Imperial College London

Natural Language Processing - CO 490H

NLP - Offensive Language Detection

Codalab Challenge

Adrian LÖWENSTEIN
Mihai LUNG
Pierre-Louis SAINT

Professor Lucia SPECIA

March 24, 2019

Contents

1 Introduction

In this report we will document our approach to addressing the CodaLab challenge on categorising offensive language in Twitter data. To this end we will attempt to answer the three tasks in the challenge, namely:

- A) Performing a binary classification on whether a tweet is offensive or not.
- B) Categorizing the offense type in a tweet, namely whether it is targeted to an individual or untargeted.
- C) Identifying the targets of offensive tweets, namely whether the target is individual, group or other.

More information on the challenge and data used can be found on CodaLab (1). Our code can be found on our GitLab Repository.

2 Word Embedding

In order to have a numerical representation of the words in the tweets, a word embedding was used for this project. An embedding is a vector representation of each word which is trained automatically to uniquely represent the meaning of the word in numerical format. We have chosen to use the Global Vectors embedding, as there exists a version which was trained on Twitter data, which we considered to be the most appropriate, as it will likely have been trained on similar data to the one we will be analysing, and should therefore contain relevant embeddings. (**GloVe**) (2).

The GloVe embedding which was pre-trained on a twitter dataset is available in several dimensions. We use the 200 dimensions vector for this project (the largest one). We experimented with smaller dimensions however in our experiments we found that higher dimensional embeddings out-performed smaller ones. We did not find this surprising - in Machine Learning applications more relevant inputs are almost always preferable assuming there is sufficient processing power to analyse them. Especially since we have a quite limiting number of observation for this application, using rich embeddings represents a useful complement to our data.

3 Data Processing

The data-set provided contains raw twitter data. By its nature this data contains a lot of hard-to-interpret features, so it would have been unwise to apply the GloVE embedding directly to the unprocessed tweets. Some of these features include: informal language, grammatical mistakes, emojis and special characters. We therefore undertook to perform extensive pre-processing work on the raw data before applying our models on it. For this purpose we use several tools, including Regular Expressions (regex) and some additional useful packages.

3.1 Data Pipeline

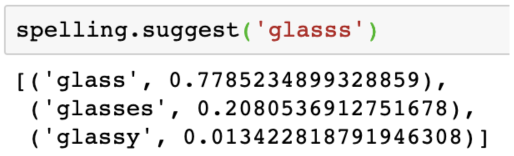
Below we describe in detail how the data was processed before the embedding is applied.

Twitter Specific : As we are dealing with natural language data, we were able to use regex to apply powerful transformations that removes every #, transforms every @USER into the tag <user>. Also as many hashtags contains words of the form "*ILoveNLP*", we apply some regular expression (regex) formulas in order to split those in "*I Love NLP*".

Emojis : Emojis are widely present across the data-set. Most of the time they represent a feeling or contextually-relevant information. We choose to ignore most of them as they are not present in the embedding, and generic transcriptions available online did not prove to be particularly useful at capturing the real meaning, based on a qualitative assessment of a small sample of tweets. We therefore settled on selecting a few emojis which occur frequently and that add important information to the classification. We have manually created a dictionary of the emojis we considered to be most important based on a qualitative analysis of the data, and replaced the emojis identified as such with their associated transcriptions.

Special Characters : We remove from the tweets most of the special characters, except: dots, commas, question marks, and exclamation marks. These are frequent punctuation characters, which are also included in the embedding.

Misspelling suggestions : For every word which is not recognised by the embedding, we apply a suggestion algorithm that finds the closest words and their probability. We use the function `suggest()` in the package `pattern.en` ((3)). The function returns a normalised probability distribution of how likely a misspelled word is to be one of several suggestions, based on simple frequency counts.



```
spelling.suggest('glasss')
```

```
[('glass', 0.7785234899328859),
 ('glasses', 0.2080536912751678),
 ('glassy', 0.013422818791946308)]
```

Figure 1: Misspelling suggestions - example

We choose to keep the suggestion if the probability returned by the package is over $p = 0.5$ and to drop the word from the data-set if it is below this value. While the probabilities only reflect relative likelihoods rather than absolute certainty of the correction. A probability of over 0.5 means either that there the word is significantly more common than other options, or that there are not many options to begin with so there are fewer wrong choices to pick from. This is a restrictive approach, however we consider that for this application getting a lot of corrections but many of them wrong is a bigger problem than getting a high accuracy rate on our corrections but dropping many more words.

After applying this to the dataset we end up with a rate of rejected words of 6.15% and a rate of corrected words of 4.76%.

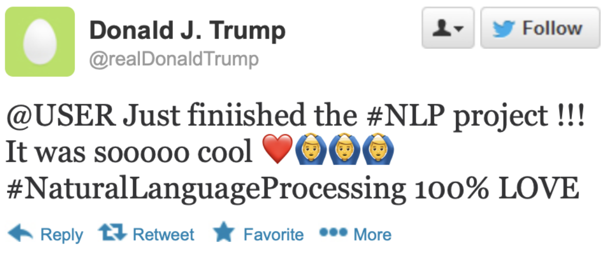
Repeated letters : Sometimes common words are spelled with one or several repeated letters. For example the word " *Yes*" could be written as " *Yeeeeees*", resulting in a word our embedding would not recognise. Given that double letters occur frequently in English, simply shortening every sequence of multiple letters to just one is not a fool-proof approach. Our approach therefore was to first apply a regex formula that reduces the number of repeated letters to two, then applying the above-mentioned misspelling correction method to try and find the correct spelling.

Tokensiation : Once the cleaning was done, the text of each tweet is split at blank spaces. We added blank spaces between punctuation and other words where necessary to ensure the tokenization separated them.

Repetition words : When we look at the sequence of tokens, we remove finally the repetitions of tokens.

3.2 Processing Examples

Here we show how a tweet is processed into clean tokens. The example provided on figure 2, shows a tweet that has all the problems mentioned previously.


Donald J. Trump
@realDonaldTrump

@USER Just finiished the #NLP project !!!
It was sooooo cool ❤️🙆🙆🙆
#NaturalLanguageProcessing 100% LOVE

Figure 2: Example of Tweet.

After applying our processing, the output would be : `<user> just finished the nlp project ! it was so cool love natural language processing love`.

4 Classification

4.1 Architecture

In order to complete the tasks we have chosen to implement three main deep learning archetypes, namely: fully connected linear models, recurrent neural networks, and convolutional neural networks. All of these have been implemented using PyTorch. The models used for our final predictions and their hyper-parameters were chosen using experimentation and by consulting literature and precedents online.

All of our architectures ended with a softmax activation, which was fed to a negative log likelihood function to obtain the final loss which was used to tune the model. We have chosen this approach due to the increased numerical stability provided, as supported by the mathematics behind it and empirically confirmed throughout the literature.

4.1.1 Linear Model

For our linear model, we make predictions by taking the mean embedding of the words in each tweet and feeding this to a fully-connected network.

This approach, as expected, ended up netting out one of the weakest performances. By simply averaging the word embedding vectors, we do not only lose the sequential information implicit in natural language, but also end up with an even less informative set of features than an un-ordered list of words, as the averaged embedding vectors are likely but not guaranteed to uniquely characterise the tweet.

In our experiments, Linear Models achieved a peak of 64.0% accuracy in Task A, 57.5% in Task B and 51.7% in Task C. This is highest for Task A, as expected - the averaging does a good job at still capturing the high level features which sometimes suffice to capture the important features of a tweet.

For Tasks B and C, where we are trying to detect more subtle nuance in the tweets, Linear Models perform poorly. We interpret this as a confirmation of our intuition that detecting more granular information about tweets such as who they target is a more sophisticated task where the sequential nature of human language cannot be overlooked without losing significant inferential power.

4.1.2 Recurrent Neural Networks

For the recurrent neural networks (RNNs), we have used PyTorch modules to implement a simple linear recurrent neural network (simple RNN), a long-short term memory (LSTM) recurrent neural network, and a gated recurrent unit (GRU) neural networks.

Our implementation of these networks works by sequentially feeding each word embedding and the hidden layer generated by the sequence preceding it to the RNN, which then generates an updated hidden layer which is fed together with the next word embedding to the same RNN, and so on until the end of the sentence sequence. The hidden layer is initiated as a zero vector for the first word embedding. Our principal output of the RNN is a tensor containing all of the hidden layers generated through this sequence.

As previously mentioned, our data loader pads all sentences with a zero-embedding padding until all the sentences in a batch are the same length as the longest one in said batch. We initially experimented with using the final output of the RNNs (eg after passing all the paddings through the RNN as well) and found that our model performed poorly and was unstable. An RNN is supposed to give a comparable output regardless of the length of the input sequence. Further modifying this output, dozens of times for some sentences, by passing padding inputs through the network ended up distorting this crucial comparability property to such an extent so as to render the model useless.

As such we instead used as our output from the RNN the hidden layer obtained after passing the final word in the tweet through the network. This was then fed to a fully-connected network which converts the input into an output of the desired shape. We used ReLU intermediary activations for specifications where additional linear layers were used to convert the output of the linear layer into a final output for the model.

All of these have been parametrised in order to maximise flexibility, which allowed us to experiment with varying sizes and numbers of layers of the RNNs, and with varying architectures for the output layer converting the final hidden layer into the desired output shape.

In all tasks, GRU performes the best, LSTM was almost as good or up to 1 percentage point behind, and the simple RNN about 1-2 percentage points behind GRU. The best results obtained (all using GRU) are: 75.3% on Task A, 59.6% on Task B, and 53.9% on Task C.

While we normally would have expected LSTMs to overall perform better, we are working on a relatively small dataset for this application. As such GRU's more conservative usage of parameters while still largely maintaining the adaptive memory property of LSTMs possibly gave it an edge over LSTMs, managing to generalise the data better by using fewer parameters.

4.1.3 Convolutional Neural Networks

We implemented two convolutional neural networks of different architectures. We used a static word embedding for both.

The first architecture was borrowed from one of the tutorial notebooks of this course. It consists in one convolutional layer which is applied to a window of h words, to produce N feature maps. We then apply a max-pooling function over these feature maps. This operation will remove one dimension from our features, which is the length of the tweets. The idea is to keep the most important feature and remove the others. Finally, we pass our features to a fully connected softmax layer. The output is the probability distribution over our labels. The key parameters of this architecture are the window size h and the number of feature maps N. After experimenting with a number of architectures we found that we obtain the best performance using $h = 3$ and $N = 100$. Shallower networks seemed to lack the explanatory power required to explain the data well.

For the second architecture, we tried to extend the previous one. We thought it was too simple because it only produced features from a fixed number of consecutive words in a tweet ($h = 3$). After reading (4), we implemented the architecture proposed by the author. This architecture is really similar to the previous one: it consists of three convolutional layers with different window sizes, each being applied in parallel on the input batch of tweets. A max-pooling function is then applied independently on the three sets of feature maps. We then concatenate the three sets of features. Finally, as in the previous architecture, we pass our features to a fully connected softmax layer. The output is again the probability distribution over our labels. For the window sizes, we fixed $h_1 = 3, h_2 = 4$ and $h_3 = 5$. We kept $N = 100$ for the number of feature maps for each convolutional layer. The activation functions used from the convolutional layers are ReLU for both architectures.

The second architecture is naturally more complex, because it is able to learn from relations between 3, 4 and 5 consecutive words. This complexity translated in better results for the second architecture compared to those of the first one.

We compared the performance of these two networks on a validation set which consisted in 20% of the shuffled provided dataset. The 3-layers CNN performed slightly better than the 1-layer CNN in all tasks. More specifically, The 3-CNN achieves the following Macro F1-scores on the validation set: 0.76 on Task A, 0.65 on Task B, and 0.55 on Task C. The 1-CNN achieves the following Macro F1-scores on the same validation set: 0.745 on Task A, 0.63 on Task B, and 0.53 for Task C.

4.2 Training Procedure

Validation Set : For the purpose of validation we create a dataset that contains a fixed percentage of each classes of the training set. We do not overfit the validation dataset and it is therefore unbalanced.

Balance in classes : The original Training set is unbalanced and it is a problem for the classification. We cope with this by over-fitting the smallest classes of the dataset. By randomly repeating tweets of those class in order to increase their number until we reach the size of the majority class.

Hyper-parameters Search : The accuracy of the classification relies heavily on the values of the parameters. For each of the previously described model we have some specific parameters such as the number of layers or their depth. All of them have in common parameters such as "batch size", "learning rate", "weight decay", etc ... In order to tune all those parameters we use a *grid search* and look at the **macro f1 score** as the objective to maximize.

4.3 Testing Procedure

After the training procedure we begin the testing. For that we apply the same Data Processing to the Test Data of each subtask before testing it on each of the classifier.

5 Test Results

They were submitted through the name : `adrianlwn`. The results can be found at : link

Here you can find the results for every classifier in every subtask

Score	Subtask	FFNN	CNN 1	CNN 3	RNN	LSTM	GRU
Macro F1	A	0.6722	0.7397	0.7488	0.6982	0.6982	0.7310
	B	0.6069	0.7157	0.6901	0.5672	0.6153	0.5714
	C	0.4998	0.5600	0.5808	0.5060	0.5777	0.5753
Accuracy	A	0.7070	0.7826	0.8070	0.7535	0.7535	0.7733
	B	0.7708	0.8583	0.8625	0.8500	0.8083	0.7667
	C	0.5399	0.6479	0.6948	0.5775	0.6479	0.6573

We clearly see that the architecture that performs the best on the Macro F1 score is the Convolutional Neural Network. In particular CNN 3 for subtasks A and C and CNN 1 for subtask B.

We plot also the Confusion Matrices for the best results.

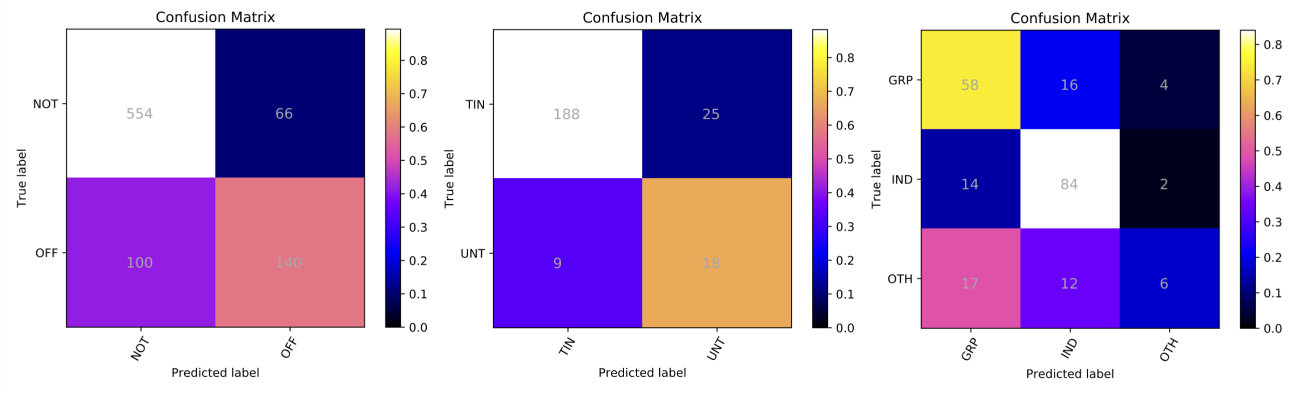


Figure 3: Confusion Matrix for each Subtask

As we can see the test dataset is very unbalanced, this is why the use of the f1 score is better to measure the effect of our classification.

Those results have allowed us to rank 21st on subtask A, 1st on subtask B and 1st on subtask C, among Imperial College Students. Worldwide we were able to rank 62nd, 5th and 11th on each subtask.

References

[1] Codalab, “Offenseval 2019.” https://competitions.codalab.org/competitions/20011, 2019.

[2] J. Pennington, R. Socher, and C. D. Manning, “Glove: Global vectors for word representation,” in *Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1532–1543, 2014.

[3] CLIPS, “pattern.en package.” https://www.clips.uantwerpen.be/pages/pattern-en, May 2013.

[4] Y. Kim, “Convolutional neural networks for sentence classification,” *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014.