasing the complexity of

my model increases my error due to variance but decreases my error due to squared bias. I would choose the model with the least total error. For an example plot that shows the relationship between the error due to variance and the error due to squared bias with the model complexity, see the image below:

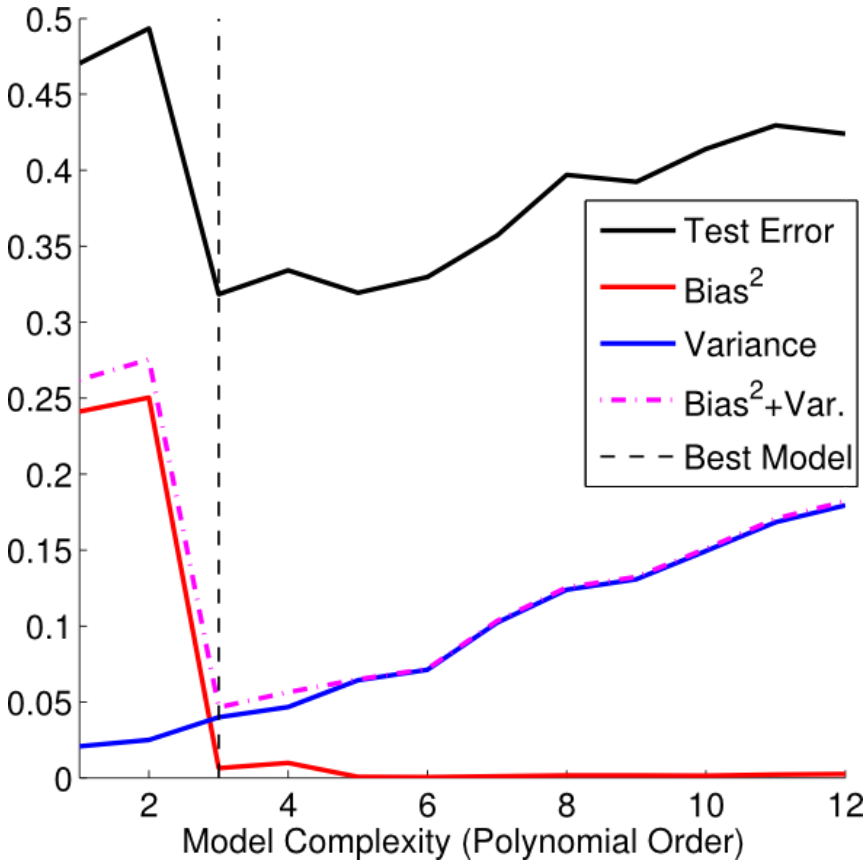

Image source: dustinstansbury, 2013.

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" (print "done") string will suffice here. (dont forget to include the Question Number and the question in the cell as a markdown multiline comment!)

In [3]:
```
print "Done"
```

```
Done
```

Write the shell script that runs our mappers and reducers

Author: Jake Williams

```
%%writefile pNaiveBayes.sh
## 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 command,
##               and focus on how arguments are
##               supplied to mapper.py/reducer.py,
##               as this will determine how the python
##               scripts take input.
##               When you are comfortable with the
##               unix code below,
##               answer the questions on the LMS for
##               HW1 about the starter 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 STDOUT to disk
####
####
./reducer.py $countfiles > $data.output
####
####

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

```
Overwriting pNaiveBayes.sh
```

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
"assistance" occurs on 8 lines but how many times does the token occur? 10 times! This is the number we are looking for!

Here, mapper.py will read in a portion (i.e., a single record corresponding to a row) of the email data, count the number of occurences of the word in question and print/emit a count to the output stream. While the utility of the reducer responsible for reading in counts of the word and summarizing them before printing that summary to the output stream.

See example in:
http://nbviewer.ipython.org/urls/dl.dropbox.com/s/ujz9w7d2a73b80o/DivideAndConquer2-python-Incomplete.ipynb (http://nbviewer.ipython.org/urls/dl.dropbox.com/s/ujz9w7d2a73b80o/DivideAndConquer2-python-Incomplete.ipynb) See video section 1.12.1 1.12.1 Poor Man's MapReduce Using Command Line (Part 2) located at: https://learn.datascience.berkeley.edu/mod/page/view.php?id=10961 (https://learn.datascience.berkeley.edu/mod/page/view.php?id=10961) NOTE in your python notebook create a cell to save your mapper/reducer to disk using magic commands (see example here)

Mapper function

This function takes a text and given word and counts the occurences of that given word in the text.

In [5]:
```
%%writefile mapper.py
#!/usr/bin/python
## mapper.py
## Author: Alex Smith
## Description: mapper code for HW1.2

# import our libraries to use regular expression and
# read from the system arguments; initalize the count
import sys
import re
count = 0

# collect user input
filename = sys.argv[1]
findwords = re.split(" ",sys.argv[2].lower())

# create a regex that matches alphanumeric characters
wordify = re.compile(r"[\w']+")

# open the file as readable
with open (filename, "r") as myfile:

    # loop through each line in the file
    for line in myfile.readlines():

        # convert the line into a list of words without punctuation
        # also lowercase all the words
        words = wordify.findall(line.lower())

        # loop through each word in each line
        for word in words:

            # if the word equals the word we're searching for
            # increment the counter by one
            if word == findwords[0]:
                count = count + 1

print findwords[0], "\t", count
```

```
Overwriting mapper.py
```

Reducer function

This function combines the multiple counts generated by the mapper function into a single count for all occurences of a given word in the entire text.

In [6]:
```
%%writefile reducer.py
#!/usr/bin/python
## reducer.py
## Author: Alex Smith
## Description: reducer code for HW1.2

# import the system libraries to read from the input and
# initalize the sum variable and the find word
import sys
sums = 0
findword = ""

# grab and store the count files
count_files = sys.argv[1:]

# loop through each file in the list of count files
for filename in count_files:

    # open the file as readable
    with open (filename, "r") as myfile:

        # loop through each line in the file
        for line in myfile.readlines():

            # split the line by tabs and take the
            # second element (the count)
            line = line.split("\t")

            # add the value found by each chunk to the sum
            sums = sums + int(line[1])

            # if the find word has not been set yet, set it
            if findword == "":
                findword = line[0]

print findword, "\t", sums
```

```
Overwriting reducer.py
```

Magic functions to set the appropriate permissions

In [7]:
```
# set the right permissions for the python files mapper and reducer
!chmod +x mapper.py; chmod +x reducer.py
```

In [8]:
```
# modifies the permission to make the shell
# command executable from the notebook
!chmod a+x pNaiveBayes.sh
```

Run our counter and print the output

In [9]:
```
# run the Naive Bayes counter with the word assistance
!./pNaiveBayes.sh 2 "assistance"

# grab the output file and open it
filename = "enronemail_1h.txt.output"
with open(filename, "r") as myfile:

    # pull out the count
    word_count = myfile.read().split()[1]

# print out our solution
print "The word assistance occurs %s times." % word_count
```

```
The word assistance occurs 10 times.
```

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 multinomial 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. For multinomial Naive Bayes, the Pr(X="assistance"|Y=SPAM) is calculated as follows:

- *the number of times "assistance" occurs in SPAM labeled documents / the number of words in documents labeled SPAM*

NOTE if "assistance" occurs 5 times in all of the documents Labeled SPAM, and the length in terms of the number of words in all documents labeld as SPAM (when concatenated) is 1,000. Then Pr(X="assistance"|Y=SPAM) = 5/1000. Note this is a multinomial estimated of the class conditional for a Naive Bayes Classifier. No smoothing is needed in this HW problem.

Understanding Naive Bayes Classifier

We want to build a Naive Bayes classifier that predicts the probability that some email is spam given the number of times some word (in this case, "assistance") occurs in the email. So that we can better understand this problem, let's go over some basic Bayesian probability and simplify our terms. Let A be the probability that an email is spam. Let B be the probability that an email has the word, assistance, in it. In essance, we want to find the probability of A given B, P(A|B). Using Bayes' theorem, we know that:

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

We divide this up to see the variables we must calculate:

- P(A|B) = probability of spam given "assistance"
- P(B|A) = probability of "assistance" given spam (calculated as the number of times assistance occurs in spam labeled documents over the number of words in documents labeled spam)
- P(A) = probability of spam (calculated as the number of times an email is labeled spam over the total number of emails)
- P(B) = probability of assistance (calculated as the number of times assistances appears over the total number of words)

Mapper function

The goal is for this function to output the information neccessary for the mapper to build the probabilities for the Naive Bayes classifier. This function loops through each word of each email in the text and outputs:

- email id
- word of email
- spam indicator
- word of interest indicator

```
%%writefile mapper.py
#!/usr/bin/python
## mapper.py
## Author: Alex Smith
## Description: mapper code for HW1.3

# import our libraries to use regular expression and
# read from the system arguments
import sys
import re

# initalize the outputs that we'll be using
email_id = None
word_act = None
spam_ind = None
word_ind = None

# collect user input
filename = sys.argv[1]
findwords = re.split(" ",sys.argv[2].lower())

# create a regex that matches alphanumeric characters
wordify = re.compile(r"[\w']+")

# open the file as readable
with open (filename, "r") as myfile:

    # loop through each line in the file
    for line in myfile.readlines():

        # separate out each line based on the tabs
        line = line.split("\t")

        # set the email id and spam indicator fields
        email_id = line[0]
        spam_ind = line[1]

        # grab the text from the body and subject and concatenate i
t
        text = line[2] + " " + line[3]

        # convert the text into a list of words without punctuation
        # also lowercase all the words
        words = wordify.findall(text.lower())

        # loop through each word
        for word in words:

            # if the word is one of the words we're searching for,
set
            # the word indicator to 1; otherwise, set it to 0
            if word in findwords:
                word_ind = 1
            else:
```

```
            word_ind = 0

        # set the word field
        word_act = word

        # collect the line that we want to print out
        info = email_id + "\t" + word_act + "\t" + \
        str(spam_ind) + "\t" + str(word_ind)

        # let's print each email id, word, spam indicator,
        # and word of interest indicator
        # we'll separate each value with a tab character
        print info
```

```
Overwriting mapper.py
```

Reducer function

This function takes the outputs from the mapper function and uses it to calculate the probabilities for the classifier. It then loops back through the outputs and classifies each email ID as spam or not spam. To calculate the probabilities for the classifier, it gathers the following information:

- number of all words (does not exclude duplicates)
- number of emails
- number of spam emails
- number of times words of interest in appear in both spam and not spam emails
- number of words in emails with the findword

```
%%writefile reducer.py
#!/usr/bin/python
## reducer.py
## Author: Alex Smith
## Description: reducer code for HW1.3

# import the system libraries to read from the input
import sys

# initalize the summary statistic counts
all_words = 0
all_emails = 0
spam_emails = 0

# number of words in spam emails
spam_ewords = 0

# number of times the findword appears in spam emails
findword_spam = 0

# number of times the findword appears in not spam emails
findword_notspam = 0

# create a dictionary to store each email id
# and whether it contains the find word
emails = {}

# grab and store the count files
count_files = sys.argv[1:]

# loop through each file in the list of count files
for filename in count_files:

    # open the file as readable
    with open (filename, "r") as myfile:

        # loop through each line in the file
        for line in myfile.readlines():

            # split the line by tabs
            line = line.split("\t")

            # pull out my values for each part of the line
            email_id = line[0]
            word_act = line[1]
            spam_ind = int(line[2])
            word_ind = int(line[3])

            # let's update the number of words and
            # the spam words (if it's spam)
            all_words = all_words + 1
            if spam_ind == 1:
                spam_ewords = spam_ewords + 1
```

```
            # if it's a findword, let's update the count
            # respective to whether or not it's in
            # a spam email
            if word_ind == 1:
                if spam_ind == 1:
                    findword_spam = findword_spam + 1
                else:
                    findword_notspam = findword_notspam + 1

            # check to see if this email is already
            # in the dictionary, and if
            # its not already there, initalize it
            if email_id not in emails:

                # create a sub-dictionary within the
                # email dictionary that is initalized
                # to not containing the find word
                emails[email_id] = {"spam":spam_ind, "findword":0}

                # if it's not already there, let's
                # also increment the email counter
                # and the spam counter if it's spam
                all_emails = all_emails + 1
                if spam_ind == 1:
                    spam_emails = spam_emails + 1

            # if the word is a findword, update
            # the find word indicator in the dictionary
            if word_ind == 1:
                emails[email_id]["findword"] = 1

# now that we have the summary statistics, we
# can calculate the probability that an email
# is spam given that a find word appears; we
# use the bayesian probability formula from above

# posterior probabilities
prob_spam = float(spam_emails) / float(all_emails)
prob_nspam = 1.0 - prob_spam

# probability of the find word, given the label
# spam is the the number of times the findword
# occurs in spam labeled documents over the
# number of words in documents labeled spam
prob_findGIVspam = float(findword_spam) / float(spam_ewords)

# probability of the find word
prob_findword = float(findword_spam) / \
float(findword_notspam+findword_spam)

# probability of spam given findword using the
# Bayesian probability formula from above
prob_spamGIVfind = (prob_findGIVspam * prob_spam) / prob_findword
```

```
# we use the same bayesian probability formula to
# calculate the probability of not spam
# given the find word
prob_findGIVnspam = float(findword_notspam) / \
float(all_words - spam_ewords)

prob_nspamGIVfind = (prob_findGIVnspam * prob_nspam) / prob_findwor
d

# now let's loop through each email in the
# dictionary and classify it as spam or not spam
for email in emails.keys():

    # get the actual classification
    truth = emails[email]["spam"]

    # if the email has no find word, then
    # set the prediction based on the posterior
    # probabilities
    if emails[email]["findword"] == 0:
        if(prob_spam > prob_nspam):
            _prediction = 1
        else:
            _prediction = 0

    # else if the email has a find word, then set
    # the prediction based on the conditional probability
    else:
        if(prob_spamGIVfind > prob_nspamGIVfind):
            _prediction = 1
        else:
            _prediction = 0

    # print the output as the email id,
    # the actual classification, the prediction
    # as a tab-delimited line
    info = email + "\t" + str(truth) + "\t" + str(_prediction)
    print info
```

```
Overwriting reducer.py
```

Magic functions to set the appropriate permissions

In [12]:
```
# set the right permissions for the python
# files mapper and reducer
!chmod +x mapper.py; chmod +x reducer.py
```

In [13]:
```
# modifies the permission to make the
# shell command executable from the notebook
!chmod a+x pNaiveBayes.sh
```

Run our model and print the output

In [14]:
```
# run the Naive Bayes counter with the word assistance
!./pNaiveBayes.sh 3 "assistance"

print "Done. Check the file enronemail_1h.txt.output"
```

```
Done. Check the file enronemail_1h.txt.output
```

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 (accuracy) To do so, make sure that

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

performs the multiple-word multinomial Naive Bayes classification via the chosen list. No smoothing is needed in this HW problem.

Mapper function

The goal of this function is to spit out the occurences for each word. The input will be a chunk of email messages. The output will be a tab delimited file:

- email id
- spam indicator
- word
- word of interest indicator

We modify our mapper from 1.3 to print the words of interest at the top of each output file.

```
%%writefile mapper.py
#!/usr/bin/python
## mapper.py
## Author: Alex Smith
## Description: mapper code for HW1.4

# import our libraries to use regular expression and
# read from the system arguments
import sys
import re

# initalize the outputs that we'll be using
email_id = None
word_act = None
spam_ind = None
word_ind = None

# collect user input
filename = sys.argv[1]
findwords = sys.argv[2:]

# let's lowercase the find words
new_findwords = []
for findword in findwords:
    new_findwords.append(findword.lower())
findwords = new_findwords

# create a regex that matches alphanumeric characters
wordify = re.compile(r"[\w']+")

# print the words of interest as the first line
words_of_interest=""
for word in findwords:
    words_of_interest = words_of_interest + word + "\t"
print words_of_interest

# open the file as readable
with open (filename, "r") as myfile:

    # loop through each line in the file
    for line in myfile.readlines()[1:]:

        # separate out each line based on the tabs
        line = line.split("\t")

        # set the email id and spam indicator fields
        email_id = line[0]
        spam_ind = line[1]

        # grab the text from the body and subject and concatenate i
t
        text = line[2] + " " + line[3]

        # convert the text into a list of words without punctuation
```

```
        # also lowercase all the words
        words = wordify.findall(text.lower())

        # loop through each word
        for word in words:

            # if the word is one of the words we're searching for,
set
            # the word indicator to 1; otherwise, set it to 0
            if word in findwords:
                word_ind = 1
            else:
                word_ind = 0

            # set the word field
            word_act = word

            # collect the line that we want to print out
            info = email_id + "\t" + word_act + "\t" + \
            str(spam_ind) + "\t" + str(word_ind)

            # let's print each email id, word, spam
            # indicator, and word of interest indicator
            # we'll separate each value with a tab character
            print info
```

```
Overwriting mapper.py
```

Reducer function

This function takes the outputs from the mapper and combines them to build a Naive Bayes classifier. It then runs the Naive Bayes classifier on the training data. We modify this from 1.3 because the reducer in 1.3 is not generalizable enough. It cannot take into account multiple words of interest.

```
%%writefile reducer.py
#!/usr/bin/python
## reducer.py
## Author: Alex Smith
## Description: reducer code for HW1.4

# import the system libraries to read from the input, from the
# math library import the log function
import sys
from math import log

# great a dictionary to store each email id, spam indciator,
# list of words of interest, and spam prediction
emails = {}

# grab and store the count files
count_files = sys.argv[1:]

# let's grab the find words from the first file
with open (count_files[0], "r") as myfile:
    findwords = myfile.readlines()[0].split("\t")
# we also need to remove the new line character that populates
if '\n' in findwords: findwords.remove('\n')

# initalize the summary statistic counts
all_words = 0
all_emails = 0
spam_emails = 0
spam_ewords = 0 # number of words in spam emails

# let's initalize a dictionary for the find words
# each find word will have it's own dictionary with the counts
# of how many times it appears in spam emails and how many in
# not spam emails
findword_prob = {}
for findword in findwords:
    findword_prob[findword] = {"spam_count":0,"not_spam_count":0}

# loop through each file in the list of count files
for filename in count_files:

    # open the file as readable
    with open (filename, "r") as myfile:

        # loop through each line in the file,
        # we ignore the first line because
        # that is the line with the find words
        for line in myfile.readlines()[1:]:

            # split the line by tabs
            line = line.split("\t")

            # pull out my values for each part of the line
            email_id = line[0]
```

```
            word_act = line[1]
            spam_ind = int(line[2])
            word_ind = int(line[3])

            # let's update the number of words
            # and the spam words (if it's spam)
            all_words = all_words + 1
            if spam_ind == 1:
                spam_ewords = spam_ewords + 1

            # if it's a findword, let's update
            # the count respective to whether
            # or not it's in a spam email
            if word_ind == 1:

                # let's grab the words dictionary
                word_dict = findword_prob[word_act]

                if spam_ind == 1:
                    word_dict["spam_count"] = \
                    word_dict["spam_count"] + 1
                else:
                    word_dict["not_spam_count"] = \
                    word_dict["not_spam_count"] + 1

            # check to see if this email is already
            # in the dictionary, and if
            # its not already there, initalize it
            if email_id not in emails:

                # create a sub-dictionary within
                # the email dictionary for each email
                emails[email_id] = {"spam":spam_ind, "words":[]}

                # if it's not already there, let's
                # also increment the email counter
                # and the spam counter if it's spam
                all_emails = all_emails + 1
                if spam_ind == 1:
                    spam_emails = spam_emails + 1

            # if the word is a findword, add
            # it to the list of find words
            if word_ind == 1:
                emails[email_id]["words"].append(word_act)

# now that we have the summary statistics,
# we can calculate the probability that an email
# is spam given that a find word appears;
# we use the bayesian probability formula from above

# posterior probabilities
prob_spam = float(spam_emails) / float(all_emails)
prob_nspam = 1.0 - prob_spam
```

```
# for each find word, let's calculate the
# conditional probability of spam given the find word
# and not spam given the find word

# look at each findword
for findword in findword_prob.keys():

    # set the find word that we'll be calculating
    # the probabilities for
    findword = findword_prob[findword]

    # calculate the probability of the word
    findword['total'] = findword['spam_count'] + \
    findword['not_spam_count']
    findword['probs'] = float(findword['total']) / \
    float(all_words)

    # calculate the probability of word given spam
    findword['wordGIVspam'] = float(findword['spam_count']) \
    / float(spam_ewords)

    # calculate the probability of the word given not spam
    findword['wordGIVnspam'] = \
    float(findword['not_spam_count']) / \
    float(all_words-spam_ewords)

    # calculate the probability of spam given the
    # word (here, we use the probability formula
    # provided in the explanation of the Naive Bayes
    # classifier from Section 1.2)
    findword['spamGIVword'] = \
    (float(findword['wordGIVspam']) * float(prob_spam)) \
    / float(findword['probs'])

    # calculate the probability of not spam given the word
    findword['nspamGIVword'] = \
    (float(findword['wordGIVnspam']) * float(prob_nspam)) \
    / float(findword['probs'])

# now let's loop through each email in
# the dictionary and classify it as spam or not
# spam
for email in emails.keys():

    # get the actual classification
    truth = emails[email]["spam"]

    # if the email has no find word, then set the
    # prediction based on the posterior
    # probabilities
    if len(emails[email]["words"]) == 0:
        if(prob_spam > prob_nspam):
            _prediction = 1
```

```
        else:
            _prediction = 0

    # else if the email has a find word,
    # then set the prediction based on the
    # conditional probability
    else:

        # initalize a set of spam and not
        # spam probabilities
        spam_prob = 1
        nspam_prob = 1

        # loop through each of the find words
        # in the list of words and
        # update the spam and not spam probabilities
        for word in emails[email]['words']:

            spam_prob = spam_prob * \
            findword_prob[word]['spamGIVword']
            nspam_prob = nspam_prob * \
            findword_prob[word]['nspamGIVword']

        if(spam_prob > nspam_prob):
            _prediction = 1
        else:
            _prediction = 0

    # print the output as the email id, the
    # actual classification, the prediction
    # as a tab-delimited line
    info = email + "\t" + str(truth) + "\t" + str(_prediction)
    print info
```

```
Overwriting reducer.py
```

In [17]:
```
# set the right permissions for the python
# files mapper and reducer
!chmod +x mapper.py; chmod +x reducer.py
```

In [18]:
```
# modifies the permission to make the shell
# command executable from the notebook
!chmod a+x pNaiveBayes.sh
```

Run our model and print the output

In [19]:
```
# run the Naive Bayes counter with the word assistance
!./pNaiveBayes.sh 3 assistance valium enlargementWithATypo

print "Done. Check the file enronemail_1h.txt.output"
```

```
Done. Check the file enronemail_1h.txt.output
```

HW1.5.

Provide a mapper/reducer pair that, when executed by pNaiveBayes.sh will classify the email messages by all words present. To do so, make sure that

- *mapper.py counts all occurrences of all words, and*
- *reducer.py performs a word-distribution-wide Naive Bayes classification.*

In this problem you should apply a Laplace (add-1) smoothing to the classifier (always on the reducer side) to safeguard code against low-data.
For a quick reference on the construction of the classifier that you will code, please consult the "Document Classification" section of the following wikipedia page:
https://en.wikipedia.org/wiki/Naive_Bayes_classifier#Document_classification (https://en.wikipedia.org/wiki/Naive_Bayes_classifier#Document_classification)
the original paper by our curators of the Enron email data:
http://www.aueb.gr/users/ion/docs/ceas2006_paper.pdf (http://www.aueb.gr/users/ion/docs/ceas2006_paper.pdf)
or the recording of this week's live lecture that you will find on the LMS.

Error function

We write a function to calculate the training error of our model. We define the training error as the number of incorrectly classified records over the total number of records.

In [20]:
```
def trainingerror(class_file):
    """a function that takes a tab delimited
    classification file with 3 entries per row
    that correspond to: record id, true class,
    and predicted class"""

    # initalize some counters so that we can
    # keep track of how many are wrong and
    # and how many total records there are
    records_wrong = 0
    records_total = 0

    # open the file
    with open (class_file, "r") as myfile:

        # read every line in the file
        for line in myfile.readlines():

            # separate each line by the tabs
            line = line.split("\t")

            # get the truth and predicted values
            _truth = int(line[1])
            _predicted = int(line[2])

            # add to the wrong records if
            # the prediction is wrong
            if _predicted != _truth:
                records_wrong = records_wrong + 1

            # add to the total records
            records_total = records_total + 1

    # calculate the error rate as wrong over total
    error = float(records_wrong) / float(records_total)

    # return this error rate
    return error
```

Mapper function

We re-use the mapper function from 1.4. We make a modification to disregard any inputs after the number of chunks because we are counting all words. Similarily, we don't print out any findwords to the output files.

```
%%writefile mapper.py
#!/usr/bin/python
## mapper.py
## Author: Alex Smith
## Description: mapper code for HW1.5

# import our libraries to use regular expression and
# read from the system arguments
import sys
import re

# initalize the outputs that we'll be using
email_id = None
word_act = None
spam_ind = None
word_ind = None

# collect user input, we are no longer
# collecting any information
# other than the number of chunks
filename = sys.argv[1]

# create a regex that matches alphanumeric characters
wordify = re.compile(r"[\w']+")

# open the file as readable
with open (filename, "r") as myfile:

    # loop through each line in the file
    for line in myfile.readlines()[1:]:

        # separate out each line based on the tabs
        line = line.split("\t")

        # set the email id and spam indicator fields
        email_id = line[0]
        spam_ind = line[1]

        # grab the text from the body and
        # subject and concatenate it
        text = line[2] + " " + line[3]

        # convert the text into a list of
        # words without punctuation
        # also lowercase all the words
        words = wordify.findall(text.lower())

        # loop through each word
        for word in words:

            # collect the line that we want to print out
            info = email_id + "\t" + word + "\t" + str(spam_ind)

            # let's print each email id, word,
```

```
            # spam indicator, and word of interest indicator
            # we'll separate each value with a tab character
            print info
```

```
Overwriting mapper.py
```

Reducer Function (with smoothing)

We modify our reducer function from 1.4 to calculate the probabilities for every word. We add a LaPlace smoother and log the probabilities.

```
%%writefile reducer.py
#!/usr/bin/python
## reducer.py
## Author: Alex Smith
## Description: reducer code for HW1.5

# import the system libraries to read
# from the input, from the
# math library import the log function
import sys
from math import log

# great a dictionary to store each email id,
# spam indciator, list of words of
# interest, and spam prediction
emails = {}

# grab and store the count files
count_files = sys.argv[1:]

# let's grab the find words from the first file
with open (count_files[0], "r") as myfile:
    findwords = myfile.readlines()[0].split("\t")
# we also need to remove the new line character
# that populates
if '\n' in findwords: findwords.remove('\n')

# initalize the summary statistic counts
all_words = 0
all_emails = 0
spam_emails = 0
spam_ewords = 0 # number of words in spam emails

# let's initalize a dictionary that will
# hold the probabilities for
# each and every word in the corpus
words_probs = {}

# loop through each file in the list of count files
for filename in count_files:

    # open the file as readable
    with open (filename, "r") as myfile:

        # loop through each line in the file
        for line in myfile.readlines():

            # split the line by tabs
            line = line.split("\t")

            # pull out my values for each part of the line
            email_id = line[0]
            word_act = line[1]
            spam_ind = int(line[2])
```

```
            # let's update the number of words and
            # the spam words (if it's spam)
            all_words = all_words + 1
            if spam_ind == 1:
                spam_ewords = spam_ewords + 1

            # if we don't already have the word in
            # our dictionary of words, let's
            # add it and initalize counts of zero
            if word_act not in words_probs:
                words_probs[word_act] = \
                {"spam_count":0,"not_spam_count":0}

            # let's grab the dictionary for the word
            word_dict = words_probs[word_act]

            # let's increment the word counts
            # for the word
            if spam_ind == 1:
                word_dict['spam_count'] = \
                word_dict['spam_count'] + 1
            else:
                word_dict['not_spam_count'] = \
                word_dict['not_spam_count'] + 1

            # check to see if this email is already
            # in the dictionary, and if
            # its not already there, initalize it
            if email_id not in emails:

                # create a sub-dictionary within
                # the email dictionary for each email
                emails[email_id] = \
                {"spam":spam_ind, "words":[]}

                # if it's not already there,
                # let's also increment the email counter
                # and the spam counter if it's spam
                all_emails = all_emails + 1
                if spam_ind == 1:
                    spam_emails = spam_emails + 1

            # let's add the word to our list
            # of words for this email
            emails[email_id]["words"].append(word_act)

# now that we have the summary statistics,
# we can calculate the probability that an email
# is spam given the words it contains;
# we use the bayesian probability formula from above

# posterior probabilities
prob_spam = float(spam_emails) / float(all_emails)
```

```
prob_nspam = 1.0 - prob_spam

# let's also take the log of these probabilities
log_prob_spam = log(prob_spam)
log_prob_nspam = log(prob_nspam)

# for each word, let's calculate the
# conditional probability of spam given the word
# and not spam given the word

# let's define our LaPlace smoother
SMOOTHER = 1.0
VOCAB = len(words_probs)

# look at each word
for word in words_probs.keys():

    # set the find word that we'll be
    # calculating the probabilities for
    word = words_probs[word]

    # calculate the probability of the word
    word['total'] = word['spam_count'] + \
    word['not_spam_count']
    word['probs'] = float(word['total']) / \
    float(all_words)

    # calculate the probability of word
    # given spam and add the smoother
    word['wordGIVspam'] = \
    float(word['spam_count'] + SMOOTHER) / \
    float(spam_ewords + VOCAB)

    # calculate the probability of the word
    # given not spam and add the smoother
    word['wordGIVnspam'] = \
    float(word['not_spam_count'] + SMOOTHER) / \
    float(all_words-spam_ewords + VOCAB)

    # calculate the probability of spam given
    # the word (here, we use the probability formula
    # provided in the explanation of the Naive
    # Bayes classifier from Section 1.2)
    word['spamGIVword'] = \
    (float(word['wordGIVspam']) * float(prob_spam)) / \
    float(word['probs'])

    # calculate the probability of not
    # spam given the word
    word['nspamGIVword'] = \
    (float(word['wordGIVnspam']) * float(prob_nspam)) / \
    float(word['probs'])

    # let's log the probabilities of interest
```

```
    word['log_spamGIVword'] = log(word['spamGIVword'])
    word['log_nspamGIVword'] = log(word['nspamGIVword'])

# now let's print our model out to a file
# this will be useful because it will make it
# easier to load it in future functions
with open("NaiveBayesSmoothing.txt","w") as myfile:

    # set the header
    myfile.write("Word \tCount \tP(Spam|word) \tP(Not Spam|word)
\n")

    # loop through each word
    for word in words_probs.keys():

        # set the word name
        word_name = word

        # set the word that we'll be
        # printing the probabilities for
        word = words_probs[word]

        # set each line as tab delimited
        info = str(word_name) + "\t" +\
        str(word['total']) + "\t" +\
        str(word['spamGIVword']) + "\t" +\
        str(word['nspamGIVword']) + "\n"

        # print each line
        myfile.write(info)

# now let's loop through each email in
# the dictionary and classify it as spam or not
# spam
for email in emails.keys():

    # get the actual classification
    truth = emails[email]["spam"]

    # if the email has no words, then set the
    # prediction based on the posterior
    # probabilities
    if len(emails[email]["words"]) == 0:
        if(log_prob_spam > log_prob_nspam):
            _prediction = 1
        else:
            _prediction = 0

    # else if the email has a word, then set
    # the prediction based on the
    # conditional probability
    else:

        # initalize a set of spam and not spam
```

```
        # probabilities
        spam_prob = 0
        nspam_prob = 0

        # loop through each of the words in
        # the list of words and
        # update the spam and not spam probabilities
        for word in emails[email]['words']:

            spam_prob = spam_prob + \
            words_probs[word]['log_spamGIVword']
            nspam_prob = nspam_prob + \
            words_probs[word]['log_nspamGIVword']

        if(spam_prob > nspam_prob):
            _prediction = 1
        else:
            _prediction = 0

    # print the output as the email id, the
    # actual classification, the prediction
    # as a tab-delimited line
    info = email + "\t" + str(truth) + "\t" + \
    str(_prediction)
    print info
```

```
Overwriting reducer.py
```

In [23]:
```
# modifies the permission to make the
# shell command executable from the notebook
!chmod a+x pNaiveBayes.sh
```

In [24]:
```
# set the right permissions for the
# python files mapper and reducer
!chmod +x mapper.py; chmod +x reducer.py
```

Run our model and print the output

In [25]:
```
# run the Naive Bayes counter
# with the word assistance
!./pNaiveBayes.sh 3 *

print "Done. Check the file enronemail_1h.txt.output"
```

```
Done. Check the file enronemail_1h.txt.output
```

Calculate the training error

In [26]:
```
# use the function defined at the start of this section to
# calculate and store the training error for the model
# with LaPlace smoothing
smoothing_error = trainingerror("enronemail_1h.txt.output")
```

Reducer Function (without smoothing)

We modify our reducer function from 1.4 to calculate the probabilities for every word. We remove the LaPlace smoother and return to regular (non-logged probabilities). We can easily remove the smoother because we have it as a variable, and we simply set it to zero.

```
%%writefile reducer.py
#!/usr/bin/python
## reducer.py
## Author: Alex Smith
## Description: reducer code for HW1.5

# import the system libraries to read
# from the input, from the
# math library import the log function
import sys
from math import log

# great a dictionary to store each email id,
# spam indciator, list of words of
# interest, and spam prediction
emails = {}

# grab and store the count files
count_files = sys.argv[1:]

# let's grab the find words from the first file
with open (count_files[0], "r") as myfile:
    findwords = myfile.readlines()[0].split("\t")
# we also need to remove the new line character
# that populates
if '\n' in findwords: findwords.remove('\n')

# initalize the summary statistic counts
all_words = 0
all_emails = 0
spam_emails = 0
spam_ewords = 0 # number of words in spam emails

# let's initalize a dictionary that will
# hold the probabilities for
# each and every word in the corpus
words_probs = {}

# loop through each file in the list of count files
for filename in count_files:

    # open the file as readable
    with open (filename, "r") as myfile:

        # loop through each line in the file
        for line in myfile.readlines():

            # split the line by tabs
            line = line.split("\t")

            # pull out my values for each part of the line
            email_id = line[0]
            word_act = line[1]
            spam_ind = int(line[2])
```

```
            # let's update the number of words and
            # the spam words (if it's spam)
            all_words = all_words + 1
            if spam_ind == 1:
                spam_ewords = spam_ewords + 1

            # if we don't already have the word in
            # our dictionary of words, let's
            # add it and initalize counts of zero
            if word_act not in words_probs:
                words_probs[word_act] = \
                {"spam_count":0,"not_spam_count":0}

            # let's grab the dictionary for the word
            word_dict = words_probs[word_act]

            # let's increment the word counts
            # for the word
            if spam_ind == 1:
                word_dict['spam_count'] = \
                word_dict['spam_count'] + 1
            else:
                word_dict['not_spam_count'] = \
                word_dict['not_spam_count'] + 1

            # check to see if this email is already
            # in the dictionary, and if
            # its not already there, initalize it
            if email_id not in emails:

                # create a sub-dictionary within
                # the email dictionary for each email
                emails[email_id] = \
                {"spam":spam_ind, "words":[]}

                # if it's not already there,
                # let's also increment the email counter
                # and the spam counter if it's spam
                all_emails = all_emails + 1
                if spam_ind == 1:
                    spam_emails = spam_emails + 1

            # let's add the word to our list
            # of words for this email
            emails[email_id]["words"].append(word_act)

# now that we have the summary statistics,
# we can calculate the probability that an email
# is spam given the words it contains;
# we use the bayesian probability formula from above

# posterior probabilities
prob_spam = float(spam_emails) / float(all_emails)
```

```
prob_nspam = 1.0 - prob_spam

# for each word, let's calculate the
# conditional probability of spam given the word
# and not spam given the word

# let's define our LaPlace smoother
# (in this case, 0)
SMOOTHER = 0
VOCAB = 0

# look at each word
for word in words_probs.keys():

    # set the find word that we'll be
    # calculating the probabilities for
    word = words_probs[word]

    # calculate the probability of the word
    word['total'] = word['spam_count'] + \
    word['not_spam_count']
    word['probs'] = float(word['total']) / \
    float(all_words)

    # calculate the probability of word
    # given spam and add the smoother
    word['wordGIVspam'] = \
    float(word['spam_count'] + SMOOTHER) / \
    float(spam_ewords + VOCAB)

    # calculate the probability of the word
    # given not spam and add the smoother
    word['wordGIVnspam'] = \
    float(word['not_spam_count'] + SMOOTHER) / \
    float(all_words-spam_ewords + VOCAB)

    # calculate the probability of spam given
    # the word (here, we use the probability formula
    # provided in the explanation of the Naive
    # Bayes classifier from Section 1.2)
    word['spamGIVword'] = \
    (float(word['wordGIVspam']) * float(prob_spam)) / \
    float(word['probs'])

    # calculate the probability of not
    # spam given the word
    word['nspamGIVword'] = \
    (float(word['wordGIVnspam']) * float(prob_nspam)) / \
    float(word['probs'])

# now let's print our model out to a file
# this will be useful because it will make it
# easier to load it in future functions
with open("NaiveBayesNoSmoothing.txt","w") as myfile:
```

```
    # set the header
    myfile.write("Word \tCount \tP(Spam|word) \tP(Not Spam|word)
\n")

    # loop through each word
    for word in words_probs.keys():

        # set the word name
        word_name = word

        # set the word that we'll be
        # printing the probabilities for
        word = words_probs[word]

        # set each line as tab delimited
        info = str(word_name) + "\t" +\
        str(word['total']) + "\t" +\
        str(word['spamGIVword']) + "\t" +\
        str(word['nspamGIVword']) + "\n"

        # print each line
        myfile.write(info)

# now let's loop through each email in
# the dictionary and classify it as spam or not
# spam
for email in emails.keys():

    # get the actual classification
    truth = emails[email]["spam"]

    # if the email has no words, then set the
    # prediction based on the posterior
    # probabilities
    if len(emails[email]["words"]) == 0:
        if(prob_spam > prob_nspam):
            _prediction = 1
        else:
            _prediction = 0

    # else if the email has a word, then set
    # the prediction based on the
    # conditional probability
    else:

        # initalize a set of spam and not spam
        # probabilities
        spam_prob = 1
        nspam_prob = 1

        # loop through each of the words in
        # the list of words and
        # update the spam and not spam probabilities
```

```
        for word in emails[email]['words']:

            spam_prob = spam_prob * \
            words_probs[word]['spamGIVword']
            nspam_prob = nspam_prob * \
            words_probs[word]['nspamGIVword']

        if(spam_prob > nspam_prob):
            _prediction = 1
        else:
            _prediction = 0

    # print the output as the email id, the
    # actual classification, the prediction
    # as a tab-delimited line
    info = email + "\t" + str(truth) + "\t" + \
    str(_prediction)
    print info
```

```
Overwriting reducer.py
```

In [28]:
```
# modifies the permission to make the
# shell command executable from the notebook
!chmod a+x pNaiveBayes.sh
```

In [29]:
```
# set the right permissions for the
# python files mapper and reducer
!chmod +x mapper.py; chmod +x reducer.py
```

Run our model and print the output

In [30]:
```
# run the Naive Bayes counter
# with the word assistance
!./pNaiveBayes.sh 3 *

print "Done. Check the file enronemail_1h.txt.output"
```

```
Done. Check the file enronemail_1h.txt.output
```

Calculate the training error

In [32]:
```
# use the function defined at the start of this section to
# calculate and store the training error for the model
# with LaPlace smoothing
no_smoothing_error = trainingerror("enronemail_1h.txt.output")
```

Compare the training errors

We compare the training errors between the model with LaPlace smoothing and the model without. We can see from the table below that the not smoothed model has a higher error rate than the smoothed model. This is because the not smoothed model fails to account for the possibility that a word could be in a class that we did not see. This will cause us to zero out a class unnecessarily.

In [33]:
```
# import the python library pretty table;
# this wil help us print out the comparison
# very clearly
from prettytable import PrettyTable

# create the pretty table and add a row for both models
prettySmooth = PrettyTable(["Model","Training Error Rate"])
prettySmooth.add_row(["Smoothed model",
                      round(smoothing_error,2)])
prettySmooth.add_row(["Not smoothed model",
                      round(no_smoothing_error,2)])

# print out the table
print prettySmooth
```

```
+--------------------+----------------------+
|       Model        | Training Error Rate  |
+--------------------+----------------------+
|   Smoothed model   |         0.0          |
| Not smoothed model |         0.03         |
+--------------------+----------------------+
```

HW1.6

Benchmark your code with the Python SciKit-Learn implementation of multinomial Naive Bayes. It always a good idea to test your solutions against publicly available libraries such as SciKit-Learn, The Machine Learning toolkit available in Python. In this exercise, we benchmark ourselves against the SciKit-Learn implementation of multinomial Naive Bayes. For more information on this implementation see: http://scikit-learn.org/stable/modules/naive_bayes.html (http://scikit-learn.org/stable/modules/naive_bayes.html).
Lets define Training error = misclassification rate with respect to a training set. It is more formally defined here:

- *Let DF represent the training set in the following:*
- *Err(Model, DF) = |{(X, c(X)) ∈ DF : c(X) != Model(x)}| / |DF|*
- *Where || denotes set cardinality; c(X) denotes the class of the tuple X in DF; and Model(X) denotes the class inferred by the Model "Model"*

In this exercise, please complete the following:

- *Run the Multinomial Naive Bayes algorithm (using default settings) from SciKit-Learn over the same training data used in HW1.5 and report the Training error (please note some data preparation might be needed to get the Multinomial Naive Bayes algorithm from SkiKit-Learn to run over this dataset)*
- *Run the Bernoulli Naive Bayes algorithm from SciKit-Learn (using default settings) over the same training data used in HW1.5 and report the Training error*
- *Run the Multinomial Naive Bayes algorithm you developed for HW1.5 over the same data used HW1.5 and report the Training error*
- *Please prepare a table to present your results*
- *Explain/justify any differences in terms of training error rates over the dataset in HW1.5 between your Multinomial Naive Bayes implementation (in Map Reduce) versus the Multinomial Naive Bayes implementation in SciKit-Learn (Hint: smoothing, which we will discuss in next lecture)*
- *Discuss the performance differences in terms of training error rates over the dataset in HW1.5 between the Multinomial Naive Bayes implementation in SciKit-Learn with the Bernoulli Naive Bayes implementation in SciKit-Learn*

Data preparation for Sklearn

Before putting the data through the sklearn algorithms, we have to prepare it. We do this by using the Sklearn's countvectorizers to transform the data into a bag of words.

In [34]:
```
# import the count vectorizer function from the
# appropriate Sklearn library and import numpy to reshape
# the labels array into a 100 x 1 array
from sklearn.feature_extraction.text import CountVectorizer
import numpy as np

# create arrays to hold the data
email_ids = []
train_labels = []
email_text = []

# begin by opening the file
with open("enronemail_1h.txt", "r") as myfile:

    # loop through each line
    for record in myfile.readlines():

        # split each line based on the tabs
        record = record.split("\t")

        # generate the text as a combination
        # of the subject and body
        text = record[2] + " " + record[3]

        # add to the email_ids, to the labels,
        # and to the email text
        email_ids.append(record[0])
        train_labels.append(record[1])
        email_text.append(text)

# genearate a bag of words on the email texts
# by using count vectorizer
bag = CountVectorizer()
train_data = bag.fit_transform(email_text)

# convert email labels into a numpy array of size 100x1
train_labels = np.array(train_labels).reshape(-1)

# to verify that we've got the right thing,
# let's print out the shape
# of our training data and training labels.
# it should be 100 rows
# have a column for every word
print "Our bag of words has a shape of:",train_data.shape
print "Our traiing labels have a shape of:", train_labels.shape
```

```
Our bag of words has a shape of: (100, 5375)
Our traiing labels have a shape of: (100,)
```

Outputing SKLearn Findings

We create a function that will output the results from running SKLearn's algorithms.

In [35]:
```
def outputSK(filename,record_ids,truths,predictions):
    """Takes three arrays (record id, truth,
    prediction) and a file name and arranges
    them as a tab delimited file with columns:
    record id, truth, and prediction and
    outputs the file with specified file name"""

    # open the file as write-able
    with open(filename, "w") as myfile:

        # loop through every record ids, truth,
        # and prediction
        for index,record in enumerate(record_ids):

            # create the line that has the information
            # we'll write to the file
            new_line = record_ids[index] + "\t" + \
            truths[index] + "\t" + \
            predictions[index] + "\n"

            # write the line to the file
            myfile.write(new_line)
```

SKLearn's Multinomial Naive Bayes Algorithm

We use SKLearn's multinomial naive bayes algorithm on our email data to generate predictions of spam or not spam. We then use our output function to save the file.

In [36]:
```
# import the multinomial naive bayes algorithm
from sklearn.naive_bayes import MultinomialNB

# create and train the model on our data
sk_multinomial = MultinomialNB()
sk_multinomial.fit(train_data,train_labels)

# use our model to predict the training data
sk_predictions = sk_multinomial.predict(train_data)

# output the results to a file that we
# can use for comparison
outputSK("sk_multinomial.txt",email_ids,\
         train_labels,sk_predictions)

print "Check out the output file \
'sk_multinomial.txt' \nto see the results from \
the SK Learn\n Multinomial Naive Bayes algorithm."
```

```
Check out the output file 'sk_multinomial.txt'
to see the results from the SK Learn
 Multinomial Naive Bayes algorithm.
```

SKLearn's Bernoulli Naive Bayes Algorithm

We use SKLearn's bernouilli naive bayes algorithm on our email data to generate predictions of spam or not spam. We then use our output function to save the file.

In [37]:
```
# import the multinomial naive bayes algorithm
from sklearn.naive_bayes import BernoulliNB

# create and train the model on our data
sk_bernoulli = BernoulliNB()
sk_bernoulli.fit(train_data,train_labels)

# use our model to predict the training data
sk_predictions = sk_bernoulli.predict(train_data)

# output the results to a file that we can
# use for comparison
outputSK("sk_bernoulli.txt",email_ids,\
         train_labels,sk_predictions)

print "Check out the output file \
'sk_bernoulli.txt' \nto see the results from \
the SK Learn\n Multinomial Naive Bayes algorithm."
```

```
Check out the output file 'sk_bernoulli.txt'
to see the results from the SK Learn
 Multinomial Naive Bayes algorithm.
```

Reporting the training error

Now let's report the training error across the 3 models (1.5, SKLearn's Multinomial Naive Bayes, and SKLearn's Bernoulli Naive Bayes. Remember that we can use the training error function we created for 1.5.

In [38]:
```
# import the python library pretty table;
# this wil help us print out the comparison
# very clearly
from prettytable import PrettyTable

# get the training errors for each algorithm
my_error = smoothing_error # already calculated in 1.5
sk_multi_error = trainingerror("sk_multinomial.txt")
sk_berno_error = trainingerror("sk_bernoulli.txt")


# create the pretty table and add a row for each model
pretty = PrettyTable(["Model","Training Error Rate"])
pretty.add_row(["My Naive Bayes Model", my_error])
pretty.add_row(["SKLearn's Multinomial Model", sk_multi_error])
pretty.add_row(["SKLearn's Bernoulli Model", sk_berno_error])

# print out the table
print pretty
```

```
+-----------------------------+----------------------+
|            Model            | Training Error Rate  |
+-----------------------------+----------------------+
|     My Naive Bayes Model    |         0.0          |
| SKLearn's Multinomial Model |         0.0          |
|  SKLearn's Bernoulli Model  |         0.16         |
+-----------------------------+----------------------+
```

We see that our Naive Bayes Model and SKLearn's Multinomial Naive Bayes model have the same error rates. This make sense because they use the same probability calculations. They even both use the same smoother of 1. They are both designed to handle this type of data. They can handle multiple values for a given feature. However, the Bernoulli model relies on the assumption that each feature is a binary value. We know this to be false for our data here because a given word can appear at different frequencies. The Bernoulli model attemps to binarize the data and results in losing information. As a side note, it's interesting that our error rate is zero for my and SKLearn's multinomial naive bayes classifier. This is because we trained the model on the training data and then tested it on the same training data. We should have tested it on separate test data. We got such a low error rate, in part, because the overlapp between words in the ham and not ham classes was pretty small, combined with a small training set and a large feature set (every vocab word) meant that we likely overfit our training data.