Sentiment analysis on Twitter

A jupyter notebook called Ex2.ipynb has been included with comments and explanations of the code. For testing purposes, classification.py can be run which contains the same code as the jupyter notebook.

The data provided for this task is a corpus of tweets. As Twitter is an informal social media platform, the data has a lot of noise from elongated words to emoticons (emojis) to hash tags. Therefore, a lot of preprocessing was required to clean up the data so that it could be used for sentiment analysis.

Preprocessing

For the preprocessing part, I have used a lot of the regex code from the previous assignment. The first and foremost task was load up and preprocess the data. For preprocessing the following tasks have been performed:

1) Lower case all the data
2) Replace all URLs in the tweets with a word called "**urllink**". The regex used for this is **'\b(http)(s)?:\/\/((([\w\/])(\.)?)+)\b'.** With hindsight, the replacement word should have been **<url>** as the latter is included in the Glove word embedding.
3) Replace user mentions with the "@" sign with a word **usermention**. The regex used for this is **^(?!.*\bRT\b)(?:.+\s)?@\w+**. With hindsight, it would have been better to use **<user>** as a replacement word as the latter is contained in the Glove word embedding.
4) To remove short noisy words such as "a" and "I" which carry very little meaning in terms of sentiment, all one word words have been removed using regex, **\b[A-Za-z0-9]{1}\b**.
5) As lots of emojis are often used in social media, I have made some attempt to convert the happy and sad face emojis to **happyface** and **sadface** words. With hindsight, it would have been better to convert these directly to happy and sad words. The two regexes are given below:
   Happy: :\)|:]|:3|:>|8\)|\(:|=\)|=]|:\'\)
   Sad: :\(|:\[|:<|8\(|\(:|=\(|=\[|:\'\(|:-\(

Additionally, I have used the **WordNetLemmatizer** with default PoS tagging to lemmatize the words. This has helped to reduce the vocabulary. After the lemmatization has been performed using NLTK's **WordNetLemmatizer**.

Feature extraction

For feature extraction, I have used 3 primary features:

Lexicons:

Having downloaded a list of words with specific sentiment values. The values of each word in the list ranges from -1 representing extremely negative words to +1 representing extremely positive words. Most of the words have a sentiment value between these two values.

NGrams

For the ngrams, I have used sklearn's **CountVectorizer** and its transform method. Additionally, I have used the td-idf transformer for normalization.

Glove – Twitter word embeddings

The Glove word embeddings had several files available. Out of these, I have used the one with just 25 dimensions. The primary reason for this was to increase the vocabulary available for the word embeddings. The number of words included in the word embeddings was significantly less. As a result, most words would have a 0 valued vector resulting in less information per word.

Classifiers

Lexicon rule-based classifier

The lexicon-based classifier is the first and the simplest classifier in this project as only the features are used without any machine learning. To calculate the sentiment for the tweet, the individual sentiment values for each word is taken and summed up. If the value is between -0.15 and +0.15 then the value is neutral. For anything below, -0.15 it's negative and for anything above +0.15, it's positive.

I have arrived at these values by experimentation to achieve a higher f1 score.

NGrams – Binary unigrams with Naïve Bayes

For this classifier, firstly from sklearn's feature extractor CountVectorizer has been used on the corpus of tweets to calculate binary_unigrams matrix. During this process, I have also built the array of the targets (positive, negative, neutral).

For the training process, the count vector was taken and normalized using the **TfidfTransformer** from sklearn. Using **MultinomialNB** as the classifier, they were fitted to the model.

For the classification part, the tweet was taken and transformed into the count vector using the original binary unigram. It's then normalized similarly to the training process. After that, using the model's **predict** function, the prediction is made.

The performance for the naïve bayes classifier has been poor with the f1 scores between 0.26 to 0.33.

Glove – Twitter word embeddings

The SVM classifier takes the sum of vectors of the sentence as the features. The target labels are converted to integer values. Using a class **GloveTwitterWordEmbedding**, the word embedding file is read and parsed. A dictionary is created where the key is the word as a string and the value is a numpy array representing the vector. The class also contains a method **get_embeddings_for_sentence** for converting a string into a sum vector. This method is used for converting the tweet strings into word embedding vectors both during the training and the inference process.

For the training process, I have used the multi-class **OneVsRestClassifier** with the **SVC** as the estimator and gamma set to auto. As the training process takes quite a long time, I have saved the trained model as a file called **my_svm_model.pkl**. To train the model from scratch, just rename or delete this file. If the file is found in the system, it'll be used instead of training the model from scratch.

For the inference task, the model's **predict** function is used. The performance of this model is very poor with extremely low f1 scores. It can be improved significantly.

Future improvements

There are many improvements that I plan on making if I had more time. Firstly, the classifiers do not account for negation words such as “not” before another word (eg “not happy”). This is especially true for the lexicon classifier. As a result, its very likely to have incorrect results where negation I used. Therefore, during the preprocessing stage, **not**_ can be added as a prefix to words which comes directly after the word **not**. When extracting the features, the sentiment value of all words with the **not**_ prefix can be multiplied by **-1** to flip their sentiment value. For example, **not happy** will become **not not_happy**. When the sentiment value of **happy is 0.75**, the value of **not_happy is “-0.75”**.

Furthermore, the most basic classifier does not use machine learning. This can be changed by using Naïve Bayes to improve the performance. Currently the hard-coded values are used as boundaries which determine which value is neutral, negative and positive. By using machine learning, much more appropriate boundaries can be set. SVM in this case would probably be the best way to go.

There is also significant overfitting in the model as the performance on the training set is SIGNIFICANTLY better than the performance on the three testing sets. This can be improved further to enable better generalization.

Code reusability and standalone code

For most of the classifiers, preprocessors and feature extractors, I have created my own classes and focused on code readability, reusability and maintainability. This also applies for the corpus data loader and test data loader.

A jupyter notebook called Ex2.ipynb has been included with comments and explanations of the code. For testing purposes, classification.py can be run which contains the same code as the jupyter notebook.