Spam filter

Overview

In general, a pattern recognition system consists of three modules; i.e., pre-processing, feature extraction and classification. A spam filterer is a typical pattern recognition task, which needs to pre-process given emails for cleaning up data, extract salient features for distinguishing between ham and spam emails that form a feature vector for an email message and classify email messages based on feature vectors for making a decision. This report describes the work that we have done to implement the spam filter, as well as how we evaluate it and improve it.

Part1

In this part, we use bag-of-words model to extract features and implemented naïve Bayes classifiers for spam filtering emails based on a given dataset (400 ham emails and 100 spam emails). Then we use 10-fold cross validation to validate our classifier. Confusion matrix and average accuracy are used to assess the performance of our Naive Bayes classifier. We also compared the classifier with a baseline, which conducts a randomly guess when making a prediction.

1.1 Preprocessing emails

We firstly preprocessed the emails by eliminating any character else other than words from emails and attached a label to each email according to its file name (all instances were labelled as "0" (ham) or "1" (spam).

1.2 Extract features

Our method to get the feature vector of an email for training a classifier is bag-of-words. The bag-of-words model is a simplifying representation used in natural language processing and information retrieval (IR). In this model, a text is represented as the bag of its words, disregarding grammar and even word order but keeping multiplicity. [1] After transforming the text into a "bag of words", we characterise an email according to its term frequency e.g. the number of times a term appears in the email dataset given, and then get a term-document matrix (TDM). Columns of the TDM contain all the terms found in all of the emails. Every email will be a sparse feature vector where each word is one feature.
Feature extraction in our program was implemented using build in functions from sklearn.feature_extraction.text.CountVectorizer class, which converts a collection of text documents to a matrix of token counts. The number of features of an email will be equal to the vocabulary size found by analysing the data, which is 28373 in our case. This implementation produces a sparse matrix representing the counts. Specifically, we called fit_transform () function to learn and establish the vocabulary dictionary from our training emails and return term-document matrix, and called transform ( ) to transform testing emails to document-term matrix.

1.3 Naive Bayes multinomial classifier

The naive Bayes multinomial classifier we implemented makes a probabilistic model of data within each class.

The Naive Bayes algorithm can be divided into 2 phases, the learning phase and the testing phase.

In the learning phase: Given a training set S of F features and 2 classes (0 for ham, 1 for spam)

```
For each target value of $c_i$ ($c_i$=0, 1)
    P ($c_i$) <- estimate P ($c_i$) with examples in S;
    For every feature value $x_{jk}$ of each feature $x_j$ (j=1,…, F; k=1,…$N_j$)
P($x_j$=$x_{jk}$|$c_i$)<- estimate P( $x_{jk}$|$c_i$) with examples in S;
```

The output of the learning phase Is a conditional probabilistic (generative) models.

In the test phase: given an unknown instance x'= (a_1,…, a_n)

Calculate P=P(a_1|c_i)…P(a_n|c_i)]P(c_i) respectively, and assign x' to the class with larger P

1.4 10-fold Cross Validation:

We used 10-fold cross-validation to evaluate our classifier. The original 500 emails are randomly partitioned into 10 equal sized subsamples. Of the 10 subsamples, a single subsample is retained as the validating data, and the rest 9 subsamples are used as training data. The cross-validation process is then repeated 10 times, with each of the 10 subsamples used exactly once as the validating data. The 10 results from the folds can then be averaged to produce a single estimation.

1.5 Result Evaluation

Table 1.5.1 below shows the confusion matrix of Naive Bayes classifier VS confusion matrix of a random guess (baseline). Table 1.5.2 compares the precision, recall, f-1score of NB and the baseline.

	Predicted Ham	Predicted Spam
Labeled Ham	363	37
Labeled Spam	10	90

	Predicted Ham	Predicted Spam
Labeled Ham	316	84
Labeled Spam	74	26

Table1.5.1

	precision	recall	F1-score	support
0	0.97	0.91	0.94	400
1	0.71	0.90	0.79	100
Avg/total	0.92	0.91	0.91	500

	precision	recall	F1-score	support
0	0.81	0.79	0.80	400
1	0.24	0.26	0.25	100
Avg/total	0.70	0.68	0.69	500

Table1.5.2:

	Average accuracy
Naive Bayes classifier	90.6%
Baseline	68.4%

Table1.5.3

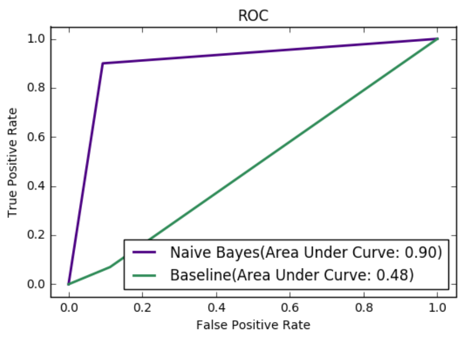


Figure1.1

Table 1.5.3 is the accuracy of Naive Bayes and the Baseline (random guess), our Naive Bayes achieved a noticeable high accuracy than a random guess. And from figuir1.1 we can see the overall performance of naive Bayes is pretty good.

Part2

In part2, we adopted a smarter feature extraction method, achieving a better prediction. We compared the result with and without this method and compared our Naive Bayes with KNN, Logistic Regression and baseline.

2.1 Smart feature processing techniques

2.1.1 The Drawback of Term Frequencies

The feature extraction method we adopted in part 1, term frequencies, are not necessarily the best representation of the text. Some words having highest term frequency in the text does not necessarily mean that the corresponding word is more important.

2.1.2 TF-IDF

To address the problem mentioned above, one of the most popular ways to "normalise" the term frequencies is to weight a term by the inverse of document frequency. Tf–idf is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus. [2] It is often used as a weighting factor in information retrieval and text mining. The tf-idf value increases proportionally to the number of times a word appears in the document, but is offset by the frequency of the word in the corpus, which helps to adjust for the fact that some words appear more frequently in general. [3]The inverse document frequency is a measure of how much information the word provides, that is, whether the term is common or rare across all documents. It is the logarithmically scaled inverse fraction of the documents that contain the word, obtained by dividing the total number of documents by the number of documents containing the term, and then taking the logarithm of that quotient. A high weight in tf–idf is reached by a high term frequency (in the given document) and a low document frequency of the term in the whole collection of documents; the weights hence tend to filter out common terms.
Tf-idf Feature extraction in our program was implemented using the build in functions from sklearn.feature_extraction.text. TfidfVectorizer class, which converts a collection of raw documents to a matrix of TF-IDF features. Equivalent to CountVectorizer followed by TfidfTransformer. SO the length of a feature vector will stay the same as it is used to be. Because the number of words is very large, the TF-idf dealt with are very small, many of them are approaching to 0. To address this problem, we conduct a $-$ log(tf-idf) operation for every entity in the matrix.

2.2 Compare the classification results with and without these techniques

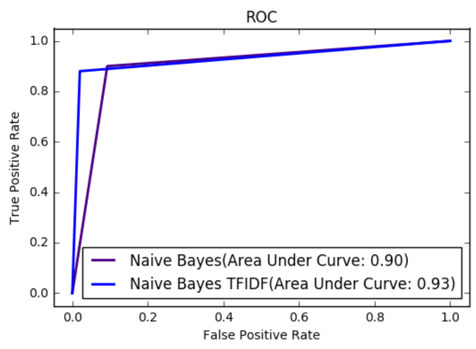


Figure2.2.1

The curve is created by plotting the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings. The ROC curve of a perfect classifier will go straight up the Y axis and then along the X axis. A random classifier will sit on the diagonal, whilst most classifiers fall somewhere in between. The AUC for a random guess classifier is 0.5 because the curve follows the diagonal. The AUC for that mythical being, the perfect classifier, is 1.0. Most classifiers have AUCs that fall somewhere between these two values. According to figure2.2.1, the classifier with TFIDF feature extractor has a larger AUC, which indicates that it has a better overall classification performance than the other. The classifier with TFIDF feature extractor enable to achieve a higher true positive rate and lower false positive rate than the classifier with term frequency feature extractor when threshold is settled

2.3 Compare results with KNN, LR, and baseline

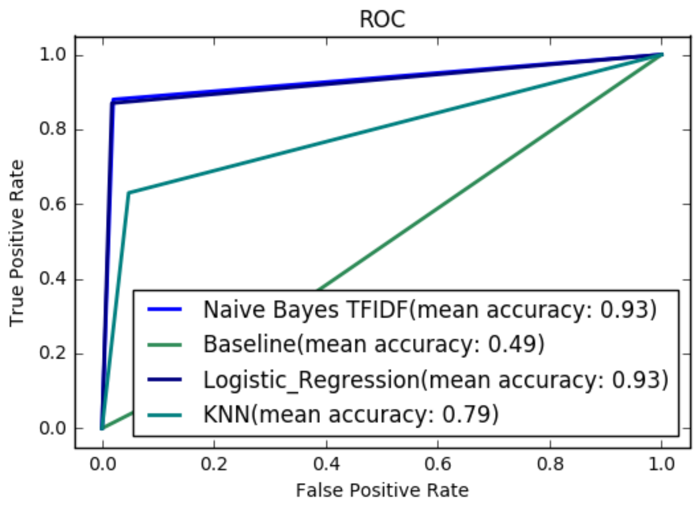


Figure2.3.1

Naive Bayes_TFIDF	Predicted Ham	Predicted Spam
Labeled Ham	392	8
Labeled Spam	12	88

Accuracy	**0.96**	
Logistic Regression	**Predicted Ham**	**Predicted Spam**
Labeled Ham	**393**	**7**
Labeled Spam	**13**	**87**
Accuracy	**0.958**	
KNN	**Predicted Ham**	**Predicted Spam**
Labeled Ham	**381**	**19**
Labeled Spam	**37**	**63**
Accuracy	**0.888**	

Table 2.3.1

algorithm		precision	recall	F1-score	support
NB	0	0.97	0.98	0.98	400
	1	0.92	0.88	0.90	100
	avg	0.96	0.96	0.96	
LR	0	0.97	0.98	0.98	400
	1	0.93	0.87	0.90	100
	avg	0.96	0.96	0.96	
KNN	0	0.91	0.95	0.93	400
	1	0.77	0.63	0.69	100
	avg	0.88	0.89	0.88	
random	0	0.80	0.91	0.85	400
	1	0.18	0.08	0.11	100
	avg	0.67	0.74	0.7	

Table2.3.2

Figure 2.3.1 compares the performance of Naive Bayes, Logistic regression, KNN and random guess. According to figure2.3.1, Naive Bayes and Logistic regression algorithms have similar noticeable better performances than random guess when sharing the same feature vectors of training data. And KNN falls somewhere between LR and random guess. Table2.3.1 is the confusion matrix, accuracy statistics of them. Table2.3.2 is the precision, recall and F1-score of these algorithms. According to Table2.3.2, all of these 4 algorithms have a higher F1-sore when detecting Ham. The reason for it is that ham training data we used is 4 times as many as spam training data.

Part3

In part3, we used Platt Scaling to calibrate Naive Bayes probabilities, which transforms the outputs of a classification model into a probability distribution over classes, achieving a lower mean squared error. We also improved our Naive Bayes spam filter by applying Adaboost algorithm.

3.1 Calibration

In binary classification, a prediction will return 1 or 0(happened or not happened).

However this is not accurate when we calculate or apply it to some learning algorithm.

However, Mean Square Error is not precise if we use returned values, which are 0s and 1s, to calculate it.

In this project, we use Platt calibration to fix this problem.

Firstly, 1,0 values are transformed to probability values t_+ and t_- correspondingly. t_+ and t_- are defined below:

$$t_+ = \frac{N_+ + 1}{N_+ + 2} \qquad t_- = \frac{1}{N_- + 2}$$

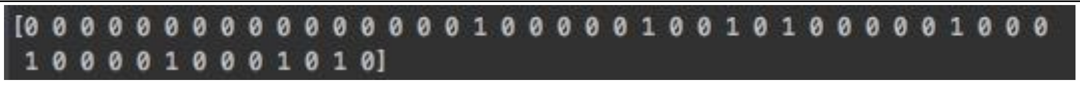
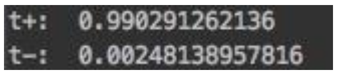
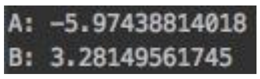
Where N+ represents the number of positive samples and N- represent the number of negative samples.

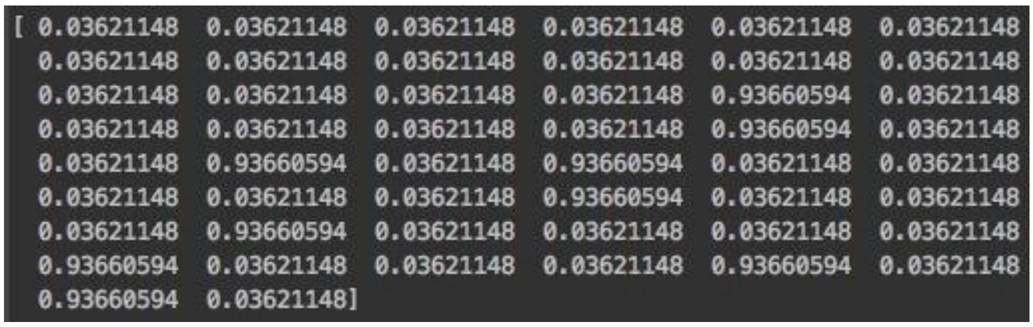
Secondly, t+ and t- are used as the parameters to estimated A and B by applying a maximum likelihood method.

Thirdly, the probability P can be calculated using this formula:

$$\mathrm{P}(y=1|x) = \frac{1}{1+\exp(Af(x)+B)}$$ [5]

And then we use P as predicted probability to calculate Mean Square Error.

Predicted value	 `[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0 1 0 0 0 0 0 1 0 0 0` `1 0 0 0 0 1 0 0 0 1 0 1 0]`
t+ t-	 `t+:  0.990291262136` `t-:  0.00248138957816`
A B	 `A: -5.97438814018` `B: 3.28149561745`

probabilities	

```
[ 0.03621148  0.03621148  0.03621148  0.03621148  0.03621148  0.03621148
  0.03621148  0.03621148  0.03621148  0.03621148  0.03621148  0.03621148
  0.03621148  0.03621148  0.03621148  0.03621148  0.93660594  0.03621148
  0.03621148  0.03621148  0.03621148  0.03621148  0.93660594  0.03621148
  0.03621148  0.93660594  0.03621148  0.93660594  0.03621148  0.03621148
  0.03621148  0.03621148  0.03621148  0.93660594  0.03621148  0.03621148
  0.03621148  0.93660594  0.03621148  0.03621148  0.03621148  0.03621148
  0.93660594  0.03621148  0.03621148  0.03621148  0.93660594  0.03621148
  0.93660594  0.03621148]
```

Table 3.1

Table 3.1 is the screenshots of the returns after running each step described above.

The calibrated mean squared error of the tf-idf Naive Bayes is 0.0363734848918, lower than the original mean squared error 0.0378486055777.

The calibrated mean squared error of the Naive Bayes implemented in part 1 is 0.0693902276232, lower than the original mean squared error 0.0876494023904.

3.2 Naive Bayes extension

The general idea of boosting is to learn a series of classifiers, where each classifier in the series pays more attention to the examples misclassified by its predecessor.[4] In other word, the classifier of boosting focus on wrong prediction examples. Specifically, after learning the classifier H_k, boosting modified the weight of training examples misclassified by H_k, and learns the next classifier H_{k+1} from the reweighted examples. This process is repeated for T (T=30 in our case) rounds. The final boosted classifier outputs a weighted sum of the outputs of the individual H_k, with each H_k weighted according to its accuracy on its training set. Figure 3.2.1 .1is the flowchart of our program.

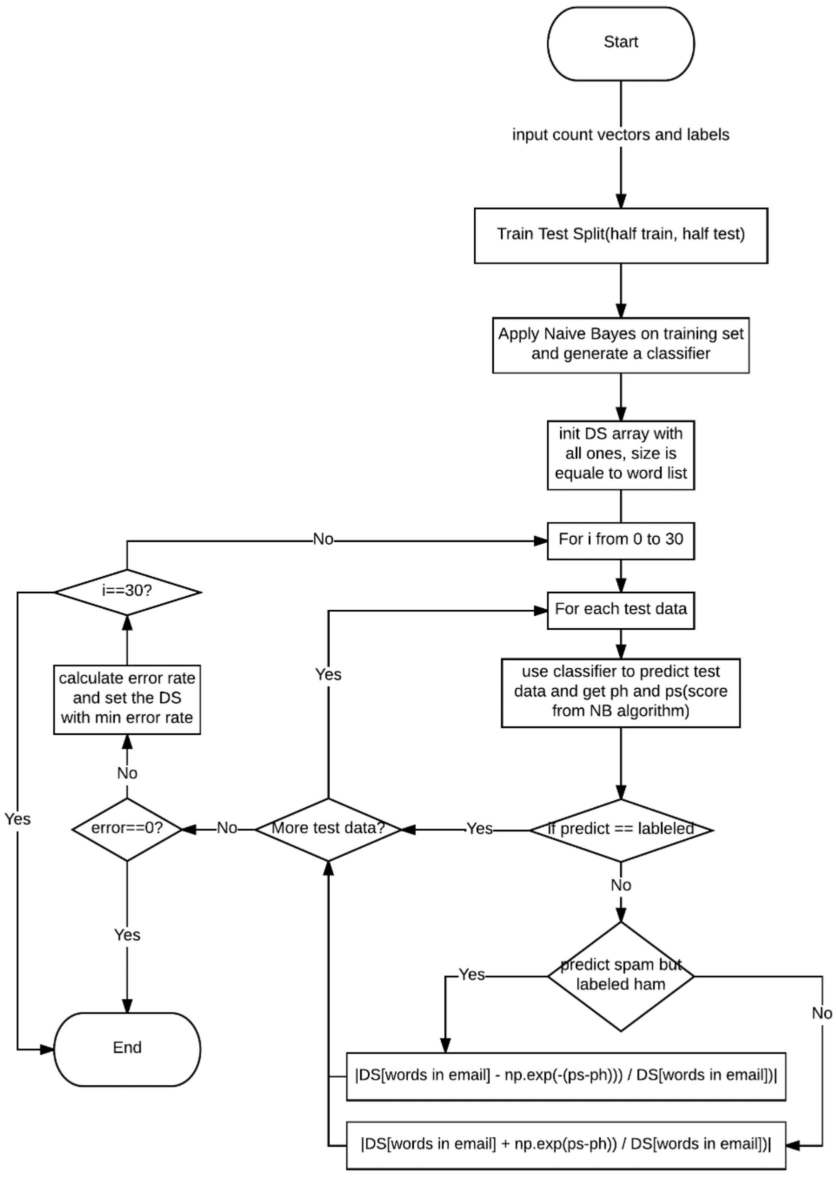
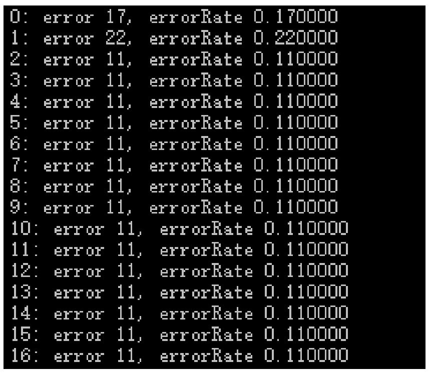
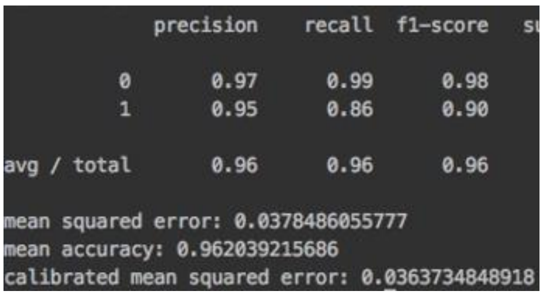

Start
input count vectors and labels
Train Test Split(half train, half test)
Apply Naive Bayes on training set and generate a classifier
init DS array with all ones, size is equale to word list
For i from 0 to 30
For each test data
use classifier to predict test data and get ph and ps(score from NB algorithm)
If predict == lableled
Yes
No
More test data?
Yes
No
error==0?
No
Yes
calculate error rate and set the DS with min error rate
i==30?
No
Yes
End
predict spam but labeled ham
Yes
No
|DS[words in email] - np.exp(-(ps-ph))) / DS[words in email])|
|DS[words in email] + np.exp(ps-ph)) / DS[words in email])|

Figure 3.2.1



```
0: error 17, errorRate 0.170000
1: error 22, errorRate 0.220000
2: error 11, errorRate 0.110000
3: error 11, errorRate 0.110000
4: error 11, errorRate 0.110000
5: error 11, errorRate 0.110000
6: error 11, errorRate 0.110000
7: error 11, errorRate 0.110000
8: error 11, errorRate 0.110000
9: error 11, errorRate 0.110000
10: error 11, errorRate 0.110000
11: error 11, errorRate 0.110000
12: error 11, errorRate 0.110000
13: error 11, errorRate 0.110000
14: error 11, errorRate 0.110000
15: error 11, errorRate 0.110000
16: error 11, errorRate 0.110000
```



```
             precision    recall  f1-score   su

          0       0.97      0.99      0.98
          1       0.95      0.86      0.90

avg / total       0.96      0.96      0.96

mean squared error: 0.0378486055777
mean accuracy: 0.962039215686
calibrated mean squared error: 0.0363734848918
```

Figure3.2.2

Figure 3.2.3

Figure 3.2.2 is a screenshot when the program was running. From it we can see, at the end of each training iteration, the number of wrong predictions (errors) is decreased and the error rate going down. During a training session, feature vector weight is adjusted to minimize errors.

Figure3.2.3 is the precision, recall, f1-score and confusion matrix of the classifier applied AdaBoost. We cannot see a noticeable improve because the training set (500 emails) is too small and the TFIDF Naive Bayes classifier has a very high accuracy, which leaves few errors for AdaBoost to learn from. If we apply the AdaBoost algorithm on a poorer classifier and give a big enough training set, the performance of it will be improved a lot.

Reference:

[1] https://en.wikipedia.org/wiki/Bag-of-words_model

[2] https://en.wikipedia.org/wiki/Tf%E2%80%93idf

[3] Leskovec J, Rajaraman A, Ullman J D. Mining of massive datasets[M]. Cambridge University Press, 2014.

[4] Elkan C. Boosting and naive Bayesian learning[R]. Technical Report CS97-557, University of California, San Diego, 1997.

[5] https://en.wikipedia.org/wiki/Platt_scaling