EPFL ML Text Classification 2019

Maxence Jouve, Rayane Laraki, Pierre Schutz
EPFL

***Abstract*—Text classification is a challenging problem that has many applications such as spam filtering and language identification. Researchers started to address the problem in the 1980s but with the recent explosion of available data and advances in Machine Learning, the field gained a lot of traction. This report details our work on the "EPFL ML Text Classification Challenge" where the goal is to use machine learning to classify tweets. We have to predict if the tweet contained a ":)" or a ":(" smiley. Thus we need to classify a tweet as positive or negative.**
Our best model achieved 0.870 accuracy on the provided test dataset using BERT.

I. Introduction

Hosted on the competitive platform AIcrowd, this project aims to introduce us to text classification tasks. Competitors are evaluated using the accuracy and the F1-score of their predictions.

The training dataset contains 2,458,297 tweets. 1,218,655 of them are labelled as positive (they contained a ":)" smiley) and 1,239,642 are labelled as negative (they contained a ":(" smiley).

In this report, we will first explore the dataset. We will then try several approaches: computing tweet embedding from words embedding and training a classifier model on the obtained vectors, using fastText model for text classification, and then BERT Transformer model. Finally we will discuss our work.

II. Dataset exploration

We realized that tweet were anonymized. Indeed it is common to mention someone using "@" or to include a link to a resource outside of Twitter. Thus the tweets provided sometimes contained <user> and <url> tags. We decided to remove those tags as they are only used to make the tweet anonymous.

Thus when we talk about the basic dataset we assume tweets with <user> and <url> tags removed.

III. Baseline Approach

The first approach we tried was separated in two parts: we first needed to obtain tweet embeddings (vectors), we then trained classification models using them.

A. Using our trained Word2Vec model

1) Word2Vec model:

We decided to use the word2vec model developed by Google. We took the implementation using CBOW (Continuous Bag of Words). Given a corpus as input, this model splits the text in context windows. Words appearing in the same windows are said to share the same context. Given a context containing n words we use combinations of $n-1$ words as input and try to predict the most likely nth word for this context. Under the hood, one matrix has word vectors as row and another contains word vector as column and is used for computing the most likely word for a context. The $cosine\ distance$ metric is used to find the closest word. Finally, during training, the $softmax$ loss function is used and word vectors are updated using back-propagation [1] [2].

2) Training the Word2Vec model:

We trained two different models on the whole training data provided: one using our basic training dataset called $word2vec.model$ and one with the same dataset after word stemming (see explanation below) called $word2vec\_stemming.model$. Here are the parameters we used:

- min_count = 5 words that appear less than in the whole corpus this number are not considered (users sometimes use their own slang which justify this parameter)
- $size$ = 250 size of the word embedding vector (recommended value is between 200 and 300)
- $window$ = 5 size of the context window used for training
- $iter$ = 30 number of training iteration

The training of these models is done in the file $word2vec\_training.py$.

Finally $word2vec.model$ contains 94,930 words whereas $word2vec\_stemming.model$ contains 77,889 words.

3) Preprocessing and tweet embeddings construction:

We considered different way to construct our tweet embeddings:

- Basic tweets
- Basic tweets without english stop words
- Basic tweets with word stemming

Stop-words are common small word that might not add significance to a sentence. Examples are "I", "am", "the"...

We tried to remove them to see if they effectively do not add meaning to a tweet.

Stemming is the process of reducing a word to its most basic root. For instance, words "visiting" and "visited" that will both end up in the word "visit" after stemming. Thus the resulting word embeddings dictionary will have less words than the one obtained without stemming as observed in the previous section. However remaining "root" words are supposed to have a better embedding vector due to more context. We use the $Porter\ stemmer$ from the $nltk$ library to do it.

Finally we construct the tweets embedding by averaging the embedding of all words in the tweet if they are present in the word dictionary. If we decide to use the stemming preprocessing we will use $word2vec\_stemming.model$ otherwise we will use $word2vec.model$.

4) Training classification models using tweets embeddings:

We can now construct tweet embeddings using our basic dataset, with or without stop-words removal and with or without word stemming.

Once we have the tweet embedding we will try different classifiers both with and without regularization:

- Logistic regression
- Ridge regression
- Support vector machine with Hinge loss

The experiments can be seen in the $models\_testing-trained\_Word2Vec.ipynb$ notebook.

We used 80% of the dataset for training and 20% for testing. Moreover, regularization parameter was tuned using cross validation with 5 fold.

5) Results:

1. Basic tweets

Model	Test Accuracy
Logistic Regression	0.7796
Ridge regression	0.7754
Support Vector Machine	0.7726

The best prediction was obtained with Logistic Regression with C = 21.5 (C is the inverse of the regularization strength).

2. Basic Tweets without stop-words

Model	Test Accuracy
Logistic Regression	0.7670
Ridge regression	0.7659
Support Vector Machine	0.7659

Removing stop-words does not increase performance. We thus did not consider this preprocessing option for the last experience.

3. Basic tweets with word stemming

Model	Test Accuracy
Logistic Regression	0.7777
Ridge regression	0.7719
Support Vector Machine	0.7748

We can see that word stemming did not increase performance either.

B. Using a pretrained GloVe model

Because our available training dataset is relatively small, we decided to use a pretrained model for obtaining word embedding. We found a model called $glove-twitter-200$ that was trained on more than 2 billions tweets. The dictionary contains 1,193,514 words with embeddings of size 200.

1) Preprocessing and tweet embeddings construction: We used the same preprocessing option as in $A.3$.

2) Training classification models using tweet embeddings: We performed the same experiments as in $A.4$. Those can be seen in the $models\_testing-pretrained\_GloVe.ipynb$ notebook.

3) Results:

Only the best model will be shown because this approach did not improve performance.

1. Basic tweets

Model	Test Accuracy
Logistic Regression	0.7750

2. Basic Tweets without stop-words

Model	Test Accuracy
Logistic Regression	0.76290

3. Basic tweets with word stemming

Model	Test Accuracy
Logistic Regression	0.7551

As we can observe, stemming words decreased the performance a lot. This might be due to the fact that the pretrained model did not contain some words once they were stemmed.

Finally taking a pretrained model did not increase our performance. Indeed we might argue that the tweets used for training did not necessarily have a positive or negative meaning compared to our training dataset.

C. Limitation

The issue with our baseline approach is that the text classification task is split in two parts. In order to obtain better result we should train a model that takes as input raw

tweet and directly outputs the label (positive or negative). Indeed, with such a pipeline the model could back-propagate the classification error all the way through the model. We will now look at such approaches.

IV. Approach using fastText

fastText is a library developed by Facebook AI Research that can perform text classification.

A. fastText classification model

The particularities of fastText is that words are split into n-grams. Then each word embedding is obtained by averaging the n grams embeddings. As a result, fastText can learn word representation that have not been seen during training.

Word embeddings are obtained using a continuous skip-gram model. Word embeddings are scored differently than Word2Vec model. Skip-gram takes a word from a context of size n and try to predict the most likely n - 1 other words from this context [3].

Finally the classification layer uses text embedding computed as the average of word embeddings as input and the Softmax loss function to compute the most likely label [4].

B. Training pipeline

The fastText model can be trained with multiple preprocessing setups as well as multiple hyper-parameters.

1) Finding the best preprocessing:

In order to find the best preprocessing we created a train set containing 80% of the data and 20% for the test set. We applied the specific preprocessing for each set and obtained the accuracy.

We then used the default fastText hyper-parameters to test each configuration.

Here are the results:

Preprocessing	**Test Accuracy**
Without preprocessing	0.8352
Removing punctuation and neutral stop words	0.8106
Adding word stemming	0.8159
Removing punctuation/special characters	0.8296
Replacing smileys by corresponding word	0.8348

We did not consider combining multiple preprocessings as we did not observe performance increase for any of them. For the stop-words removal we only consider neutral stop words as we negative stop-words can change the meaning of a sentence. For the special characters removal it included "#", "@" plus basic punctuation. Finally we replaced some smileys by their meaning: "xD" was replaced by "smile", "<3" was replaced by "love", ":c" was replaced by "sad".

As we can see the best setup was to not use any preprocessings.

2) Finding the best hyper-parameters:

In order to tune hyper-parameters we create 3 sets: a training set containing 70% of the data, and a validation and a test set each containing 15% of the data. The model with a combination of hyper-parameters is trained on the training set, and evaluated on the validation set (for finding best hyper-parameters). We thus obtained the best combination of hyper-parameters and evaluated the model on the test set.

Here are the hyper-parameters that we tuned:

- $minCount$: minimal number of word occurrences
- $wordNgrams$: max length of word n-gram
- $bucket$: number of buckets
- lr: learning rate
- dim: size of word vectors
- ws: size of the context window
- $epoch$: number of epochs

Our algorithm selected us the following parameters: $minCount$ = 1, $wordMgrams$ = 3, $bucket$ = $10,000,000$, lr = 0.01, dim = 200, ws = 1, and $epoch$ = 7.

The experiments can be seen in the $fastText\_model\_tuning.ipynb$ notebook.

3) Results:

We obtained an accuracy of 0.8732 on our test set and an accuracy of 0.8680 on the provided test set evaluated on AIcrowd.

V. Approach using BERT

In order to perform even better performances, we search the state-of-the-art methods for natural language processing and more precisely text classification and sentiment analysis. We found the $Transformer$ [5] architecture that was introduced in 2017 and uses $encoder$ and $decoder$ to transform a sentence. It differs from the previous top-notch models with an architecture that do not use any recurrent networks (e.g. GRU, LSTM). An interesting aspect of this architecture is that it does not need to process a sentence in order (this enables much more parallelization during the training). Using this new technology, Google developed in 2018 a new tool for natural language tasks called Bidirectional Encoder Representations from Transformers: BERT [6]. This model is already pretrained on a huge amount of general language data.

We used a PyTorch implementation of BERT (already pre-trained) and trained our model with the tweets we had.

A. Disclaimer

BERT being a complex deep-learning model, the training requires a lot of computational power. Unfortunately, we did not have any cloud platform GPU access nor adequate hardware devices to perform the training on the full dataset. We therefore performed as much as we could do, and trained our model on 80% of the small tweets dataset (which corresponds to 8% of the total data provided). The training

took more than 30h for only one epoch and the full pipeline (including train and test data processing) took around 35h.

Nevertheless, the accuracy we obtained was good. We decided to keep this option even if we weren't able to train completely our model, as our results were very encouraging.

Finally, as the tasks was requiring Deep Learning knowledge that we were just starting to discover, we followed a tutorial [7] [8] and used some of its code to use BERT with PyTorch.

B. Preprocessing

First of all, as always, we need to apply a basic preprocessing to our text file. We therefore remove the <user> and <url> tags from it.

Once this is done, we need to make our dataset BERT friendly. Indeed, our BERT model takes .tsv files with dataframe containing 4 columns (index, label, alpha, and text). We finally split our dataset into a train set and a test set.

This preprocessing step can be seen in the $./BERT/data\_prep\_tweets.ipynb$ notebook.

Then, we need to convert our data to features that can be given to the model. For this step, we used classes code given in the previously mentioned tutorial. Multiple python classes are defined to transform our tsv files into a list of objects (for each row) corresponding to input examples. Finally, as BERT is a neural network, we could not give him directly text features but first tokenize our tweets. BERT has a constraint of length for a sequence of 512 tokens. In our case, we use a maximum of 128 tokens to reduce the training time.

C. Training

Our model is a PyTorch implementation made by Hugging Face [9], it uses an Adam optimizer and download automatically the Google pretrained BERT model for us. As discussed above, due to our lack of adapted material, our training is only made on a small part of the full dataset. We therefore trained during only one epoch with batches of size 24. The final classification layer has also already been added so we only need to specify it as a parameter.

The result of our training is 3 files, the model containing all the weights ($pytorch\_model.bin$), a configuration file that gives metadata about the model file ($config.json$), and finally a text file containing the vocabulary extracted from the different tweets ($vocab.txt$).

The data adaptation and model training can be seen in the $./BERT/BERT\_train\_tweets.ipynb$ notebook.

D. Results

Once our model has been trained, we need to evaluate it on a test dataset. To do so, we load our model files (previously archived into a tar.gz file and placed in our cache folder), apply the preprocessing to the test data, convert it into BERT features and then run our model to predict the labels.

For this model, we did two training with different dataset size. The first attempt goal was to (roughly) evaluate the model performances without having a long training time. The second one is the longest training and obtained the best performances we were able to do. We present here the results for those two attempts with BERT model.

First attempt:

Train set	1576 rows
Test set	1576 rows
Total time	3 hours
Test loss	0.3424
Test accuracy	0.854

	True	**False**
Positive	731	156
Negative	615	74

Final attempt:

Train set	157576 rows
Test set	39394 rows
Total time	35 hours
Test loss	0.2855
Test accuracy	0.8751

	True	**False**
Positive	17335	2717
Negative	17139	2203

VI. Discussions

We would like to discuss here the different paths that we didn't explore due to a lack of time, computing power, or skills.

First of all, we were not able to train our BERT model on the full data so this is obviously our first vector of improvement.

Then, we found multiple other Transformer models (especially into the Hugging Faces library [9]) that were introducted recently (GPT-2, RoBERTa, XLM, DistilBert, XLNet, CTRL, ...) and have impressive performances. Those are state-of-the-art models for Natural Language Understanding and Natural Language Generation. We could have tested them on our dataset in order to obtain better performances. Those models gives very interesting perspectives for future NLP applications. This project raised our interest for this domain and its recent discoveries [10].

REFERENCES

[1] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013.

[2] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," 2013.

[3] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," 2016.

[4] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of tricks for efficient text classification," 2016.

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017.

[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2018.

[7] T. Rajapakse, "A simple guide on using bert for binary text classification," 2019.

[8] ——, "Bert binary text classification," https://github.com/ThilinaRajapakse/BERT_binary_text_classification, 2019.

[9] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, and J. Brew, "Huggingface's transformers: State-of-the-art natural language processing," *ArXiv*, vol. abs/1910.03771, 2019.

[10] J.-C. Chappelier and M. Rajman, "Cs 431 - introduction to natural language processing," 2019.