CS918 - EXERCISE TWO - REPORT

RICHARD COLLINS

1. Overview

Three sentiment classifiers were built to classify tweets into three classes: *positive*, *neutral*, and *negative*. The classifiers leverage two main categories of features in the tweets. The first are features pertaining to the structure of the text i.e. hashtags, laughing text (haha, lol...), elongated words etc. The second feature category is the sentiment of individual words in the text. The general principle governing the development of these classifiers was to use as few features as possible to keep the classifiers simple, and to reduce processing time. It was therefore decided to reject features such as unigrams, bigrams, and word embeddings. Random forest models with tuned parameters were implemented to perform the fitting of training data, and prediction of test data labels.

2. Workflow and Explanation of Code

Lines 1 - 15. .
A flow chart representing the work flow of the code can be found in appendix A. As is typical for most python programs, all the necessary packages are imported at the head of the code. This python program is no exception, making sure to import the all functions and definitions from `evaluation.py` and `testsets.py`.

Lines 18 - 38. .
The next section fo code represents the definitions of all the regular expressions used to extract features from the tweets. Special mention must go to `regExEmoji`, which lists the unicode representation of all emojis [1]. Emojis are grouped in ranges of hexadecimals. Unfortunately, there is not a range for emojis with a particular emotion; one would have to define each emoji separately. There are thousands of emojis in popular use, so defining each individually would be too cumbersome.

It is a similar situation with the handling of emojis composed of punctuation (matched using the `regExPosSmile` and `regExNegSmile` regular expressions). The entire set of all of these characters was far too cumbersome to implement fully, so a selection fo the most frequently found emojis were defined.

Lines 41 - 95. .
Functions were then defined to be able to take in a string and replace all instances matched using a regular expression with a placeholder text. Fig. 1 lists the set of extracted features from the tweets, and the text replacing the features.

The spaces either side fo the replacement string are to ensure that it is not joined to adjacent words in the tweet. This is a particular issue with emojis where, in the original tweet, no space is left between the emoji and the adjacent word. If the replacement string is joined to another

Feature	Example	Replacement Text
URL links	http://t.co	urllink
User mentions	@user	usermention
Hashtags	#hashtag	hashtag
Emojis	🙂 😀	emoji
Elongated words	whyyyy	elongatedword
Laugh onomatopoeia	haha, hehe, lol	laughtext
All uppercased letters	NEVER, WOW	allcaps
Mixture of questions/exclamations	??!!, ?!?!	questionexclaim
Three-dot continuation	...	ellipsis
Happy face from punctuation	:), :D, :p, ;)	positiveface
Unhappy face from punctuation	:(, :'(, :-c, :-/	negativeface

FIGURE 1. List of features and their replacement text

word, it will be considered as a unique unigram when the term frequency is calculated, which would therefore reduce the number of instances observed for that particular feature.

Also defined in this section is the tokenise(doc) function. This function takes in a document (tweet) and returns a list of tokens. The tokeniser is based on the `TweetTokenizer` from the `nltk.tokenize` package. It is specific to Twitter in the sense that it does not separate hashtags from words, and keeps intact URL links; two vital features used in this exercise.

Lines 99 - 109. .
The `getUnigrams(vec, training_corpus = None, testing_corpus = None)` function takes in a count vectoriser and a corpus of text (a list of strings). The output is an array of size $D \times V$ where D is the number of tweets and V is the vocabulary size. Two non-essential arguments are defined, the first is used or training data to generate the initial array, and the second to transform the testing data to include vocabulary terms that only appear in both training and testing data. `TfidfVectorizer(stop_words = ‘‘english")` was used for the vectoriser instead of `CountVectorizer()` because it penalises words that occur frequently in the corpus, therefore giving more importance to rarely occurring words.

Lines 122 - 136. .
The two word lists `negative-words.txt` and `positive-words.txt` used in exercise one of CS918 were again used in this exercise. This section of code follows a very similar pattern to that employed in the first exercise; process the text files to represent each word as a key to a dictionary, where the values are either ‘‘`positiveword`’’ or ‘‘`negativeword`’’.

A further text file `twitter-lexicon.txt` was used [2]. This is a file containing around 60,000 Twitter-specific words and their associated sentiment score (-5: strong negative sentiment, +5: strong positive sentiment). These words include hashtags, user mentions, URLs, misspelled word etc. This makes it highly applicable to the classification of tweets since the lexicon domain is the same. Like the positive and negative word lists described above, the Twitter lexicon was converted to a dictionary where the keys are the words, and values the sentiment score. The two dictionaries defined in this part of the code are `pos_neg_dict` and `twitterLexiconDict`.

Lines 139 - 186. .
The `get_feature_space(tweetList, training = True)` function defined in this section of the code performs the text processing and generates the feature space for both training and testing

data. The optional argument `training` controls the `getUnigrams()` function to return either the training unigram array, or the transformed testing unigram array.

Firstly, a number of empty lists are defined, these are to represent the tweet IDs (`idVec`), the true labels of each tweet (`y`), the sentiment score calculated using the `twitterLexiconDict` (`lexicon_vector`), and the total token count for each tweet (`count_vector`). The sentiment score is calculated before any other text processing in order to take advantage of the Twitter-specific lexicon in `twitterLexiconDict`. Each tweets score is then appended to the list `tweetScore` to be further appended to `lexicon_vector` ready to be returned as an output of the `get_feature_space()` function.

The text processing is applied in stages. At each stage, a function is applied to the tweet and the variable `processed_tweet` is updated. The order of functions is important to be able to match the highest number of features. For example, features involving punctuation (like emojis and URL links) were replaced first, then purely alphanumeric features were replaced. Some features may be replaced with more than one function. For example, the string "HAHAHA" can be replaced with both `replace_laugh_text()` and `replace_elongated_words()`. Some concessions needed to be made in terms of which feature to attribute strings that could be more than one feature. The ordering of the functions reflects these concessions. Here is an example of a tweet that has been processed:

Before: @user123 Hey! Take a look at this video https://t.co/H8kvKn3ANW, soooo funny, right?!?! HAHA #amazing.

After: usermention hey take look video urllink elongatedword funny right questionexclaim laughword hashtag

Lines 189 - 197. .
The training data is then opened and saved as the variable `trainingTweets` as a list of lists, with each sublist containing the tweet ID, the label, and the tweet text itself. This is then fed into the `get_feature_space()` function to return 6 outputs. All outputs except `X_train` and `mapping_train` are column vectors. `mapping_train` is a dictionary which maps unigrams to the column number of `X_train`.

Lines 199 - 249. .
Using the feature space generated above, three random forest classifiers are used to fit the data. It was mentioned earlier in this report that classification will be attempted with as few features as possible. That means most of the unigrams in the `X_train` feature space will be eliminated. This is achieved by putting all relevant features in a list, and then looping through this list in conjunction with the dictionary `mapping_train` to select only the columns that correspond to those features. This results in the reduced feature space `reduced_X_train` which has 45101 rows, but now only 13 columns.

A further two columns are added via the `np.append()` method; `lexicon_score_train` and `token_count_train` (a count of all tokens in each tweets). A random forest classifier object is defined as rfc. This is then used to fit the training data to their corresponding labels in `y_train`. Three classifiers with differing parameters are looped through and fitted.

Lines 251 - 275. .
Once a classifier is fitted, it is then tested on three test sets. The same procedure as above is applied to the test data to generate a feature space and reduced feature space. The labels from

the test data is then predicted using the method `rfc.predict(reduced_X_test)`. Predictions are transform into a dictionary with their true labels and evaluated using the `evaluate()` function provided in the `evaluation.py` file.

3. Testing and Results

The same set of features were used in all three classifiers, with only the parameters changing. The original names of the classifiers were kept: `myclassifier1`, `myclassifier2`, and `myclassifier3`. `myclassifier1` applies a random forest classifier with default parameters. This is to serve as a baseline classifier from which improvements can be applied for `myclassifier2`, and `myclassifier3`. A random forest is an ensemble classifier made up of a number of decision trees built from subsets of the training data. Averaging results over all the decision trees increases classification accuracy and avoids overfitting. A random forest classifier tends to overfit when there are few samples with meany features. The opposite is true for this training set, where there are 45101 samples (tweets) and a reduced feature set of only 15. If all unigrams and bigrams were incorporated, the feature set would be substantially increased (unigrams alone would add $\sim$ 30000 features), increasing the risk of overfitting.

Four parameters were adjusted to improve accuracy. These were n_estimators (the number of decision trees over which the results were averaged), criterion (splitting on the gini index or on information gain), max_depth (the number of layers in each decision tree), and min_samples_leaf (minimum number of samples in each leaf). Evaluation was carried out by assessing the F1-score which is given by:

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{1}$$

where

$$\text{recall} = \frac{TP}{TP + FN} \qquad \text{and} \qquad \text{precision} = \frac{TP}{TP + FP} \tag{2}$$

myclassifier1. .

- n_estimators = 10
- criterion = “gini”
- max_depth = None
- min_samples_leaf = 1

For `myclassifier1`, all the parameters taken their default values. The F_1 scores over the three test sets were between 0.460 to 0.490. The classifier is able to distinguish positive tweets fairly robustly, with a true-positive rate of between 0.600 - 0.680. The exact figures quoted vary from test to test. A screenshot of one such test is shown in Fig. 2.

While positive tweets can be distinguished, particularly from negative tweets (as shown by the low false-positive rates of 0.077, 0.059, and 0.087 across the three test sets in Fig. 2), negative tweets were classified at around 0.333, or, no better than random. Neutral tweets have a slightly

```
Training myclassifier1
Training done!
twitter-test1.txt (myclassifier1): 0.488
            positive  negative  neutral
positive    0.606     0.077     0.317
negative    0.253     0.460     0.287
neutral     0.317     0.143     0.540

twitter-test2.txt (myclassifier1): 0.462
            positive  negative  neutral
positive    0.689     0.059     0.252
negative    0.352     0.324     0.324
neutral     0.440     0.100     0.460

twitter-test3.txt (myclassifier1): 0.469
            positive  negative  neutral
positive    0.625     0.087     0.288
negative    0.292     0.384     0.324
neutral     0.348     0.126     0.526
```

FIGURE 2. Example results of `myclassifier1` over the three test sets.

better true-positive rate at about 0.500. Improvements can definitely be made adjusting the parameters of the random forest classifier.

myclassifier2. .

- n_estimators = 1000
- criterion = "entropy"
- max_depth = None
- min_samples_leaf = 1

Two key changes were made for `myclassifier2`. The first was to increase the number of decision trees from which the average result is calculated. The second was to change the splitting criterion to "entropy" (information gain). The choice of gini index or information gain, during preliminary testing, appeared not to have much effect on the performance of the classifier. Neither did running time seem to be affected. So changing the criterion parameter was a personal preference for information gain. Changing the n_estimators parameter did however affect the performance. An example of one test run is shown in Fig. 3.

The true-positive rate for positive tweets is again the highest. Negative tweets are still difficult to distinguish from positive and neutral tweets, albeit at a slightly higher rate that `myclassifier1`. The neutral tweet true-positive rate is also improved. Overall performance is improved from `myclassifier1` as shown by the higher F_1 scores across all testing sets. Further improvements can be made by adjusting the final two parameters; max_depth and min_samples_leaf.

```
Training myclassifier2
Training done!
twitter-test1.txt (myclassifier2): 0.511
           positive  negative  neutral
positive    0.617     0.069     0.314
negative    0.233     0.518     0.249
neutral     0.288     0.155     0.557

twitter-test2.txt (myclassifier2): 0.513
           positive  negative  neutral
positive    0.685     0.062     0.253
negative    0.329     0.429     0.242
neutral     0.403     0.097     0.500

twitter-test3.txt (myclassifier2): 0.487
           positive  negative  neutral
positive    0.642     0.081     0.277
negative    0.274     0.416     0.309
neutral     0.326     0.135     0.540
```

FIGURE 3. Example results of `myclassifier2` over the three test sets.

myclassifier3. .

- n_estimators = 1000
- criterion = "entropy"
- max_depth = 13
- min_samples_leaf = 10

Limiting the depth of each decision tree has the effect of providing more samples in each leaf, from which a label can be predicted. Therefore, for `myclassifier3`, max_depth = 13 (based on preliminary testing this gave optimum performance). To ensure there are enough samples in each leaf node, min_samples_leaf was set to 10. An example of one test run is shown in Fig. 4.

As indicated by the higher F_1 score, limiting the tree depths has improved the performance across all test sets. The same pattern still exists; high true-positive rate for positive tweets, with the true-positive rates for negative and neutral tweets given a boost.

4. IMPROVEMENTS

Evaluating the performance of the three classifiers across all test sets, a number of conclusions can be made. The feature set, when combined with a random forest classifier of appropriate depth and number of decision tress, is able to classify positive tweets fairly adequately. Careful attention was given to limit the effect of overfitting by increasing estimator numbers and setting a minimum number of samples in each leaf. So, we can be more confident that the classifiers are

```
Training myclassifier3
Training done!
twitter-test1.txt (myclassifier3): 0.537
            positive  negative  neutral
positive    0.698     0.049     0.253
negative    0.177     0.668     0.155
neutral     0.271     0.158     0.570

twitter-test2.txt (myclassifier3): 0.556
            positive  negative  neutral
positive    0.744     0.045     0.211
negative    0.262     0.545     0.193
neutral     0.393     0.095     0.512

twitter-test3.txt (myclassifier3): 0.532
            positive  negative  neutral
positive    0.728     0.056     0.217
negative    0.225     0.554     0.221
neutral     0.318     0.129     0.553
```

FIGURE 4. Example results of `myclassifier3` over the three test sets.

not overfitting. The classifiers have trouble classifying neutral tweets, and to a greater extend negative tweets. Several improvements are suggested.

Unigrams, bigrams, and word embeddings. .
An obvious omission to this classifier are unigrams, bigrams, and word embeddings. Features such as these could have the potential to greater distinguish tweet labels. The principle governing the development of the classifiers for this report was to reduce the number of features, speeding up running time and lowering the sparsity of the feature space, so other more numerous features were disregarded. However, given carefully adjusted parameters to avoid overfitting, and using Python packages like `pickle` to save model objects, incorporation of unigrams, bigrams, and word embeddings could improve the overall accuracy.

Other classifiers. .
Only random forests were considered in this report, but it is of course not the only option. A multinomial naive bayes classifier would be appropriate for integer counts of features, or tf-idf fractions. Only positive values can be fed into the algorithm, so the scale of sentiment values (-5 to +5) used in this report would have to be adjusted.

Optimised regular expressions. .
Although the regular expressions used in these classifiers were able to gather many features from the raw tweets, they were not perfect. In particular, it would be interesting to notice the performance improvement (if any) if emojis were selected based on their emotional significance, not just a discrete count of present or not.

Appendix A. Workflow of Sentiment Classifier

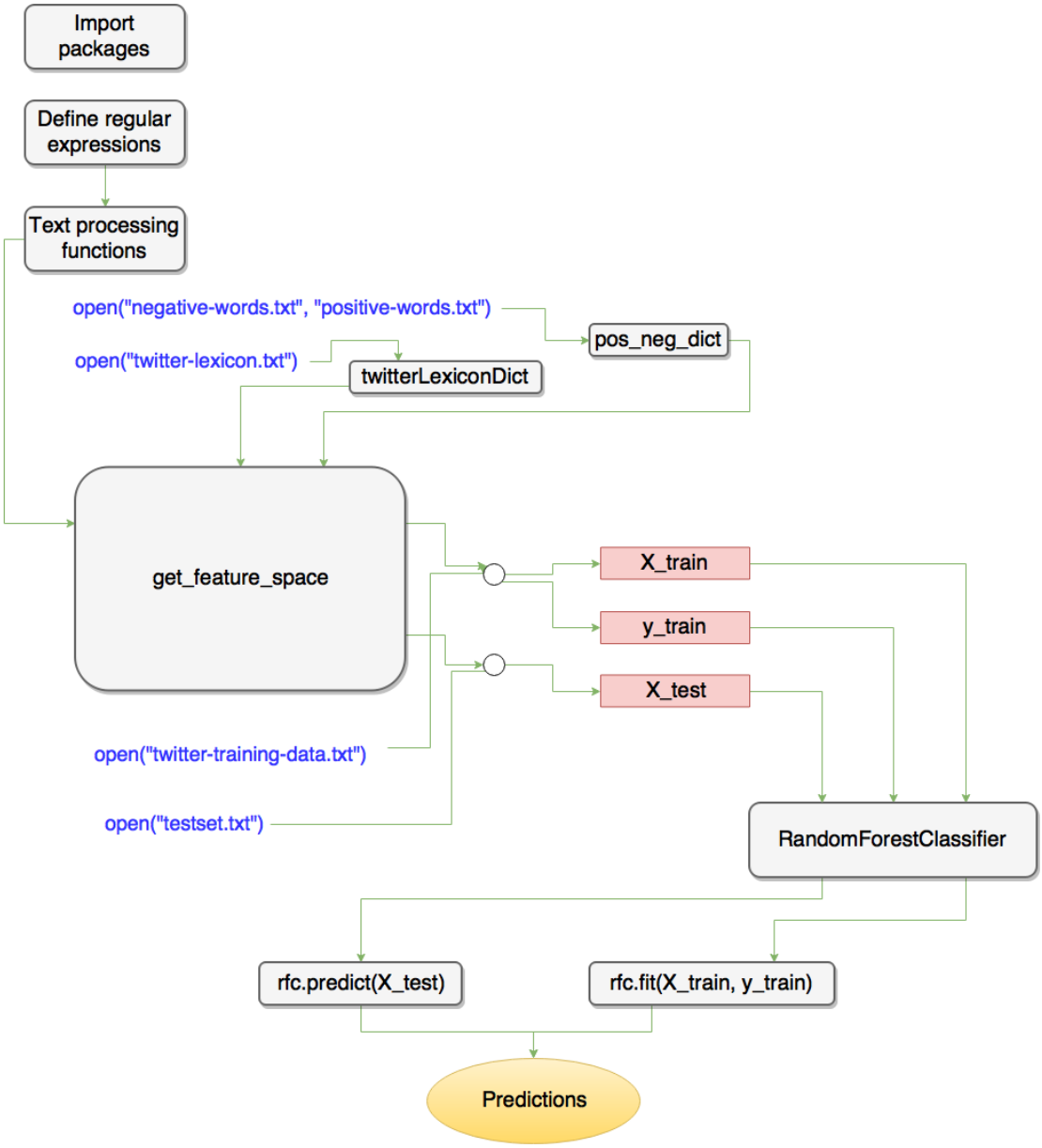

Figure 5. Flow chart representing the work flow of the three sentiment classifiers developed in this exercise.

References

[1] https://unicode.org/emoji/charts/full-emoji-list.html
[2] In Proceedings of the seventh international workshop on Semantic Evaluation Exercises (SemEval-2013), June 2013, Atlanta, Georgia, USA.

[3] Knuth: Computers and Typesetting,
http://www-cs-faculty.stanford.edu/~uno/abcde.html