University of Bristol

Introduction to Machine Learning

Project report

Naive Bayes Spam Filter

Authors:
Prisca Aeby, Alexis Semple

December 11, 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

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

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 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:

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

```
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);
      }
```

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. According to the project description, we made the word selection process case-sensitive, and also decided to interpret any form of punctuation or whitespace as a word separator. So the scanner iterating over all words in file `f` behaves like such:

```
scanner = new Scanner(f).useDelimiter("[\\s\\p{Punct}]+");
```

The choice concerning punctuation was a personal one and based on the assumption that we had enough training data provided to ignore varying forms of punctuation. Results may have varied slightly had we taken these different forms into account, particularly considering use that is excessive and non-conform to standards which is dominant in some types of spam.

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. This determines whether a test instance is classified as spam (for $p_1 > p_2$) or ham (otherwise).

3 Improving the classifier and testing

3.1 Simple preprocessing

As indicated by the project specification, in order to test the consistency of the classifier and to evaluate its precision, we performed a 10-fold cross-validation on the provided training set, which means that we split the training set into 10 folds of equal size, and we did so randomly in order to obtain an approximately equal spread of members of both classes in every test sets. To each test set (10% of the original training set, i.e. 250 files) we then match the remaining 90% as a training set. So we end up being able to test our classifier on 10 different folds with 250 test instances and different training sets paired with them.

We have provided the bash scripts that we used to perform the random splitting into 10 test-training pairs (called `divide_train.bash`) and the cross-validation (called `cross_validation.bash`). The latter outputs a result file for each fold with the number of successful classifications and the total number of test instances. It also saves every misclassified instance to a separate folder for further examination.

3.1.1 10-fold cross-validation

To begin with, we ran a cross-validation on our Naive Bayesian classifier as described up to this point, that is to say with no text preprocessing steps and using whitespaces and any punctuation mark as word separators. Following this, we ran another cross-validation, this time considering all punctuation

	Punctuation as word separator		Punctuation as word part	
Fold	**Successful classifications**	**Success rate**	**Successful classifications**	**Success rate**
1	238	0.952	239	0.956
2	244	0.976	247	0.988
3	242	0.968	241	0.964
4	237	0.948	241	0.964
5	243	0.972	245	0.98
6	245	0.98	245	0.98
7	244	0.976	243	0.972
8	244	0.976	247	0.988
9	235	0.94	243	0.972
10	233	0.932	235	0.94
Mean	240.5	0.962	242.6	0.9704
StDev	4.3525216191	0.0174100865	3.7475918193	0.0149903673

Figure 1: Results of the first two 10-fold cross-validations. On the left, we have the Naive Bayes classifier using white spaces and punctuation as separators. On the right, Naive Bayes using only white spaces.

marks as part of the word rather than as a separator. The results can be seen in figure 1.

Paired t-test We can see that the second classifier seems to be slightly more efficient, with a slightly higher mean success rate (97.04%) and a lower standard deviation (≈ 0.015). To be able to conclusively decide whether this observed higher performance is due to chance or superiority of the classifier, we'll compute the *p-value* for this pair of results. We first state our null and alternative hypotheses:

$$H_0 : \text{true difference in means is equal to } 0$$

$$H_1 : \text{true difference in means is not equal to } 0$$

In other words, if our test accepts the null hypothesis, it indicates that the difference between the results of the two 10-fold cross-validations are due to random behaviour of the data. As is customary, we'll choose 95% confidence interval, so the test will decide whether or not the differences are significant at the $\alpha = 0.05$ level.

The *p-value* can be computed easily using R. The result for the following commands is then given as follows (NB is Naive Bayes, NBwp is Naive Bayes

with punctuation as word part):

```
> NB <- c(0.952, 0.976, 0.968, 0.948, 0.972, 0.980, 0.976, 0.976,
    0.94, 0.932)
> NBwp <- c(0.956, 0.988, 0.964, 0.964, 0.98, 0.98, 0.972, 0.988,
    0.972, 0.94)
> t.test(NB, NBwp, paired=TRUE)

Paired t-test

data: NB and NBwp
t = -2.473, df = 9, p-value = 0.0354
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.0160839381 -0.0007160619
sample estimates:
mean of the differences
               -0.0084
```

The R program gives us several results, e.g. the t-statistic (= -2.473), the degrees of freedom (= 9), the confidence interval with $\alpha = 0.05$, but most importantly, we can tell from the fact that the p-value is $0.0354 < 0.05$ that we can reject H_0 and say with a confidence of 95% that the difference between these classifiers is not due to chance. Considering punctuation marks as part of a word yields a better spam-filter, and this is the technique we adopted for our filter.

A reason for the superiority of this technique in this context could be the use of certain words paired with strong punctuation could be very decisive indicators for one class or the other (some simple examples could be 'viagra!', 'investment?', 'lottery!!!', etc.).

Friedman test In order to further test these results, we ran a Friedman test on the data, and obtained the following

```
> NB <- c(0.952, 0.976, 0.968, 0.948, 0.972, 0.98, 0.976, 0.976,
    0.94, 0.932)
> NBp <- c(0.956, 0.988, 0.964, 0.964, 0.98, 0.98, 0.972, 0.988,
    0.972, 0.94)
```

```
> F1 = matrix(
+ c(NB, NBp),
+ nrow=10,
+ ncol=2)

> friedman.test(F1)

   Friedman rank sum test

data: F1
Friedman chi-squared = 2.7778, df = 1, p-value = 0.09558
```

The Friedman test ranks each row of a matrix according to the highest performance. It then tests whether differences between the mean ranking for each column is significant or not (again here at level $\alpha = 0.05$). It has the following null hypothesis:

$$H_0 : \text{All algorithms perform equally.}$$

This can be rejected only if the critical value is less or equal to the Friedman statistic. In the results above, we can see that the Friedman statistic is 2.7778, and by looking it up in a chi-squared table for $\alpha = 0.05, df = 1$, we find a critical value of $3.84 \geq 2.7778$. So we have to accept H_0 which states that the two methods perform essentially with the same efficiency.

This can be seen to be contradictory to our previous results, since we had found that including punctuation marks into the words was seen to yield superior classification efficiency. However, we could also remark that in both tests, the result is relatively close to the threshold (i.e. the chosen α and critical value), so the outcome can legitimately vary from one test to the next.

3.2 Advanced preprocessing techniques

To continue to try and improve our filter, we implemented and tested some other preprocessing techniques that could result in higher success rates.

3.2.1 Stop-words removal

This technique consists in removing from the set of all words in the data some basic words that would result in inconclusive evidence to either the spam or ham class. In most cases, these words are common function words, but there is no defined standard set. In our case, we decided to remove the entries of our stop-words once our `vocabulary` HashMap was complete:

```
ArrayList<String> stopWords = new
    ArrayList<String>(Arrays.asList("a", "an", "and", "are", "as",
    "at", "be", "but", "by", "for", "if", "in", "into", "is", "it",
    "no", "not", "of", "on", "or", "such", "that", "the", "their",
    "then", "there", "these", "they", "this", "to", "was", "will",
    "with"));
        for(String word: vocabulary.keySet()){
                if(stopWords.contains(word)){
                        vocabulary.remove(word);
                }
```

3.2.2 Stemming

The stemming preprocessing technique consists in reducing words to their morphological root. This does not always result in the stem being the actual root of the word or it being an actual word. For example the words "argue", "argument", "arguing", "argues", "argus" would all be stemmed to "argu", which is neither a real world, nor the actual root of the listed words. [3]

We implemented this technique in our filter using the Snowball Stemmer for Java. [4] This resulted in us adding the stem of the word in our `vocabulary` collection, rather than the actual word. As a consequence, the training data would then be quite a lot smaller than without the stemming technique ($\approx$ 4.1 Mb rather than $\approx$ 4.7 Mb without stemming).

[3] Wikipedia on Stemming - http://www.wikiwand.com/en/Stemming

[4] The .jar can be downloaded from this website - http://trimc-nlp.blogspot.co.uk/2013/08/snowball-stemmer-for-java.html

	Naive Bayes		NB with stop-words removal	
Fold	**Successful classifications**	**Success rate**	**Successful classifications**	**Success rate**
1	239	0.956	239	0.956
2	247	0.988	246	0.984
3	241	0.964	244	0.976
4	241	0.964	242	0.968
5	245	0.98	246	0.984
6	245	0.98	243	0.972
7	243	0.972	242	0.968
8	247	0.988	247	0.988
9	243	0.972	244	0.976
10	235	0.94	236	0.944
Mean	242.6	0.9704	242.9	0.9716
StDev	3.7475918193	0.0149903673	3.3813212408	0.013525285

Figure 2: Results of the Naive Bayes and Naive Bayes with stop-words removal cross-validations.

3.2.3 10-fold cross-validation

We ran cross validations on three combinations of these techniques, using stop-words and stemming alone and then together. The results can be seen in figures 2 and 3.

Paired t-test As for the previous cross-validations, we want test these results at a level of significance of $\alpha = 0.05$. We ran the same R commands as previously, pairwise to compare with the Naive Bayes filter without any preprocessing and obtained the following:

- **NB - NB with stop-words removal**: $p - value = 0.5203$, H_0 is accepted.
- **NB - NB with stemming**: $p - value = 0.0001733$, H_0 is rejected.
- **NB - NB with stop-words removal and stemming**: $p - value = 0.0003579$, H_0 is accepted.

From this, we know that we can assume with a confidence level of 95% that stemming preprocessing has a negative impact on the performance of the filter, whereas stop-words removal seems to have a positive effect, but

	NB with stemming preprocessing		NB with stop-words removal and stemming preprocessing	
Fold	**Successful classifications**	**Success rate**	**Successful classifications**	**Success rate**
1	237	0.948	236	0.944
2	245	0.98	245	0.98
3	241	0.964	241	0.964
4	236	0.944	234	0.936
5	242	0.968	241	0.964
6	239	0.956	237	0.948
7	240	0.96	239	0.956
8	243	0.972	242	0.968
9	239	0.956	237	0.948
10	231	0.924	232	0.928
Mean	239.3	0.9572	238.4	0.9536
StDev	3.9735234854	0.0158940939	3.9496835316	0.0157987341

Figure 3: Results of the Naive Bayes with stemming and Naive Bayes with stemming and stop-words removal preprocessing.

the p-value points to the difference being due to chance and therefore non-conclusive.

A way to interpret the failure of stemming to improve the filter is to look at both the smaller size of the training data and the relatively weaker performance as an instance of generalisation of the data by the classifier, i.e. it reduces the words to a too general state and they lose information relevant to their probability of occurring in the respective classes of the model.

Friedman test Once again we ran a Friedman test on these results to see if it validates our findings.

- **Test on all four cross-validations:** Friedman statistic = 24.7419, for $\alpha = 0.05, df = 3$, critical value = 7.81, H_0 is rejected.
- **Test on NB and NB with stop-words removal:** Friedman statistic = 0.5, for $\alpha = 0.05, df = 1$, critical value = 3.84, H_0 is accepted.
- **Test on NB and NB with stemming:** Friedman statistic = 9, for $\alpha = 0.05, df = 1$, critical value = 3.84, H_0 is rejected.

Briefly summarised, these results concur with our t-test results, since they state that all 4 methods compared together do not perform equally, nor do

Naive Bayes without advanced preprocessing and Naive Bayes with stemming. However, when comparing the basic Naive Bayes with the one with stop-words removal, the null hypothesis is accepted, and the test states that both rank equally, with a level of confidence of 95%.

4 Other classifiers

In order to evaluate the efficiency of our classifier within the range of other widely used classifiers in the real world, we ran two more cross-validations, using two of WEKA's defined classifiers: the SMO implementation of support vector machine classifying, and the J48 implementation of Decision Tree classification.