Imperial College London

IMPERIAL COLLEGE LONDON
DEPARTMENT OF COMPUTING

NATURAL LANGUAGE PROCESSING

Offensive language detection challenge

$1^{st} of March, 2019$

Made by:
Vincent Jarasse

Challenge provided on the platform

CodaLab

Code available at :
https://github.com/VJarasse/NLP_offensive_language_detection

Contents

1 Introduction to the challenge

This report presents the methods and results that have been used to address the OffensEval (Offensive Language Detection) challenge provided on the platform codalab. The full details of the challenge can be found here (https://competitions.codalab.org/competitions/20011learn_the_details); however, we will present the main goals here.

1.1 Motivation

Offensive language is pervasive in social media. Individuals frequently take advantage of the perceived anonymity of computer-mediated communication, using this to engage in behavior that many of them would not consider in real life. Online communities, social media platforms, and technology companies have been investing heavily in ways to cope with offensive language to prevent abusive behavior in social media.

1.2 Sub tasks

In the OffensEval, the challenge is broken down into three sub-tasks taking the type and target of offenses into account.

1.2.1 Sub-task A - Offensive language identification

The goal here is to classify tweets as being offensive or not offensive. All types of offenses are considered in this sub task.

1.2.2 Sub-task B - Automatic categorization of offense types

In this sub task, one has to automatically classify an offensive tweet as being targeted towards an individual or not.

1.2.3 Sub-task C - Offense target identification

This final sub-task is different from the two previous ones because the labels are not binary. One must classify a targeted offensive tweet as being targeted towards a single individual, towards a group of people or towards another type of group.

1.3 Dataset

In order to perform the tasks described above, we are provided with a dataset of 13240 tweets written in English and pre-processed as follows: all the usernames have been replaced by "@user", and the urls by "url". The labels for each tasks are also provided, please refer to the instructions on codalab for more explanations.

2 Solution description

2.1 Data preprocessing

As one can easily imagine, tweets are far from being a simple corpus of words to analyse. In any NLP work, data preprocessing is an fundamental step to get significant results. When dealing with tweets, this step is even more important. Indeed, after getting rid of the capital letters, comas and points, still about a third of the tokens are not recognised according to common dictionaries.
This is explained by a conjunction of facts: the first one is the use of specialized tweeter jargon: for example when people invent new terms that become hastags. Moreover, the use of emojis represent a considerable amount of tokens, and even if some of them can be interpreted by a word or a description, the fact that they are often sticked together without spaces in between make the job harder. Finally, many spell mistakes and uncommon characters make it hard to recognised known words.

For this challenge, we have implemented a small function that takes all characters to be considered as delimiters (not wanted), such as "," or "%" or "" and have built a dictionary base on the lowered case output. Moreover, shortenings such as "you're" or "I'm" are replaced by the full forms "you are" or "I am". This is important for task B and C where the network has to find patterns relating a subject to an offense. In the end, approximately 14% of all cleaned tokens of the corpus are still unknown, but when counting the total frequency of these tokens in the corpus, they only represent 3.35%, which is a reasonable amount to deal with.

2.2 Data management

For this challenge, we are provided with a dataset of approximately 13k tweets. This dataset is unbalanced: slightly for task A with 66%-34%, but more importantly for tasks B and C. In order for the networks to learn evenly the features describing one class or the other, we write a pytorch data loading function that takes care of balancing the datasets. Moreover, we extract thanks to the Pandas library the subsets for task B and C respectively, because tweets that have been classified as NULL in the previous task(s) would otherwise pollute the learning.

On a practical point of view, an offline work is done to create .txt files containing the cleaned tokenized corpus: when loading a dataset, an instance of the class "Corpus" is created, and this leads to the tokenization (space splitting) of the corpus as well as the cleaning of the tokens, that are then stored as class members. From this instance, we can write a .txt file for each sub-dataset containing on each line the appropriate clean tweets.
We operate the same kind of work on the labels to transform them from string representations like "NOT" or "TIN" to an appropriate corresponding integer value (class value actually) that will be useful when dealing with the loss function and the accuracy prediction of our networks.

2.3 Word embedding

Word embedding performs the translation of the word from a string human readable form to a floating point vector representation readable by a machine. This step is as important as unavoidable. Two choices can be made for this challenge: either constructing a custom word embedding representation

from the data that we have, or rely on a pre-trained word embedding. Considering the limited size of the dataset (only 13k samples) and the highly specific nature of tweets tokens, we decided to go for the famous GloVe embedding trained on several millions of tweets. The 100 features one has been used for this challenge.

The GloVe dictionary is heavy (about 1GB in the 100 features form), so we operate an offline work to build a custom embedding dictionary specific to our corpus: we extract only the tokens that appear in our corpus from the GloVe file, and store them in the same way (emb_dic.txt file).

2.4 Neural network architectures

Now that we have all the ingredients to feed a neural network, we can think about which architecture to adopt.

2.4.1 First model : FCNN

The first model that we have designed is very simple three-hidden layers fully-connected neural network. Its architecture is described in the figure below.

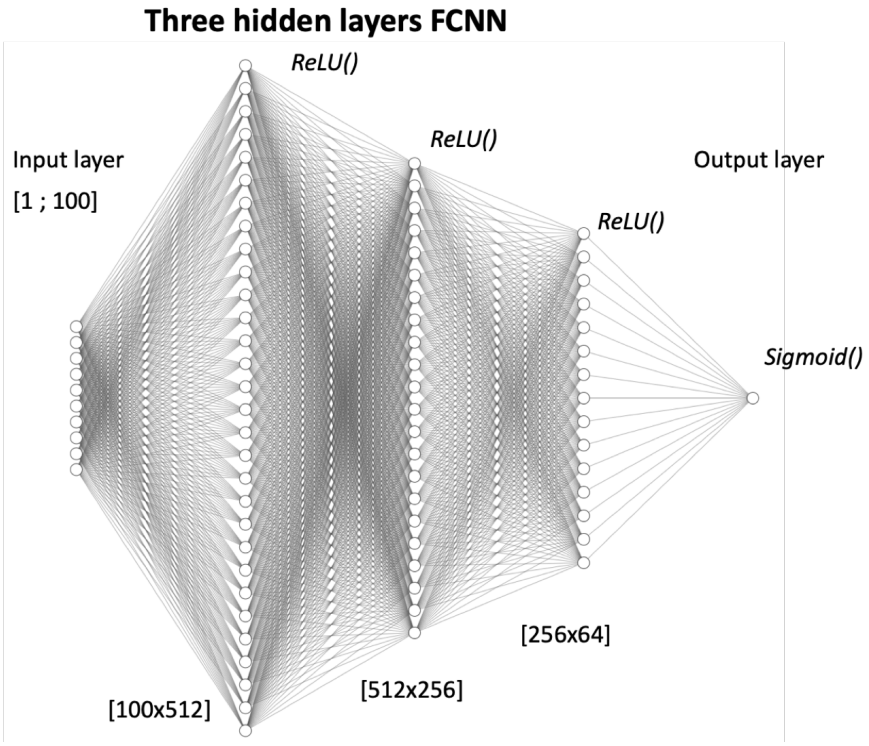

Figure 1: First model tested

FCNN, even with a simple architecture, are often good classifiers. In the context of the task A, i.e classify whether a tweet is offensive or not, one could think that detecting offensive words is a pretty

simple task, as long as the embedding is of good quality (which is the case here). Because the links between the words does not seem to have a role to play here, we even used the average of all the tokens embeddings of a sentence to represent the sentence itself. The input of this NN is then a vector of size 100 (the embedding size). Because the objective of this network is to perform a binary classification, we only use one output neuron with a sigmoid activation function. The appropriate loss function to use in pair with this acitvation function is the BCE (binary cross entropy) loss. The prediction of the model for a given input is easily obtained by using the python round() function.
This simple network achieved on a test dataset (separated from the training dataset) an accuracy of about 73% with a macro-average F1 measure of 0.7.

2.4.2 Second model : CNN

The first model performs quite well but can clearly be improved. The second model that we have tried is a simple convolutional neural network. Its architecture is described in the figure below.

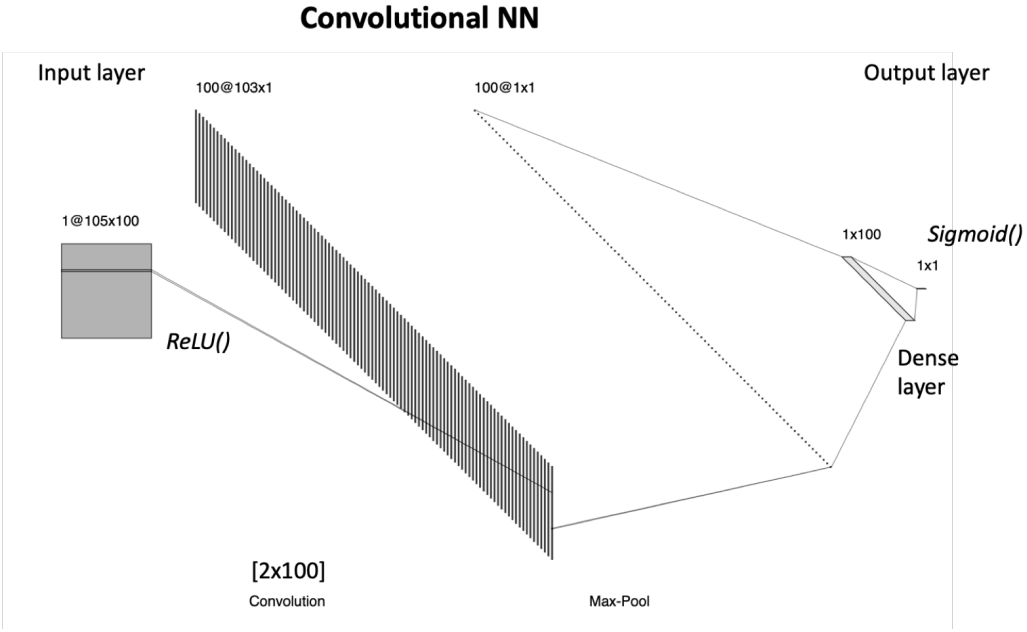

Figure 2: Second model tested

The first big difference compared with the previous architecture is the shape of the input. Here, we do not average the tokens' embeddings anymore, but store all the sentence's tokens' representation in a matrix. The first dimension of the matrix corresponds to the length of the longest tweet of our corpus (105 tokens). The shorter sentences are padded with zeros. The kernel size that was used is [2x100], in order to overlap between two consecutive tokens. However, a few trials have shown that for task A, the window size (first dimension of the kernel) does not really impact the results. This may be because no relation between the tokens is necessary to classify a tweet as offensive.

This CNN achieved on a test dataset (separated from the training dataset) an accuracy of about

76% with a macro-average F1 measure of 0.73.

2.4.3 Third and fourth models: RNNs

Motivated by tasks B and C which require to recognize a pattern linking multiple tokens together, we decided to give recurrent neural network a try. Indeed their architecture is well-fitted for sequential data. Moreover, tweets are by definition limited to 280 characters so we can't have a situation where two related tokens are very far away from one another.

Among RNNs, long-short term memory is a good choice because its architecture deals with vanishing and exploding gradients problems posed by the nature of RNNs. When using the Pytorch library, the implementation of a LSTM network is very quite easy thanks to the nn.LSTM() class. We implemented this LSTM which is fed by the same input as the CNN described in the previous section, and contains 20 hidden features. This architecture outputs a 105x40 matrix.

In our third model, this LSTM output was then reshaped and went trough a dense layer to produce the output. A fourth and final model uses a convolution layer to squeeze the LSTM output towards the output. The final architecture is described in the figure below.

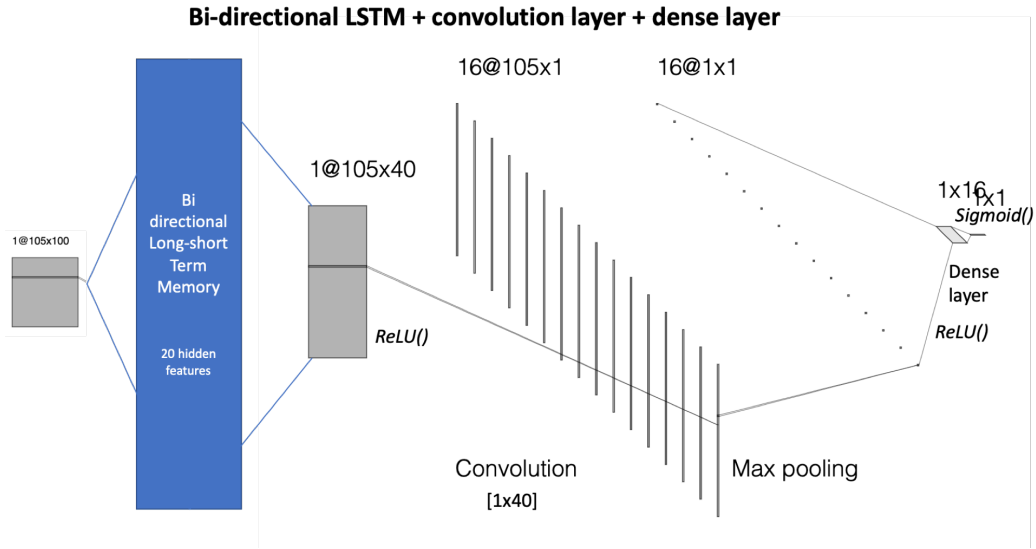

Figure 3: Fourth and last model tested and adopted

This architecture has to be slightly refined for task C, where the classification is not binary anymore. Indeed we have to classify a target offensive tweet to be targeted towards an individual, a group of individual or something else, which makes three classes. The output layer is in that case composed of 3 neurons instead of 1. Accordingly, we have to change from sigmoid() to softmax() output function. However, the softmax is not applied in the code of the NN because we use the pytorch CrossEntropyLoss() function which, according to the documentation, "combines nn.LogSoftmax() and nn.NLLLoss() in one single class.". One has to make sure that when prediction an output, the softmax() function is

well applied, and the argmax() in the good dimension is also used to get the final class label prediction.

Surprisingly, this latest model is quite quick to train on Google Collabs' GPUs (1 epoch of appr. 10k samples runs in about 2 seconds). It overfits the data after about 10 to 15 epochs, and all attempts to implement weight decay or another regularization technique has failed. To keep the best generalisation capacity of our model, we adopted the early-stopping method, thus stopping the training after 15 epochs for this architecture.

This neural network achieved on a test dataset (separated from the training dataset) an accuracy of about 78% with a macro-average F1 measure of 0.75, which is the best so far. We have used this architecture on tasks B and C for the aforementioned reasons, but also for task A where, more surprisingly, it performs quite well too! A final note on the batch sizes used: for task A, the number of sample is still quite high, so the batch size used was 32. However the number of training samples for tasks B and C is much smaller, we adopted a batch size of 16.

2.5 Submission scores

The platform evaluates the submission we have provided on all 3 tasks, and gives the macro-averaged F1-score as the performance/ranking criterion. The results of the above architecture are detailed in the table below.

$SubTask$	$F1$	$Ranking$
$Task\_A$	0.759	13/60
$Task\_B$	0.609	21/54
$Task\_C$	0.557	3/52

First, the macro F1 scores are not very high: this confirms that being able to automatically identify an offensive tweet is a difficult task. No doubt that the specific vocabulary involved and all the data preprocessing that is necessary explains a part of these values. Secondly, whereas identifying an offensive tweet (task A) is doable with a certain confidence rate, the patterns linking an offense to an individual or a group of individual is much more complicated, and the F1 scores are pretty clear.

The ranking values show that using the combination of a good preprocessing, GloVe embedding and recurrent neural networks achieved good results. These could have been improved by reducing he number of unknown tokens (implementing automatic mispelling detection and correction, emojis translation), and more fine tuning of the architecture. Indeed, considering the time constraints imposed by this challenge, only a few parameters have been tried.

3 Code

All the code used for this challenge is freely available on my Github at the following link : https://github.com/VJarasse/NLP_offensive_language_detection . This includes different python files (.py) that have been used to work locally and to organise the code properly, as well as a jupyter notebook file (.ipynb) intended to be executed on a Google collab environment. This latest file provides

a good overview of the code but does not aim at being complete in terms of data management. Moreover, this jupyter notebook file has been heavily used for testing purposes and has evolved a lot, so a few typos make be present.