Homework 4

Statistics W4240: Data Mining

Columbia University

Due Tuesday, November 11 (Sections 01, 02, 03, 04)

For your .R submission, submit a file for questions 1,2,3,4 labeled `hw04_q1.R`, `hw04_q2.R`, and so on. The write up should be saved as a .pdf of size less than 4MB. **DO NOT** submit .rar, .tar, .zip, .docx, or other file types.

Problem 1. Naive Bayes Text Classification (25 Points) Working with data can often be messy and frustrating, especially when you are learning a language. The following is designed to be an introduction to working with real world data. You will implement an authorship attribution algorithm for the `Federalist` dataset using a Naive Bayes classifier. This dataset contains historically important papers written by two founding American politicians (Hamilton and Madison), and your task will be to classify the author from the text. The authorship of these papers has been a subject of interesting historical debate, so this use of Naive Bayes is meaningful. Please note that the vast majority of the code has been supplied for you (inline and in `hw04.R`); your code should only involve a few commands per part, where a number of those commands will call the functions we supply.

a. **Step 1 (5 points)**

Download the `Federalist Paper` documents from the course website. Place them in your working directory for `R`. First we must preprocess the data to 1) remove non-letter characters, 2)remove stopwords, and 3)stem words. Stopwords are short functional words, such as *in, is, the*. Stemming involves trimming inflected words to their stem, such as reducing *running* to *run*.

```
########################################
# This code uses tm to preprocess the papers into a format useful for NB
preprocess.directory = function(dirname){

# the directory must have all the relevant text files
ds = DirSource(dirname)
# Corpus will make a tm document corpus from this directory
fp = Corpus( ds )
# inspect to verify
# inspect(fp[1])
# another useful command
# identical(fp[[1]], fp[["Federalist01.txt"]])
# now let us iterate through and clean this up using tm functionality
for (i in 1:length(fp)){
# make all words lower case
fp[i] = tm_map( fp[i] , tolower);
```

```
# remove all punctuation
fp[i] = tm_map( fp[i] , removePunctuation);
# remove stopwords like the, a, and so on.
fp[i] = tm_map( fp[i], removeWords, stopwords("english"));
# remove stems like suffixes
fp[i] = tm_map( fp[i], stemDocument)
# remove extra whitespace
fp[i] = tm_map( fp[i], stripWhitespace)
}
# now write the corpus out to the files for our future use.
# MAKE SURE THE _CLEAN DIRECTORY EXISTS
writeCorpus( fp , sprintf('%s_clean',dirname) )
}
##########################################
```

The above code creates a reusable *function* in `R`, which takes the input argument `dirname`. Use functions when you need to repeatedly use a bit of code. In this case, we will be reading in data from four different directories. Your task is to run this code on each of the four directories: `fp_hamilton_train, fp_hamilton_test, fp_madison_train, fp_madison_test`. Look at the files before (namely, in the original directory) and after (namely, in the '_clean' directory) you have used the above function.

Your submitted code for problem 1 should, for part a, include this function with `source('hw04.R')`, after which you should call the above function on each of the four directories. The code should be very easy once you understand the above steps.

Please note that the above function uses the package `tm`, which has much of the functionality below. However, for instructive purposes, we will implement the remaining functions manually. Be sure you install `tm` correctly.

b. **Step 2 (5 points)**

We are next going to use a function to load each of the Federalist Papers from their corresponding directory into your workspace:

```
##########################################
# To read in data from the directories:
# Partially based on code from C. Shalizi
read.directory <- function(dirname) {
    # Store the infiles in a list
    infiles = list();
    # Get a list of filenames in the directory
    filenames = dir(dirname,full.names=TRUE);
    for (i in 1:length(filenames)){
        infiles[[i]] = scan(filenames[i],what="",quiet=TRUE);
         }
    return(infiles)
```

```
}
########################################
```

This code creates a *list* of input files, which here are each federalist paper, and which we will refer to as *infiles*. Lists store objects with an ordering. In this case, the order is the dictated by the position in the directory. The code is structured as follows:

- `infiles = list()` instantiates the variable `infile` as a list, albeit with no entries currently.
- `filenames = dir(dirname,full.names=TRUE)` creates a vector of strings, populated by the names of the files in directory `dirname`. The option `full.names=TRUE` means that the path directory is prepended onto the file name to give a path; if this is set to `FALSE`, only the file names are given.
- `for (i in 1:length(filenames)){` loops through all of the file names.
- `infiles[[i]] = scan(filenames[i],what="",quiet=TRUE)` fills list element `i` with a vector (or list) of input values. Here `what=""` means that all fields will be read as strings. The default separator is white space; this can be changed through the `sep` argument. The argument `quiet = TRUE` keeps `scan` from displaying statistics about the imported document, which will fill up a command screen when hundreds of documents are read. The net result of this command is that element `i` of `infile` is filled by a vector of strings, with each string representing a word.
- `return(infiles)` returns the object `infiles` when the function `read.directory(dirname)` is called.

Your code should now read in all four infile directories (the **cleaned** directories that you created in the previous part) into the following variable names:

```
hamilton.train
hamilton.test
madison.train
madison.test
```

c. **Step 3 (5 points)**

We are next going to use all of the files (training and testing for both authors) to make a dictionary, or a list of all the words used. We have again supplied the key function:

```
########################################
# Make dictionary sorted by number of times a word appears in corpus
# (useful for using commonly appearing words as factors)
# NOTE: Use the *entire* corpus: training, testing, spam and ham
make.sorted.dictionary.df <- function(infiles){
    # This returns a dataframe that is sorted by the number of times
    # a word appears
```

```
    # List of vectors to one big vetor
    dictionary.full <- unlist(infiles)
    # Tabulates the full dictionary
    tabulate.dic <- tabulate(factor(dictionary.full))
    # Find unique values
    dictionary <- unique(dictionary.full)
    # Sort them alphabetically
    dictionary <- sort(dictionary)
    dictionary.df <- data.frame(word = dictionary, count = tabulate.dic)
    sort.dictionary.df <- dictionary.df[order(dictionary.df$count,decreasing=TRUE),];
    return(sort.dictionary.df)
}
########################################
```

The function `make.sorted.dictionary.df(infiles)` takes the list `infiles` generated by `read.directory()` and creates a data frame that holds 1) the list of unique words, and 2) the number of times each appears in the corpus, which are sorted in descending order based on frequency. We want the sorted list so that we can easily access the most used words. The code is structured as follows:

- `dictionary.full <- unlist(infiles)` takes the list `infiles` and converts it into a very long vector, `dictionary.full`.
- `tabulate.dic <- tabulate(factor(dictionary.full))` makes a vector of counts for each word that appears in `dictionary.full`. The argument `factor(dictionary.full)` tells the tabulate function that it is to bin data according to the enumerated categories for `dictionary.full`.
- `dictionary <- unique(dictionary.full)` returns a vector of unique words in `dictionary.full`.
- `dictionary <- sort(dictionary)` sorts the vector `dictionary` from a to z; this is necessary because `factor(dictionary.full)` is a sorted list of words in `dictionary.full`.
- `dictionary.df <- data.frame(word = dictionary, count = tabulate.dic)` creates a new data frame, `dictionary.df`, with two categories, `word` and `count`. The `word` category is populated by the alphabetized dictionary of factors, the `count` category is populated by the counts for the alphabetized words.
- `sort.dictionary.df <- dictionary.df[order(dictionary.df$count,decreasing=TRUE),]` reorders the elements of `dictionary.df` to be sorted in descending order according to the count values. `order(dictionary.df$count,decreasing=TRUE)` returns an index ordering when the count values are sorted in a descending manner; this is used to rearrange the rows, but not the columns, of the data frame.
- `return(sort.dictionary.df)` returns the sorted values as a data frame.

Your task is to now create a dictionary for the full dataset (all test and training across both authors) by concatenating the individual lists into a single large one for all infiles, and then using the large list to create a dictionary.

d. **Step 4 (5 points)**

We are now going to create counts for each dictionary word in all documents and place them in a document (rows) by word (columns) matrix. The following code will be used:

```
########################################
# Make a document-term matrix, which counts the number of times each
# dictionary element is used in a document
make.document.term.matrix <- function(infiles,dictionary){
    # This takes the text and dictionary objects from above and outputs a
    # document term matrix
    num.infiles <- length(infiles);
    num.words <- length(dictionary);
    # Instantiate a matrix where rows are documents and columns are words
    dtm <- mat.or.vec(num.infiles,num.words); # A matrix filled with zeros
    for (i in 1:num.infiles){
        num.words.infile <- length(infiles[[i]]);
        infile.temp <- infiles[[i]];
        for (j in 1:num.words.infile){
            ind <- which(dictionary == infile.temp[j]);
            dtm[i,ind] <- dtm[i,ind] + 1;
        }
    }
return(dtm);
}
########################################
```

In many cases, producing a full document-term matrix is a poor idea. It requires a large amount of storage space for relatively little information. In settings with large corpora or large dictionaries, it is often best to store the data in the form of *tokens* that simply denote document number, word number in the dictionary, and count for that word-document combination. Only counts of at least 1 are stored, yielding something like this:

document	word	count
3	1	2
3	5	1
3	11	2
3	12	2
...	...	...

However, matrix representation greatly simplifies the coding required for implementing a naive Bayes classifier. The function is structured as follows:

- `num.infiles <- length(infiles)` calculates the number of infiles by computing the length of the list.

- `num.words <- length(dictionary)` calculates the length of the dictionary.
- `dtm <- mat.or.vec(num.infiles,num.words)` instantiates a matrix with `num.infiles` rows and `num.words` columns that is filled with zeros. Note that `mat.or.vec(num.infiles)` would instantiate a vector with `num.infiles` entries that is filled with zeros.
- `for (i in 1:num.infiles){` loops through each of the infiles.
- `num.words.infile <- length(infiles[[i]])` calculates the number of words in infile `i`.
- `infile.temp <- infiles[[i]]` temporarily stores the vector infile `i` in `infile.temp`.
- `for (j in 1:num.words.infile){` loops through each of the words in infile `i`.
- `ind <- which(dictionary == infile.temp[j])` stores the dictionary index equal to the `j`th word of infile `i`.
- `dtm[i,ind] <- dtm[i,ind] + 1` adds a count of 1 to to the observation matrix for word `ind` in infile `i`.
- `return(dtm)` returns the document-term matrix.

Your taks is to create a document term matrix for each group of papers. Since we are looping over a large space in `R`, this may take a few minutes to run. Make the following:

```
dtm.hamilton.train
dtm.hamilton.test
dtm.madison.train
dtm.madison.test
```

e. **Step 5 (5 points)**

This exercise culminates in building a Naive Bayes classifier. All of your data is now in matrix form. We are now going to use it to compute naive Bayes probabilities. Computationally, we may ignore the normalizer $p(\mathbf{x}^{test})$ can be ignored since we only need to determine which class has a higher likelihood. A test document is declared to be authored by Hamilton ($y = 1$) if

$$p(y=1)\prod_{j=1}^{n^{test}} p(X_j = x_j^{test} \,|\, y=1) \geq p(y=0)\prod_{j=1}^{n^{test}} p(X_j = x_j^{test} \,|\, y=0),$$

and Madison ($y = 0$) otherwise. Therefore, we only need to estimate the probabilities $p(y = 1)$, $p(y = 0)$, $p(x_j = k \,|\, y = 1)$ for $k = 1, \ldots, |D|$, and $p(x_j = k \,|\, y = 0)$ for $k = 1, \ldots, |D|$.

When computing probabilities defined by products, it is more convenient to work in *log probabilities*. Consider a dictionary with two words, each with equal probability given that the document is by Hamilton, and a test document with 100 words. Then,

$$p(y = 1 \,|\, \mathbf{x}^{test}) = \frac{1}{2} \times \prod_{j=1}^{100} \frac{1}{2} = \left(\frac{1}{2}\right)^{101} \approx 3.9 \times 10^{-31}.$$

While best case for a small document, this is way beyond machine precision. If we compute the log probability,

$$\log\left(p(y=1\,|\,\mathbf{x}^{test})\right) = \log\left(\left(\frac{1}{2}\right)^{101}\right) = -101 \times \log(2) \approx -70.01.$$

This can easily be stored within machine precision. Even when computing a vector of probabilities, it is often a good idea work in log probabilities. Note that programs like `R` or `Matlab` use base e.

To compute a vector of log probabilities from a document term matrix with μ phantom examples, use the following function:

```
########################################
make.log.pvec <- function(dtm,mu){
    # Sum up the number of instances per word
    pvec.no.mu <- colSums(dtm)
    # Sum up number of words
    n.words <- sum(pvec.no.mu)
    # Get dictionary size
    dic.len <- length(pvec.no.mu)
    # Incorporate mu and normalize
    log.pvec <- log(pvec.no.mu + mu) - log(mu*dic.len + n.words)
    return(log.pvec)
}
########################################
```

The following commands were introduced in this code:

- `sum(pvec.no.mu)` sums over all values in the vector `pvec.no.mu`.
- `colSums(dtm)` sums over the rows of the matrix `dtm`, producing a vector with the number of elements equal to the number of columns in `dtm`.
- `pvec.no.mu + mu` adds the scalar value `mu` to all elements of the vector `pvec.no.mu`.
- `(pvec.no.mu + mu)/(mu*dic.len + n.words)` divides all elements of the vector `pvec.no.mu + mu` by the scalar value `mu*dic.len + n.words`.

Your task is to calculate the log probability vectors for all document term matrices. Make your code compute the following with $\mu = \frac{100}{|D|}$, where $|D|$ is the size of the dictionary (number of words in the dictionary):

```
logp.hamilton.train
logp.hamilton.test
logp.madison.train
logp.madison.test
```

Question 2. (25 points) Write a function to compute $\log\left(\hat{p}(y = \tilde{y} \mid \mathbf{x}^{test})\right)$ and assign authorship to a test paper:
`naive.bayes = function(logp.hamilton.train, logp.madison.train, log.prior.hamilton, log.prior.madison , dtm.test).`
This function takes the log probabilities for the dictionary in a Hamilton-authored document (`logp.hamilton.train`), a Madison-authored document (`logp.madison.train`), the log priors (`log.prior.hamilton, log.prior.madison`) and a document term matrix for the testing corpus. Here you must write the Naive Bayes classifier code as a function.

Question 3. (20 points) Set $\mu = \frac{100}{|D|}$ as before, use the training sets for training, and use the testing sets for testing. What percentage of test papers are classified correctly? Additionally, report the proportion of your: true positives (Hamilton classified as Hamilton divided by the total amount of testing Hamilton papers), true negatives (Madison classified as Madison divided by the total amount of testing Madison papers), false positives (Madison classified as Hamilton divided by the total amount of testing Madison) and false negatives (Hamilton classified as Madison divided by the total amount of testing Hamilton).

Question 4. (30 points) Naive Bayes has a tunable parameter, μ. We will use 5-fold cross-validation over the training set to search the parameter set

$$\mu = \{\frac{1}{|D|}, 10\frac{1}{|D|}, 100\frac{1}{|D|}, 1000\frac{1}{|D|}, 10000\frac{1}{|D|}\}.$$

Store the proportion correctly classified, proportion of false negatives, and proportion of false positives. (You may consider using matrices, lists, or data frames to store these values.)

a. (10 points) For each value of μ, estimate the correct classification rate, the false negative rate and the false positive rate using 5-fold cross-validation on the training set. Summarize your results in three graphs. (Note: if you are plotting μ as a value rather than a factor, use log scale.)

b. (5 points) What seems to be the best value for μ? Why?

c. (10 points) For each value of μ, train on the full training set and test on the full testing set. Summarize the correct classification rate, the false negative rate and the false positive rate in three graphs. Does your answer from (b) still seem the best value? Why or why not?

d. (5 points) How close are the rates estimated from cross-validation to the true rates on the testing set (give percentage error)? What could account for differences? Give one way the differences between the cross-validation rate estimates and the rates on the training sets could be minimized.