Implementation of a Spam Filter

Michael Aquilina, Student no.: 1329899, Account: ma13673
Uwe Korn, Student no.: 1348203, Account: uk13734

I. Introduction

The following report describes our implementation of a Spam Filter. The main aim of any Spam Filter is to distinguish emails into one of two classes - unwanted *Spam* emails and wanted *Ham* emails. The fight against spam has been on-going effort since the inception of Email technology and is still heavily researched in the field of machine learning to this day. Our Spam Filter is attempted in the form of a Naive Bayes classifier along with several pre-processing techniques to filter out noise and improve performance. Naive Bayes is a commonly used classification technique used for Spam Filtering, still in use till this day by popular email clients due to its simplicity and surprisingly good performance.

In this report we will be making use of a given labelled set of emails to perform training. This labelled set will also be used to evaluate the performance of our classifier using 10-fold cross validation. Each fold will be used to calculate most of the standard machine learning performance metrics such as accuracy, recall and precision and as a final measure we will be reporting performance metrics as the overall average between each of the 10 folds. The standard deviation is also derived from the results in order to provide a base for comparison between the different classifiers. As with all standard binary classification processes, we shall be using the terms positive and negative within the report to distinguish between the two possible classes - *Spam* as positive and *Ham* as negative.

To optimise the classification performance, we attempt to improve the quality of input data using various pre-processing techniques to filter out noise, normalise features and minimise the overall dimensionality. Doing so is an important step into achieving best results without altering the actual classification process itself.

A focus on our Naive Bayes implementation using the Bag of Words model will be made throughout the report. An emphasis on optimising the performance through the selection of correct parameters is also given throughout the various sections covered. However we shall also take the time at the end of the report to compare the performance of our implementation with another third party machine learning model - the C4.5 Decision Tree as implemented in WEKA as J48.

In addition to the quality of classification results, we will be taking a look at improving the runtime of the classification and training process. With a longer runtime, one might be able to increase the quality of a classifier significantly (e.g. doing an exhaustive search over all possible models) but the runtime required to do so could be infeasibly large. We therefore attempt to minimise the runtime of the training and classification process with little or (at best) no loss in classification quality.

All reported times are based on running our implementation on a dual core Intel i7-640L CPU with a frequency of 2.1GHz per core and 8Gb of main memory.

Our final classifier performs training and 10-fold cross validation in four minutes based on about 40 thousand terms extracted from the input data. On 10-fold cross validation we achieve an *Accuracy* of 0.984, *Precision* of 0.968 and a *Recall* of 0.942 which is significantly better than random. We show that even when using a weaker classifier that was trained on less input data, our Naive Bayes implementation still outperforms the J48 decision tree with high statistical significance in both speed and classification quality.

II. Naive Bayes

A Naive Bayes implementation was developed as Part 1 of our assignment and is used as our classifier throughout the rest of the report. We shall be using the Bag of Words model to extract features from emails, which treats each independent word in the document corpus as an individual feature. Our specific implementation of Naive Bayes makes use of a multinomial distribution in its underlying model which takes the frequency of each word

as its distribution rather than the binary value of simple existence. It is important to note that the in general, Naive Bayes models make the assumption that words are independently drawn from each other and does not model any dependencies between them.

The probabilities generated during training and classification are always stored as log-likelihoods to minimize numerical errors. Due to the low number of numerical errors, the Naive Bayes implementation is able to cope with a large number of 143820 dimensions (i.e. number of different words in the input data) as its initial feature set.

To obtain an initial baseline performance, we ran the Naive Bayes classifier on the given training set using 10-fold cross validation. This resulted in an accuracy of 0.944 and a standard deviation across all folds of 0.024 without using the underlying class distribution as a-priori probabilities. With the usage of the underlying distribution, we get a classification accuracy of 0.9472 and a standard deviation of 0.032.

Although the class distribution is anything but uniform (81% Ham, 19% Spam), the resulting classification score is not significantly biased to either side. As the feature size is not affected by the prior distribution, changing it will not make a difference to the overall runtime. We use maximum-a-posteriori for our Naive Bayes implementation, which is the most commonly used technique. We use the class distribution as the a-priori distribution.

We will now use the next section of our report to explain the optimisations used in our implementation to improve classification performance using various preprocessing techniques.

III. Preprocessing

The bag of words model used by our classifiers is a good way to represent the data found in the given training documents. However this feature representation is prone to very high dimensions if not handled well due to the large number of possible combinations in words.

According to a recent survey performed by Google and Harvard, there are approximately 1,022,000 different words in the English language [10]. While we do not expect to find all possible words in the document corpus, a large number of them will be found along with their derivations in the form of spelling mistakes and grammatical variations (e.g.*they're* and *theyre*). Other non-English words such as HTML code, URLs and seemingly nonsensical data such as PGP keys will also be found within the documents.

Most of these words will provide no benefit to the classifier and in most cases will even harm the classifier's performance. It is therefore of utmost importance that noise in the text is filtered out and the number of word combinations is reduced to the most representative, yet minimal subset of the available words. Once this is done, steps can be taken to transform each document into a feature vector that can be understood by the classifier.

The following steps are taken to do this:

- Smart Email Parsing
- Simple Text Processing Techniques
- Domain Extraction
- Stemming
- Inverted Index Storage
- Feature Selection based on Document Frequency
- Vector Generation / Feature Weighting

Each of these steps contribute towards achieving a higher accuracy and performance in the classifier and will be described in detail in the sections below.

A. Email Parsing

As a first preprocessing step, we make use of a modified parser to extract information from each email's structure. The provided emails will sometimes contain metadata which is very redundant w.r.t. to the actual class. An example of this is the occurrence of "From:" which is contained within the headers of all emails. The *presence* of metadata in emails is in fact independent from any class label, i.e. the classifier should not use it its presence as a feature for distinguishing between classes of emails. On the other hand the information *contained* within the metadata itself can be very useful in determining the class if available, but such a feature will be left for future work due to time constraints. Finally emails do not always consist of just text but may also have attachments encoded plain text or beautified with HTML markup.

As a first step we strip most of the metadata from the email so that we only return the actual content of the email to the classifier. This drastically reduces the amount of data passed to the classifier as most terms in the metadata are either common among all emails, e.g. the names of the header lines

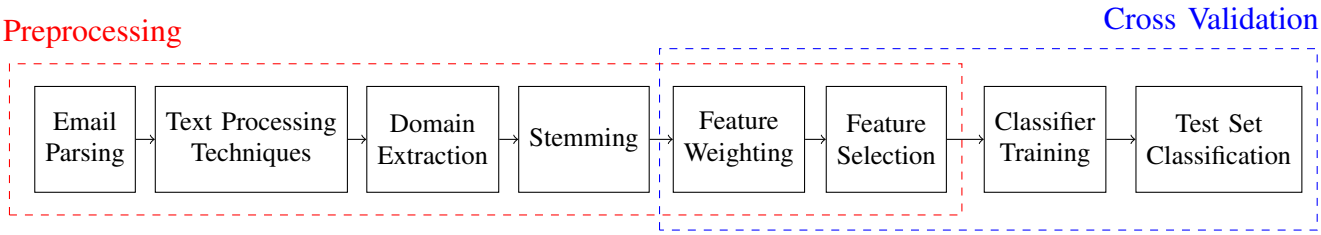

Fig. 1: Steps taken during training and cross validation

("From", "Content-Type", ...), or are completely unique to the mail itself like the date of sending or the message-id. From the metadata we only extract some basic information like the encoding or flags stating if the email is multi-part.

If the mail is split into multiple parts, we only extract parts which contain plain text as we cannot infer information from the *base64* representation of images and other binary attachments. These binary attachments are represented in the form of seemingly random text (to the classifier) and would only add a large number of garbled text [4] to the resultant index. Such combinations of text are very unlikely to be found in other emails due to high entropy within binary data. Another problem that is solved by handling multiple parts is the recursive parsing of attached emails. For each embedded email, we strip down each inner email to just their content as we have just described in this section.

Furthermore not all text within the email is encoded using same encoding as the declared by the email itself. For example the main text could be encoded as Base64 which would not reveal the actual words when just splitting the mail into words by separating at each space. Another typical encoding is quoted-printable which transforms non-ASCII characters into another format (so called escape characters) but leaves all ASCII characters intact. Handling these multiple encodings in the correct fashion transforms meaningless, long character sequences into words that are present in other emails and useful for the classifier. As Java does not have built-in support for transforming these encodings, we are utilising Apache Commons Codes [1] to handle the transformation for us.

As a final parsing step we strip down as much HTML content as possible from the emails' content in order to prevent HTML tags from entering our word index. This markup is not displayed to the user and does not carry any useful content for the parser or classifier to make use of. For extracting the text from the HTML document we are using the Java Library jsoup [9] which provides error-resistant parsing of even non-standard HTML.

Although "smart" email parsing did not cause any significant increase in the quality scores of the classifier, it did manage to reduce the dimensionality from 143820 down to 94903 words. As a consequence of this lower dimensionality the runtime decreases from 16 to just 10 minutes. In conclusion email parsing is responsible for stripping down a lot of irrelevant or redundant dependencies from the input data set without the cost of any loss in quality.

B. *Simple Text Processing Techniques*

A number of simple processing techniques are used to conflate strings being returned by the parser so that variations of the same word are mapped to the same feature. Although the email parser is built in such a way as to strip out HTML content where possible, a number of artefacts could still linger in the data which could cause noise and incorrectly distinguish variations of the same word. Because it is not the parser's job to perform this form of filtering, a separate step is taken to perform simple text processing tasks before passing them on the later pre-processing stages.

All incoming words that are composed purely of symbols (i.e. no numbers or letters) are simply discarded as they are most commonly noise in the data that do not represent anything in the corpus. Without this step, the number of features used by the classifier would grow substantially as a large number of "rubbish" symbols are included as part of the feature set. Additionally, all remaining words will have all symbols removed as this prevents variations in words like *they're* and *theyre* from being treated as different features.

As a second step, all valid incoming words are simply reduced to lower-case format. The same word in different cases should not be distinguished from one another during classification so it is important that e.g. *Bristol* is considered the same as *bristol*.

Finally, variations in number representations (e.g. 4,000 and 4000) are detected using regular expressions and conflated to a single representative feature "9999". We do this because allowing all possible number combinations a unique feature each will increase the dimensionality substantially and rarely contribute to classification performance. Doing this also provides some assurance that we do not overfit our classifier to the training set with specific numeric combinations.

Using the techniques described above we were able to reduce the number of combinations from 94,903 words to just 56149. This provides a substantial decrease in dimensions and provides a very notable improvement to memory usage, training speed and classification speed.

C. Domain Extraction

Due to the fact that Emails are a product of the web, it is very common to find items such as Uniform Resource Locators (URLs) and email addresses within the their contents. When left unprocessed, it is hard to group these items into corresponding features due to the additional data included within them.

What we really care about is the domain each URL and email address contains. If we find two URLs which point to the same domain such as *www.cs.bris.ac.uk/maths* and *www.cs.bris.ac.uk/eng*, then it would make sense to group these two items together as one feature - i.e. coming from the domain *cs.bris.ac.uk*. This will prove useful in being able to identify URLs which are associated with spam and those which are well known and credible ham domains.

To perform this step, we use a regular expression parser to detect items that are URLs and email addresses. Detected items are put through a *Domain Extraction* function which is able to strip out the username and '@' symbol in the case of emails (e.g. *johndoe@example.com* would become *example.com*) and remove paths and protocol declarations in the case of URLs (e.g. *https://www.example.com/myimage.jpg* would become *example.com* once again). Conflating both email address domains and web URL domains makes sense because they represent the same source of information.

D. Stemming

Most words in the English language are derived from a *morphological root* word that contains no prefixes or suffixes and conveys a very similar meaning to its derivation. A simple example of this is *subscriber* and *subscription* with their morphological root *subscribe*.

If we can reduce all incoming words into their root form, we would be able to substantially reduce the number of dimensions for our model while also ensuring that words representing the same underlying feature are stored under the same value.

Unfortunately, such a task is quite hard and would probably require the creation of a very large lookup table for each word in the English language along with its root. This is due to the fact that the English language is not a formal language and hence does not follow a strict set of rules.

We could however, take an approximation of the described process and instead derive the *stem* of each word. Like the morphological root, the stem is a representation of a words underlying meaning. However the stem does not guarantee to be a correct English word or generate the right root as its aim is simply to map variations of the same word to the same to item.

For our Spam Filter implementation, we made use of the Porter Stemmer algorithm [3], which in the authors words is "a process for removing the commoner morphological and inflexional endings from words in English". In simpler terms, it is capable of removing known suffixes from words passed to it. The Porter Stemmer algorithm is available as open source code under the BSD License and is available in multiple languages, including Java which is made use of in our implementation.

Using the same examples shown before, passing *subscriber* and *subscribe* to the Porter Stemmer would reduce the words to the stem *subscrib*. On the other hand however, the word *subscription* will be wrongly mapped to a different stem - *subscript*. The latter is an example of where the approximation fails to produce the correct result, however in general

most words passed to the algorithm have shown to produce favourable results.

In terms of the Spam Filter implementation, using Porter Stemming on the given set of training emails reduced the number of words from 24813 words to 18932 *stems* (both after text pre-processing). This is a substantial reduction in the number of dimensions and plays a crucial role in ensuring that the classifier is able to train with the given documents in a short amount of time and without requiring large amounts of memory.

E. Inverted Index Storage

After a word has gone through the stages of simple text processing, domain extraction and stemming - it is stored along with the name of its document of origin in a fast inverted index structure using a Hashed List. The Hash List is built so that a given word is an index that is able to quickly retrieve or update information about its metadata in constant $\mathcal{O}(1)$ time. Metadata included for each term includes a Hashed Map from a *document name* to the frequency of the term in the document.

As more words are added over time, the inverted index will accumulate a list of frequencies for each word in several documents. Once all words have been passed, information about the corpus such as each word's total frequency and its frequency within each document can easily be calculated from the underlying metadata. The information stored is especially important for the next two stages that will occur - Feature Selection and Feature Weighting.

F. Feature selection

Although we have substantially reduced the number of features through the use of various text processing techniques, we still have a substantial number of features for the classifier to train with. Interestingly, the frequency of occurrence for each word in the text can be grouped into one of 3 general categories of *very common*, *common* and *rare*.

Very frequent elements in the text and commonly referred to as stopwords in linguistic morphology. These are words which are very frequently found in English texts (Some good examples are *the*, *and*, *it* etc..) and usually do not provide any particular indication about the category of the document's class. On the other hand, very rare elements in a text are usually attributed to spelling mistakes and other artefacts such as IDs (PGP keys being the most common in emails). In between these two categories we have words not too commonly found, but not rare enough to constitute as errors either.

A corpus of documents that shows this pattern is said to obey *Zipf's law* and thus follow what is called a *Zipfian Distribution*. It so happens that most English documents follow this law and so does our corpus. This pattern occurs in English texts mainly due to a phenomenon pointed out by Zipf in his paper which he calls *the principle of least effort* [12].

The frequency of occurrence of each term in the given training set is shown in Figure 2 mapped in descending order. Some words on the graph are specified on the x-axis to provide an sample at each portion of the graph (Note that these features are stems rather than words due to our previous Stemming technique). It should be immediately noted how many different rare words are found within the corpus with low frequency by the long "tail" shown in the graph. On the other hand one should also notice how very frequent words such as *the* do not have many combinations when compared to the infrequent words.

What we really want to keep is the words which are in between these two categories as these are the most representative of the text. This area can be considered somewhat of a *"Goldilocks Zone"* as its just about right for giving us the right information. To extract this area and remove the excess, we use a technique called *feature selection* to trim out all features from the index with a document frequency less than $\alpha * no.\,documents$ and more than $\beta * no.\,documents$, where α and β are values in the range $[0, 1]$.

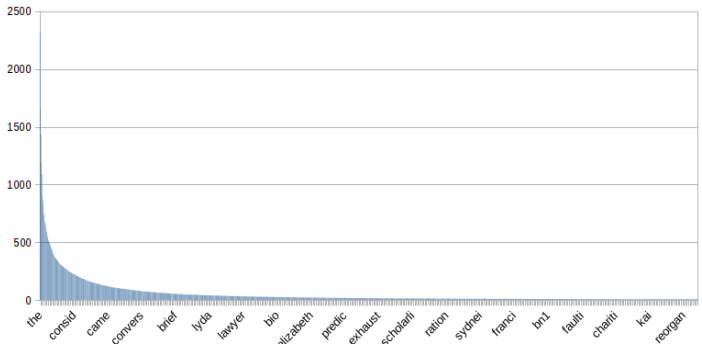

Fig. 2: Zipfian Distribution exhibited in the Training Corpus

Unfortunately, choosing ideal α and β is a somewhat tedious process of trial and error. Finding the right values ensures that the classifier does not overfit on features that are only available in the training process or accidentally strips out features which are useful to the classifier. In Section III-H, we will try to optimise our selected thresholds by the empirical evaluation of the performance of the classifier on our data set.

G. Vector Generation / Feature Weighting

Now that we have filtered out all unnecessary features and determined which features should be kept for classification, we need to convert email documents into data that is understandable by the Classifier. More specifically, we need to convert each email into a corresponding feature vector using the features we have just identified. In this report, we have attempted two ways of converting frequency information found in emails into corresponding weights.

The first and most basic attempt uses *Frequency Weighting* which is simply assigning each feature in a vector the frequency of the corresponding term in the email. Although seemingly simple as an approach, it has proven to be an effective representation of each email in the corpus during the writing of this report.

The second approach attempted was *Term Frequency Inverse Document Frequency* also commonly known as *tfidf* [8]. Tfidf is a commonly used metric in natural language processing that portrays the importance of a word in a document with respect to the rest of the corpus. It is technique often used by Search Engines and has proven many times to be a reliable way of representing the significance of a term within a document. The equation for calculating the weight for a term i in a document j is given as:

$$\text{tfidf}_{ij} = \frac{f_{ij}}{\text{maxf}} * \log\left(\frac{N}{n_i}\right)$$

Where N represents the number of documents, n_i represents the number of times the term i appears in a document and *maxf* represents the highest frequency in the document corpus.

Interestingly, after testing these two approaches, it was immediately noted that *tfidf* gave significantly worse results than *frequency weighting*. While this was initially a surprise, It was noted that this behaviour could possibly be occurring due to use of a multinomial Naive Bayes model rather than Bernoulli model. Further investigation as to why this is the case could be a possible area for future work on our implementation.

We shall assume the use of Frequency Weighting for the rest of the report due to its superior performance. Note that an implementation of the tfidf weighting scheme is still included as part of this reports implementation but is left disabled in favour of frequency weighting.

H. Optimising Feature Thresholds

After all preprocessing steps have been performed, the total number of dimensions has now been brought down to 43437 different words. Although this is less than a third of the initial dimensionality, it is slowing down the training of the classifier as well as the classification of the samples. A high number of features can also have a negative impact on quality of the classification results as the classifier may learn to concentrate on words that are rare and only appear in the training set while leaving out features that could separate the two classes very well.

To increase the quality and the runtime, we need to remove infrequent and very frequent words. However we still need to keep semi-frequent features without having a negative effect on the classification quality. As an exhaustive search of all possible feature combination is infeasible, we will try to empirically find the two thresholds α and β. We do this by setting one of the thresholds to a fixed value while iterating over possible values for the other threshold.

To compare the impact on the behaviour of the classification of both classes *Spam* and *Ham* we are using the measures *Precision*, *Recall* and *Negative Recall* as they are less dependent of the class distribution in the test set than *Accuracy*.

In Figure 3, we have set $\alpha = 0$ to have all lower frequent words in the training and iterating over β to empirically determine which high frequent words can be removed without an effect on the cluster quality. As seen in the graph, for a value of β larger than 0.19 all three quality measures remain constant. In the interval between 0.05 and 0.2 there is a high variance of the scores, so we increased the resolution

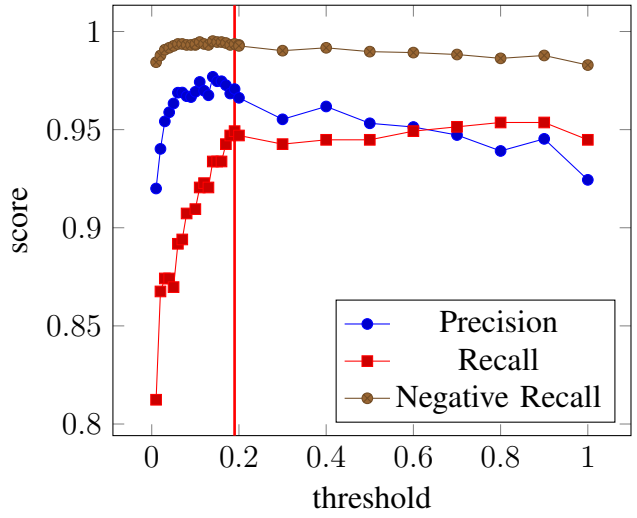

Fig. 3: Accuracy w.r.t. upper threshold (lower threshold = 0)

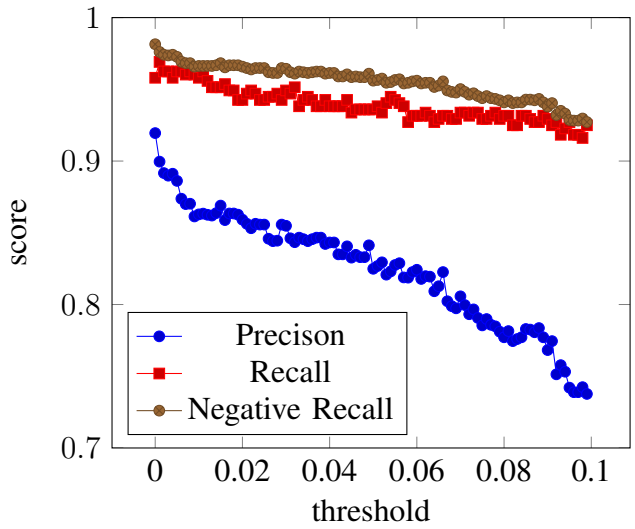

Fig. 4: Precision and recall w.r.t. lower threshold (no cap on the upper bound)

in the plot to a simulation in each 0.01 step to better view the behaviour of the classifier. As a final choice for β we have taken $\beta = 0.19$ as *Recall* has its maximum here and decreases steep towards $\beta = 0$. The other measures are at a very good level (near their maximum) for $\beta = 0.19$ and only increase slightly in this neighbourhood.

This choice of β reduces the dimensionality down to 43331 by pruning 106 very frequent words. These pruned words are mostly stopwords which do not provide any indication about the class of the mail as they occur in all kinds of text. As only a low number of words were reduced, the runtime stays constant at six minutes.

Figure 4 shows the behaviour of the quality measures if we would evaluate the classification performance with $\beta = 1$ to determine the best lower threshold. As we already have chosen an upper threshold of $\beta = 0.19$, we are taking this into account in Figure 5. Using the upper threshold leads to slightly different behaviour were *Precision* still decreases when increasing the threshold to a lower level. Whereas in the situation $\beta = 1$ *Recall* remained at a near constant level among all parameter choices, it decreases with the same pace as *Precision*. Overall it should be remarked that *Precision* is at a much higher level for $\alpha = 0$ in Figure 5 than in Figure 4 whilst *Recall* is at approximately the same level in both situations.

In both situations, we get the best scores for $\alpha = 0$, which is no further improvement on the selection of the upper threshold. This means that there is a significant number of words that occur only once in the training set but which still help separate classes in the test set. It should be noted that not all words with a frequency of one in the training set are relevant but there are enough relevant words with a frequency of one that would have a significant impact on the scores if pruned.

Although the best results were achieved using $\alpha = 0$, for $\alpha = 0.17$ we can still generate a classifier that produces very good results with a *Precision* of 0.916 and a *Recall* of 0.944. As this classifier only utilises 1568 dimensions, which is about 4% of the classifier with $\alpha = 0$, it runs the cross validation cycle in less then 100 seconds.

IV. Storage

In order to save the state of the machine for re-use at later stage, a method for storing the model to disk was implemented. After training, the model is automatically saved to disk in JSON format which provides a simple and human readable way of storing results. This file is read back into memory during classification when the *filter* class is run. We make use of Google's *Gson* open source Java library [5] for automatically reading and writing Java classes to disk.

Making use of a JSON storage format provided us with the additional benefit of being able to analyse the features that were selected in a human

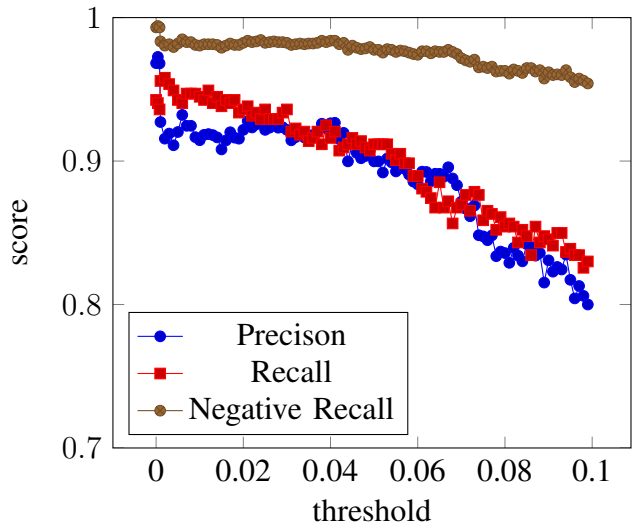

Fig. 5: Precision and recall w.r.t. lower threshold ($\beta = 0.19$)

readable format. Each time a change was made to the Spam Filter implementation, the output JSON file was analysed to check for possible areas of improvement. A number of optimisations mentioned in this document were due to such analysis.

V. Alternatives to Naive Bayes

As a classifier is modelled after some given assumptions and with specific architectural proprieties, e.g. all attributes are equally important, it is important to check if the used classifier is the right model for the examined problem. A very good feature of the Naive Bayes classifier is its ability to train on a dataset with a very high dimensionality in decent time. Because this property is not shared among all other classifier and some even severely suffer from high dimensionality (the so called Curse of Dimensionality [2]), we are using the reduced set of features we estimated in the feature selection process without further research on the impact of other thresholds on feature trimming w.r.t. to non-bayesian classifiers.

A. Decision Trees

We are going to use Weka's [6] implementation of the C4.5 decision tree [11] (known in Weka as J48) as the first alternative to Naive Bayes classification. C4.5 builds a tree based on the features using information entropy. The dimension with the highest information gain will be used at each node to generate a new split. After the creation of the tree, C4.5 tries to prune all unnecessary branches into simple leafs to keep the total number of nodes to a minimum. Although this feature could be disabled, we are using it to avoid overfitting.

As the runtime of C4.5 increases in a higher level with the number of dimensions than the Naive Bayes classifier, we are going to use it with the thresholds $\alpha = 0.017$ and $\beta = 0.19$. This reduced data set of only 1568 dimensions can be used to train a C4.5 tree in a feasible amount of time.

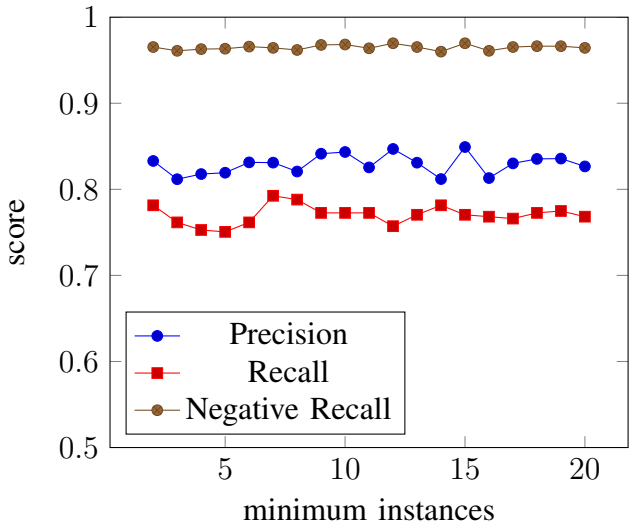

Fig. 6: Performance of J48 w.r.t. the number of minimum instances per leaf

One of the parameters that can be adjusted is the number of instances each leaf of the tree needs to represent at least during the training phase. The default setting of WEKA requires at minimum two instances at each leaf, in Figure 6 we outline the scores with varying number of minimum instances. From the results we can see that changing the minimum number of instances has not a large impact on the score of the classifier, scores stay rather the same. Because of this we are going to use WEKA's default setting of a minimum of two instances per leaf for comparing the performance of J48 to our best Naive Bayes instantiation.

To compare the performance of both classifiers, we are testing the null hypothesis that they generate equal good results. The testing will be done using 10-fold cross validation on the same training set and the statistical significance of the results will be

examined using Wilcoxon's signed-rank test. As we already have used different measures throughout the whole document, we will be testing the performance of both classifiers using *Accuracy*, *Precision* and *Recall*. For the statistical test, we are using 5% as our confidence level, i.e. to successfully reject the null hypothesis that both classifiers produce similar results, the minimum sum of either positive or negative ranks must be eight or less.

Measure	**Difference**	$\min(\sum +ranks, \sum -ranks)$
Accuracy	0.048	0
Precision	0.111	1
Recall	0.167	0

TABLE I: Average differences in the quality scores and minimum sum of either positive or negative ranks of Naive Bayes and J48 based on 10-fold cross validation

Table I shows the average differences between both classifiers and shows clearly that on average Naive Bayes performs better. With all ranks below 8, we can reject the null hypothesis for all measures that the classifiers produce similar results. As the scores of Naive Bayes were always better, we can conclude that Naive Bayes outperforms C4.5/J48 for our dataset of email data for the given feature vectors. Remarkably J48 could only outperform Naive Bayes in one fold and in this fold it was only better in *Precision* where Naive Bayes still lead to better scores in *Accuracy* and *Recall*.

VI. Conclusion

The Naive Bayes classifier proved to be a surprisingly efficient machine learning technique despite its aptly given name (It is sometimes even lovingly referred to as Idiots Bayes due to its simplicity [7]). It has proven to output results with high accuracy, precision and recall - even outperforming more advanced machine learning techniques such as Decision Trees. With a final accuracy of 0.984 and precision of 96.8 calculated through cross validation, our email classifier should correctly distinguish between ham and spam emails on a number of different test sets.

Future work can be done to both improve preprocessing techniques by further analysing metadata within the headers and including additional text processing techniques such as spell correction to conflate words further into their respective features. Further natural language processing techniques can also be attempted on the text and a comparison in performance to the Bernoulli model for Naive Bayes can also be attempted in future work.

References

[1] Apache. Commons codecs. http://commons.apache.org/proper/commons-codec/.

[2] R. Bellman. Dynamic programming, princeton. *NJ: Princeton UP*, 1957.

[3] M.F. Porter C.J. van Rijsbergen, S.E. Robertson. New models in probabilistic information retrieval. *British Library Research and Development Report, no. 5587*, 1980.

[4] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. RFC 2045 (Draft Standard), November 1996. Updated by RFCs 2184, 2231, 5335, 6532.

[5] Google. Gson. https://code.google.com/p/google-gson/.

[6] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[7] Yu K Hand, D.J. Idiots bayes - not so stupid after all? *International Statistical Review*, 69:385–399, 2001.

[8] K. S. Jones. Index term weighting. *Information Storage and Retrieval*, pages 619–633, 1973.

[9] jsoup. Java html parser. http://jsoup.org/.

[10] Jean-Baptiste Michel, Yuan Kui Shen, Aviva Presser Aiden, and The Google Books Team. Quantitative analysis of culture using millions of digitized books. *Sience*, 331(6014):176–182, 2010.

[11] Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[12] George K. Zipf. Human behavior and the principle of least effort, 1949.