Katie McCorkell
Kmccork 0822555
Machine Learning 446
Winter 2014
Homework 3

2.2 Naïve Bayes Classifier

To run my code: run the NiaveBayesClassifier.java file with a parameter given.

The code works as follows: first it learns on the training set. To do this, it calculates the percentage of spam emails out of the whole training set. This is **totalCount.pSpam**. Also, out of all the words in the emails, it makes a wordMap<String, WordCount>. WordCount calculates the probability of a given word in spam and non-spam email. This is used for probability (word | spam) and p(word | not spam). For example:

$$P(\text{word} \mid \text{spam}) = \frac{countOfThisWordInSpamEmails + (m * p)}{countsOfAllWordsInSpamEmails + m}$$

$$P = \text{prior probability of this word being spam} = \frac{1}{vocabsize}$$

where vocabsize is number of unique words, not total number of words

m = the given smoothing parameter

For the testing set: a new example email is spam if the probability that it is spam is greater than the probability that it is ham. The probability that it is spam is calculated by the prior probability of spam (totalCount.pSpam) multiplied by the product of (for each of the email's words) p(word|spam). If a word occurs more than once, each occurrence is factored in. In my code I added the logs of the probabilities instead of multiplying the probabilities, in order to avoid underflow. Since we are taking the max, this is acceptable.

The accuracy is the number of emails that the classifier correctly predicts if they are spam or not.

I obtained an accuracy of 90.4% .

2.3 I implemented the smoothing parameter, as described above it is the m value. The most effective was when m was equal to 10000 or 100000, however this was based on the test data, so this may well be a case of over fitting to the test data. It would be better if I split the training data up to also include a verification set, and then tested on it to find the optimal parameter.

These parameters were optimal because they have the effect of moderate smoothing. Set m as too small and there is basically no effect, since you're only adding a very small amount to the probability. Set m as too big and you are making too big assumptions about the prior probability, since m is multiplied by the prior probability, this would magnify whatever you pick for prior probability.

The main importance of smoothing is that it handles the situation where there is a probability of 0. Say for example you see a word in a test case that you never saw in a training case, so the probability of the word would be 0, without the smoothing. Then since the probabilities are multiplied together, the probability of that entire email being spam would be zero. Smoothing prevents that by putting in a place-holder value that is not zero. Since it is the same for any word that doesn't exist in training data, it doesn't really skew the data at all, it just prevents it from being zero.

```
Naive Bayes Classifier
M           accuracy
1.0E-4          0.904
0.0010          0.903
0.01            0.902
0.1             0.902
1.0             0.902
10.0            0.902
100.0           0.902
1000.0          0.902
10000.0         0.904
100000.0        0.904
1000000.0       0.871
1.0E7           0.765
1.0E8           0.699
1.0E9           0.623
1.0E10          0.592
1.0E11          0.581
1.0E12          0.58
1.0E13           0.58
```

2.4 Bayes Extra Credit

As I read in the Sahami paper, there are a number of domain specific features that can be used to strengthen the generic Naive Bayes Algorithm for working specifically on spam emails. For example the words "FREE!" and "only $" or the use of excessive exclamation marks or other punctuation was highly indicative of spam, and modifying the treatment of these features helped improve accuracy. To focus my work for the extra credit, I focused on the number of emails that were marked as Spam, but in actuality are ham. There were 79 of these (when running with M = 10000). My goal was to reduce this, because that is really annoying when you miss emails that you do need. I decided to look at the .edu feature: Sahami says emails sent from .edu addresses are almost never spam.

This feature is given based on prior knowledge, not from the training set. However I did look at the training set and I found that 12/50 emails from edu accounts were spam. This is not the "almost never" that Sahami describes. I changed the is **totalCount.pSpam** probability for all edu emails to be 12/50, (and the inverse for ham), but saw no effect. So I decided to apply Sahami's rule more strictly: if an email is sent from a .edu address, it is not spam. This corrected 2 errors so there are only 77 false negative errors compared to 79.

(negative = spam). But this then causes two different emails to be marked as ham when they are not, (originally 17 such errors, after modifications 19 such errors). So I decided to look closer at the probabilities.

Note that the code I discuss here is at the end of my testData method, and there are some printlns there which can display a little bit what is going on, for example display the false negative rate. The bash code for data file manipulation is at the bottom of this or the next page, and the java method that interacts with the bash output files is `getEduEmails`.

```
Naive Bayes Classifier
@edu emails
guessItsSpam: true targetIsSpam: false
pHam : -1638.7568066270421 pSpam: -1626.0032421914702
(pHam-pSpam) / pSpam 0.007843504923386918
guessItsSpam: false targetIsSpam: false
pHam : -1265.1401830312536 pSpam: -1386.8532782579894
(pHam-pSpam) / pSpam 0.08776205611282699
guessItsSpam: true targetIsSpam: false
pHam : -1256.197585706773 pSpam: -1217.7326214265022
(pHam-pSpam) / pSpam 0.031587364585184026
guessItsSpam: true targetIsSpam: true
pHam : -1475.1670002770552 pSpam: -1401.473322616192
(pHam-pSpam) / pSpam 0.05258300423678128
guessItsSpam: true targetIsSpam: true
pHam : -4542.591691706505 pSpam: -4262.65345625496
(pHam-pSpam) / pSpam 0.0656722950444794
Marked Spam but is Ham 79
Marked Ham but is Spam 17
M                    accuracy
10000.0              0.904
```

I was hoping there would be a clear distinction, like the spam .edu emails have a really high probability of spam compared to probability of ham. In fact I do not see an OBVIOUS distinction. By looking closely you could calculate the difference (pHam-pSpam)/pSpam and when the difference is less than .05 AND the email is sent from .edu, then you should vote it is a ham email. This will give you an accuracy of .906 However that number is based on my visual learning from the test data, so that's sketchy - as I am basically cheating. I would need a different training set to learn the proper cutoff.

If you don't make such a qualification, but do modify your vote based on .edu email address, then you get the same accuracy as before .904, but with 2 less false-negative errors. You also have two more false positive errors, but these errors are considered less damaging by users. (Where positive = ham). (Note the 4 errors are all on unique files).

Bash Script: This script outputs the email ids of test emails with .edu addresses. Execute this command in a folder where you have both test.index and the trec05p-1 data corpus . You can copy and paste the below command as one line. It will produce a file called eduEmailIds.txt which is the list of test emails ids that were sent from .edu email addresses. You can also substitute train.index in order to get email ids of edu training emails.

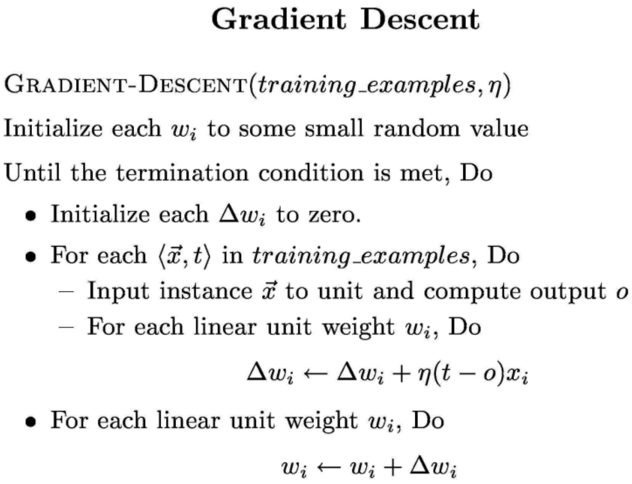
```
while read p; do  echo ${p:4}; done < test.index >TestIndex.index; while read fileName;  do echo -n ${fileName}; grep "From: " trec05p-1/data${fileName} |  tr -d '\n'; echo  ;  done <TestIndex.index  > emails.txt; grep "\.edu" emails.txt > eduEmails.txt; while read p; do echo ${p:0:8}; done <eduEmails.txt> testEmailEduIds
```

Katie McCorkell

Problem 2: Textbook problems re: Neural Networks

MITCHELL, 4.5

The gradient descent algorithm should be implemented as in the course lecture slides on Nueral Networks, slide 15 (below), however a different formula for Δw_i should be used.



Gradient Descent

```
GRADIENT-DESCENT($training\_examples, \eta$)
Initialize each $w_i$ to some small random value
Until the termination condition is met, Do
  • Initialize each $\Delta w_i$ to zero.
  • For each $\langle \vec{x}, t \rangle$ in $training\_examples$, Do
      – Input instance $\vec{x}$ to unit and compute output $o$
      – For each linear unit weight $w_i$, Do
            $\Delta w_i \leftarrow \Delta w_i + \eta(t-o)x_i$
  • For each linear unit weight $w_i$, Do
            $w_i \leftarrow w_i + \Delta w_i$
```

Δw_i should be equal to $-\eta\ (\frac{\partial E}{\partial w_i})$ as before, but a different value should be used for $\frac{\partial E}{\partial w_i}$.

$\frac{\partial E}{\partial w_i}$ should be derived as follows

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} * E$$

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} * \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

where o_d is the o given in problem 4.5, see next page

$$\frac{\partial E}{\partial w_i} = \frac{1}{2} \sum_{d \in D} 2(t_d - o_d)\ \frac{\partial}{\partial w_i}(t_d - o_d)$$

$$\frac{\partial E}{\partial w_i} = \sum_{d \in D} (t_d - o_d)\ \frac{\partial}{\partial w_i}(t_d - (\overset{r}{w} \bullet \overset{r}{x}_d + \overset{r}{w} \bullet \overset{r}{x}^2{}_d))$$

*sorry those funny r symbols should be vector symbols

$$\frac{\partial E}{\partial w_i} = \sum_{d \in D} (t_d - o_d)\ (-(x_{i,d} + x^2{}_{i,d}))$$

DERIVATION OF o_d

o_d refers to the equation given in the problem, with respect to each example d.

$$o = w_o + w_i x_i + w_i x_i^2 + w_n x_n + w_n x_n^2$$

This can be rewritten as $o_d = (\vec{w} \bullet \vec{x}_d) + (\vec{w} \bullet \vec{x}_d{}^2)$ ****see note below**

Thus, the derivative of this with respect to weight i, for any x$_d$ is $(x_{i,d} + x_{i,d}^2)$

Since only the i terms remain after taking the derivative with repsect to weight i.
This is used in my work on the previous page.

****note:** the equation o is able to be rewritten as this because of the definition of the dot product. You might be wondering where the $w_o x_o + w_o x_o^2$ terms are in the expanded version of o. Well you can show that there is an x_o such that $w_o x_o + w_o x_o^2$ is equal to w_o.

That value is in fact $x_0 = \frac{1}{2}(\sqrt{5} - 1)$

MITCHELL, 4.10

This can be implemented by multiplying each weight by the constant (1-2γŋ) upon each iteration before performing the standard gradient descent update.

$$E(\vec{w}) = (\frac{1}{2} \sum_{d \in D} \sum_{k \in K} (t_{k,d} - o_{k,d})^2) + \gamma \sum_{i,j} w^2{}_{j,i}$$

The derivative of this is equal to the derivative of the left half of the equation plus the derivative of the right half of the equation. The left half of this equation is equal to the previous gradient descent rule, and therefore its derivative is that which follows from pages 113-116 of the PDF file of the Mitchell textbook: $\partial_j x_{j,i}$

The derivative of the right side of the equation with respect to $w_{j,i}$ is $2\gamma w_{j,i}$, since only the i and j terms matter when you take the derivative.

Now Δwᵢ is equal to –ŋ ($\frac{\partial E}{\partial w_i}$) as before, but the $\frac{\partial E}{\partial w_i}$ is replaced by the above error euqation's derivative.

So Δwᵢ is $-\eta(-\partial_j x_{j,i} + 2\gamma w_{j,i})$.

= ($\eta\partial_j x_{j,i} - 2\eta\gamma w_{j,i}$).

So the overall update rule is: Wji = Wji + ($\eta\partial_j x_{j,i} - 2\eta\gamma w_{j,i}$)).

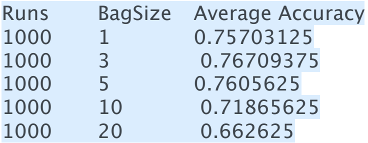
This is equal to $w_{i,j}(1-2n\gamma)+\eta\partial_j x_{j,i}$. This is the udpate to be performed.

The second part of that is the original error update, Δwᵢ, so it can be said that it is equal to multiplying each weight by the constant (1-2γŋ) upon each iteration before performing the standard gradient descent update.

PROBLEM 4: BAGGING

Run the BaggingEnsemble.java file with a integer argument (N).

My code works as follows:
N classfiers are trained on N data sets which are created by sampling with replacement from the training data file. The classifier in this case is the Weka Id3 tree. Next each test example is classified based by each of the N decision trees. For each test example, my learner votes based on the majority of the N decision trees' votes. The vote of my learner is compared to the true vote in order to measure accuracy.



Runs	BagSize	Average Accuracy
1000	1	0.75703125
1000	3	0.76709375
1000	5	0.7605625
1000	10	0.71865625
1000	20	0.662625

Note, I did not see the expected benefit from bagging. One would expect the accuracy to go up as the size of the bag increases, as bagging improves accuracy for unstable classifiers, since it effectively averages over discontinuities (Duda, Chapter 9). Because the decision trees could be very different depending on slight variations in the data set, this would lead to many different responses, and the bagged learner would average over these differences, rather than just taking one of them which might be the wrong one.

I am not sure why I did not see the expected benefit of bagging. Robert suggested in his office hours to many of us that it could be the effect of Random object being used too fast in a row, so it is not truly random (since it is based on clock time). However, I tested the random object and observed what appeared to me to be decently random numbers. Furthermore, I tried using Java's thread.sleep to sleep the code between calls to the random object, so that it would be different clock times. This generated more or less the same result as above. My hypothesis is that if I were to run it many more times, I eventually wouldn't see the 20 –size be the worst accuracy – I would see them all about equal, and that these results were just an anomoly. It's possible that because the data set is so small there is no noticeable gain from bagging, and there are noticeable fluctuations in accuracy.

Note, I used the Weka library for the Id3 tree. I used the Weka library Classifier interface, which provides the "black box generic interface" requested in the HW writeup. You could easily exchange the Id3 tree for a different Weka tree or classifier, by replacing just one word in the code. (instead of new Id3();, put new J48();, etc.).

The Classifier interface accepts an Instances object to train on. I used the Weka Instances class to store the arff data. I made new Instances objects for each new random sample, and implemented the "sampling" myself in a method called "bootstrap". More info about Weka can be found here: http://www.cs.waikato.ac.nz/ml/weka/.