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(word | spam) = 

P = prior probability of this word being spam =

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

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

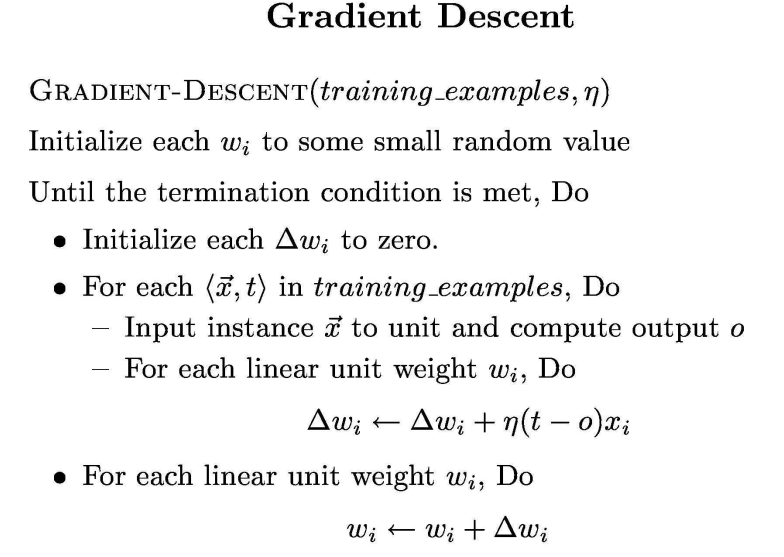
1.0E12 0.58

1.0E13 0.58

**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 ∆wi should be used.



∆wi should be equal to –ŋ () as before, but a different value should be used for.



should be derived as follows



=



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



=



= \*sorry those funny symbols should be vector signs



=



refers to the equation given in the problem:



This can be rewritten as  **\*\*see note below**

Thus, the derivative of this with respect to weight i, for any is



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 terms are in the expanded version of o. Well you can show that there is an such that is



equal to .

That value is in fact

**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.



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: 

The derivative of the right side of the equation with respect to  is 

Now ∆wi is equal to –ŋ () as before, but theis replaced by the above error euqation’s derivative.

So ∆wi is ). Or ().

So Wji = Wji + ()).

This is equal to . This is the udpate to be performed.

The second part of that is the original error update, ∆wi , 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.

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. 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/>.