University of Bristol

Introduction to Machine Learning

Project report

Naive Bayes Spam Filter

Authors:
Prisca Aeby, Alexis Semple

December 9, 2014

Abstract

With today's world becoming more and more flooded with e-mails, spammers are responding with an alarming increase of unsolicited messages. The natural reaction to this has been the development of many different spam filtering techniques, of varying efficiency and reliability. We are going to be looking more specifically at one of these techniques, Naive Bayes spam filtering, how it can be implemented and improved, and how it compares to other filtering techniques.

Contents

1 Introduction

Naive Bayes text classification is a popular technique used in spam e-mail filtering. It relies on correlating the use of tokens (in our case words) and then using Bayesian inference to calculate the probability that a given e-mail is spam or ham. [1] Naive Bayes classifiers suppose strong independence between the features of the model.

In the case of our project, the goal was to develop a binomial Naive Bayes text classifier written in Java to be tested and trained on real world data, or more specifically a large set of real world e-mails.

2 The Algorithm

We based the main algorithm of our spam filter on `LEARN_NAIVE_BAYES_TEXT` and `CLASSIFY_NAIVE_BAYES_TEXT` found in Tom Mitchell's book (on page

[1] Wikipedia on Naive Bayes spam filtering - http://www.wikiwand.com/en/Naive_Bayes_spam_filtering

183).[2] Our program is split into two parts.

2.1 Training

The file `filter_train.java` runs all of the computations and produces the results necessary to the supervised training part of the Naive Bayes text classification algorithm. In it, we iterate over all the files contained in a 'training set' directory and create a `java.util.HashMap` of all of the words therein. Each word is then a key in the HashMap and is associated with its value, a static array, each containing 4 precise values in the following order (`double[0..3]`):

- `double[0]`: The number of times the word has occurred in all files marked 'spam*' in the training data.
- `double[1]`: The number of times the word has occurred in all files marked 'ham*' in the training data.
- `double[2]`: The probability that a randomly drawn word from a document in class $Spam$ will be this word $w_k \rightarrow P(w_k|Spam)$
- `double[3]`: The probability that a randomly drawn word from a document in class Ham will be this word $w_k \rightarrow P(w_k|Ham)$

where we have $pSpam$ and $pHam$ respectively being computed by dividing the number of files marked spam* (or ham*) over the total number of training files. The values of `double[2]` and `double[3]` as explained above are computed as follows:

```
for(String word: vocabulary.keySet()){
   double[] tab = vocabulary.get(word);
   tab[2] = (tab[0] + 1) / ((double)(totalWords[0] +
       vocabulary.size()));
   tab[3] = (tab[1] + 1) / ((double)(totalWords[1] +
       vocabulary.size()));
   vocabulary.put(word, tab);
      }
```

[2] http://personal.disco.unimib.it/Vanneschi/McGrawHill_-_Machine_Learning_-Tom_Mitchell.pdf

where `totalWords[0..1]` contains the total number of words in all spam (resp. all ham) files and `vocabulary` is the HashMap of all words.

At the end of the program, the content of vocabulary is written to a text file named `trainging_data.txt` in a specific format: the first line contains $pSpam$ and $pHam$ in that order, separated by a space. Each of the following lines is dedicated to a single entry of the `vocabulary` HashMap, starting by the word itself, followed by the four values of its corresponding array, each separated by a space. The result looks something like this:

```
...
invested 7.0 0.0 2.088031403992316E-5 4.980206170575049E-7
...
```

meaning that for the word 'invested' we have that it appears 7 times in files marked 'spam*', 0 times in files marked 'ham*', has a probability of $2.088031403992316 * 10^{-5}$ of being drawn at random from class $Spam$ and a probability of $4.980206170575049 * 10^{-7}$ of being drawn at random from class Ham. One obvious observation is that these probabilities are extremely small and can pose problems of arithmetic underflow later on, in the classification part of the algorithm. The solution we implemented to avoid this is to use logarithms, which converts very small positive numbers into large negatives. So the code given above for computing `double[2]` and `double[3]` becomes

```
...
tab[2] = Math.log(tab[0] + 1) / ((double)(totalWords[0] +
    vocabulary.size()));
tab[3] = Math.log((tab[1] + 1) / ((double)(totalWords[1] +
    vocabulary.size()));
...
```

and the much more reasonable result in the text file is

```
...
invested 7.0 0.0 -10.776703754836734 -14.512624361064837
...
```

2.2 Classification

Once the training is completed, the program in `filter.java` can then use the results in `training_data.txt` to rebuild a Hashmap of `<word, double[]>` and classify instances provided to it. The formula provided in Mitchell's book returns an estimated target value for a given instance that is being tested. For spam

$$p_1 = pSpam \prod_{w_k \in vocabulary} P(w_k|Spam)$$

and for ham

$$p_2 = pHam \prod_{w_k \in vocabulary} P(w_k|Ham)$$

and the return value is

```
if(max(p1, p2) == p1) System.out.println("spam\n");
else System.out.println("ham\n");
```

However, since in the training part we modified $P(w_k|Spam)$ to $log(P(w_k|Spam))$ (resp. $P(w_k|Ham)$ to $log(P(w_k|Ham))$) we get

$$p_1 = log(pSpam) + \sum_{w_k \in vocabulary} log(P(w_k|Spam))$$

with the same going for p_2, obviously.