Tweet Sentiment Classification

Adam Chellaoui
adam.chellaoui@epfl.ch

Douglas Bouchet
douglas.bouchet@epfl.ch

Marina Rapellini
marina.rapellini@epfl.ch

December 23, 2021

Abstract - This paper details our approach to address the « EPFL ML Text Classification Challenge » where the aim is to classify whether a tweet message used to contain a positive ':)' or negative ':(' smiley. Different approach were used: preprocessing, various word representations, classical machine learning models, neural network and the most recent transformers architecture. The best result was obtained fine tuning a BERT pretrained model, giving an accuracy of 0.905.

I Introduction

Sentiment classification is a task in natural language processing (NLP): the goal is assigning a label (positive, neutral or negative) to text by detecting emotions or opinions expressed by its contents.

Social media data are of great interest in NLP due to the utility of the information that can be retrieved from them and to the challenge they represent for automatic processing of text; they are indeed very challenging to process due to the presence of slang, emojis, neologisms. Tweets are known to be extremely complex data to interpret, because of the language used, free and not attentive to the grammar; they have been used in the late years for many purposes such as marketing and sentiment analysis. The dataset of this projects consists of tweets to be analyzed and it is provided by the AI-Crowd platform [13].

In the following section, data exploration is discussed. In Section III the choices made for pre-processing are explained; in Section IV, V, VI the results obtained with the considered models are presented and the performance of the models and the word representations used as well as the relevance of data pre-processing for this task are discussed.

II Exploratory Data Analysis

The dataset consists of two classes, positive and negative tweets organized in 2 labeled datasets: a small one, consisting of 200,000 tweets, and a larger one of 2,500,000 tweets. The data sets are balanced. A key part of any Natural Language Processing task is data pre-processing; before applying any binary classification algorithm, we explore the data to understand what text pre-processing techniques to apply and what text representation to use; a careful preprocessing can help to emphasize the sentiment of a tweet as well as to reduce noise and the dimensionality of the vocabulary. It is important to note that almost 12% (288,789) of the tweets are duplicates and that all words are in a lowercase format.

III Preprocessing

We decided to perform the following preprocessing steps in the attempt to improve our sentiment classification:

Contractions: we decided to expand the contracted words and expressions in order to recover the sentiment (for example won't–>will not).

Numbers: all the numbers were replaced by the <number> tag to unify the information about the presence of numbers.

Auxiliary tags: we removed the twitter special syntax like indications of "RT", < url > or < users > because not relevant to determine the sentiment.

Emojis: we decided to combine different emojis that express similar sentiment and to replace them with one tag; for example :p was replaced by <laughface>.

Extra letters: we decided to delete multiple occurrences of a character and to replace it with a single character (for example: "extraaaaa"–>"extra").

Repeated punctuation: repeated punctuation from tweets was eliminated and any occurrence was replaced by the tag <emphasize>.

Hastags: hastags are really popular and they express sentiment; we decided to remove the # symbols and replace them with the tag <hashtag>; this indicates the presence of a hashtag and may inform of the sentiment of the words it contained.

Lemmatization: it consists in analyzing words such as 'worse' and 'bad' together by grouping them; we did not implement this during the pre-processing but we implemented it for some embeddings when it could increase the performance.

Laugh: different laughs such as lol, hahaha, ahah were replaced by <lolexpr>.

Even though removing the so-called stopwords is common for normal NLP tasks, when it comes to sentiment analysis it is a bad practice because it would eliminate words which show the emotion of the person writing; for

this reason we have decided not to remove them during the pre-processing [14].

IV Representation Learning

It is essential to find a numerical representation for the components of the text that is used as input data. Here the different techniques that have been used are presented.

IV.I Naive Text Representation

We started with some simple methods (i.e we are only interested in the frequency of words); even if not complex, some of these models can still achieve satisfying performances.

BoW. This model extracts features from text by counting the frequency of words in a document; it is called a "bag" of words because any information about the order or structure of words in the document is discarded. It is simple to implement, but it completely fails to recognize the context of a message, it loses all similarities between words and the most common words such as 'the', 'a', 'an' become the most important ones, which does not improve sentiment prediction ability. While implementing BoW, stopwords are removed; the collection of tweets is converted to a matrix of token counts with a value of 10000 as max-features (i.e given a corpus, the maximum number of words that will be used as the vocabulary): increasing it even more does not produce a higher accuracy.

TF-IDF. TF-IDF weighs a term's frequency (TF) and its inverse document frequency (IDF): each word/term that occurs in the text has its respective TF and IDF score. We used the sklearn Library in order to train a TF-IDF vectorizer, and convert our training data into a TF-IDF matrix. The hyper parameters we add to tune were the number of n-gram and the maximum number of features. For the n-gram, after trying the values {(1,1), (1,2),(1,3),(2,2)}, the (1,2) n-gram gave the best results in term of accuracy. This could be explained by the fact that during our pre-processing we split for example "wasn't" into "was not". As vectorizer allows bi-gram, it can deal with "was not" as one word, which can help to determine sentiment of a tweet (we can also think of "not bad" for example which is equivalent to "good"); the (2,2) gram cannot handle single worlds, thus we might lose a lot of features. For the maximum number of features, we tried values {5k, 30k, 100k, 450k}. The 100k choice was the best one while 5k, 30k were very limiting in terms of size of vocabulary that could be used, it was kind of "under fitting" the sense of the words. Our vocabulary consists of 450k words, so representing them using less than 8% of the original word doesn't succeed in capturing the features of the tweets; the 450k choice was too big, and it allows all words present in the corpus to be represented. We could more say that we "over fit", adding noise to our numerical data.

IV.II Word Embeddings

Word embeddings are a class of approaches for learning representations of words and documents that use a dense vector representation for each word. It is an improvement of the traditional bag-of-word models encoding schemes where large sparse matrices are created.

Word2Vec. Word2Vec is a word representation technique that uses a two-layer neural network to learn the linguistic contexts of words in a large text corpus [6]. The training is made under the assumption that words that have a similar meaning, are likely to appear in the same « context ». We trained our embedding model using the Gensim Skip-Gram implementation of Word2Vec, which relies on the central word to predict the surrounding « context » words, by summing in its objective function the log probabilities of the words in the window to the left and right of the target word. It can be observed that skipgram models works better with subword information than the alternative model cbow [**6**]. We fixed a window-size of 5, a dimension n=500, and min-count=1 because Word2Vec cannot handle out-of-vocabulary words well. At the end, each word is represented as a fixed n-dimensional vector, encapsulating its meaning.

FastText.FastText extends the concept of Word2Vec and it was created by Facebook's AI Research lab in 2015; it is based on the idea of using word n-grams (phrases), and character n-grams (subword embeddings) [7]. The embeddings models we have presented were based on the assumption that the vocabulary consists of words. The use of word n-grams allows to capture meaning that cannot be assigned simply to the constituents of the sentence. The use of character n-grams captures similar word variations, and it enables to process unseen words. We constructed our embedding using the FastText open source library, by imposing a vector dimension n=300.

GloVe. GloVe is an unsupervised learning algorithm for obtaining vector representations for words. The idea is to use a co-occurrence matrix in order to derive some relationship between words that appear together; the vector space of the glove embedding maps words with similar meaning to close values in the vector space. We loaded a trained glove embedding [**2**]; one advantage is that it has been trained on a twitter's data set, thus it is likely that the words we will have to embed for our prediction task will have an embedding vector available in this set (we only had 4.5% of words without any embedding.) The words without any embedding were mapped to the null vector (only 0's). Another advantage is that it offers multiples dimensions of embedding, that we have tuned in order to find the best one for our task. We had the following choices for the embedding dimension: $\{25, 50, 100, 200\}$. After trying each of them, the 200 dimension embedding gave the best results. This could be explained by the ability of the 200d embedding to capture enough features (i.e allows words with different topics to be mapped to distant vector in the vector space repre-

sentation); this may not have been the case for lower dimensions. At the same time, the 200d didn't capture too many features, which can prevent the embedded vectors to be compared, because of the curse of dimensionality [**3**].

V Machine Learning Models

As a reference point, different common models were used: Random Forest, Logistic Regression, SVM and Naive Bayes (Table 1). A 5-fold cross validation was performed using 10% of the dataset, in order to define the best hyper-parameters for each of the models trained. Then we used 80% of the full dataset for training, and 20% for the validation set.

VI Deep Learning Models

As Deep Learning models usually outperforms classical machine learning approaches by huge margins, we tried to see the results they bring in resolving our task.

LSTM. We used the following architecture for our LSTM [**4**]:
(Conv1D)×3→MaxPooling→BiDirectionnalLSTM
→(Dense->Dropout)×3→Dense(1)
The Conv1D layer is useful to capture topic of tweets. Next we apply a MaxPooling [**11**], which identify the most important feature inside our tweet. We then use a BidirectionnalLSTM, which helps to figure out the influence of the word's position inside a tweet. The last 3 hidden layers increase the complexity of the network. Finally, there is one output neuron using a sigmoid activation function, to make binary classification. We also add some dropout between some of the layers, in order to limit over fitting.

We used the FastText because it was giving good results on some more simple models (see Section VI).

We had to use a padding in order to have each of our embedded tweet with the same length (the number of time steps used by the LSTM). We tuned the following parameters:

- pre/post padding: by comparing our accuracy on validation set and according to [**5**], we decided to use pre padding as it was giving better results

- size of padding: we tried the following values: $\{8, 15, 20, 22, 25\}$. We found that 22 was the best one. One explanation is that too small padding doesn't capture enough features of each tweet whilst too large padding introduces a lot of noise for all the tweet that needs to be padded with 0's.

- batch size in [10, 64, 128, 256, 512, 1024]: the 512 size was the best one, with a nice trade-off between performances and accuracy (the lower the batch size was, the longer the training was).

For the training, we used the Adam optimizer with a learning rate of 0.001 and a weight decay of e^{-6}. Among the learning rates $\{1e^{-4}, 1e^{-3}, 2e^{-3}, 4e^{-2}\}$ $1e^{-3}$ gave the best results. The lowest one was performing correctly, but required many epochs compare to the higher ones. The highest one prohibits any improvement giving an accuracy of 50%.
We tried the following epoch: $\{2, 5, 10, 20\}$. Epoch inside (5, 10) were giving best results. However when using more than 10 epochs, we started to observe some over fitting, with the training accuracy increasing and the validation testing decreasing.

To tune all these parameters, first we had to reduce the size of the training set. We used google Colab in order to have a faster training. There was a limitation on maximum RAM memory that we could use of 12GB. However the embedding of the complete training set far exceeded this limit and it was impossible to run without any crash. We decided to use 10 % of the total data in order to limit the RAM used (this was the maximum training size we could use). We would probably have obtained better results with the complete dataset, also because the number of parameters of the LSTM was large ($\sim$ 3 million of trainable parameters), requiring a lot of data. We tried some grid search in order to find some of them, but we didn't try a global grid search over all hyper parameters (this would have been roughly 720 trials of 10 minutes each, giving a total training of 5 days.)

After tuning of all these hyper parameters, we obtained an LSTM achieving 84% accuracy on our validation set.

We have seen that the LSTM has brought us the possibility to take into account both the position of a word inside a tweet and also the main topic. We will see in the next section that there is a method for making inferences between position of the words inside a tweet and its sentiment more efficient.

BERT. In order to perform even better, we looked for state-of-the-art methods in the field of natural language processing. We came across BERT, which stands for Bidirectional Encoder Representations from Transformers. It is a very sophisticated language representation model developed by Google AI Language researchers in 2018, that has been an outstanding breakthrough in the field of NLP [10]. The model relies on the encoder part of a Transformer. The Transformer architecture is a self-attention mechanism that learns contextual relationships between words within a text. Until now, the text representation models that we considered were context-independent: the same word is represented with the same vector independently of his position in the sentence, and the surrounding words. On the other hand, BERT will generate two different vectors for the same word, used in two different contexts and here is all its strength.

Our model is based on a pre-trained BERT model named « bertweet-base-sentiment-analysis » and available in the Hugging Face library [8]. This model itself is based on « Bertweet », which is a RoBERTa based model trained on a dataset of 850 millions of english tweets [9],

Model	BoW	TF-IDF	Glove	Word2Vec	FastText
Random Forest	0.725	0.653	0.698	0.639	0.762
Logistic Regression	**0.813**	**0.85**	**0.787**	0.779	**0.805**
Linear SVM	0.758	0.783	0.781	**0.780**	0.804
Naive Bayes	0.748	0.795	0.652	0.615	0.637

Table 1: Classification Accuracy of the Classical Machine Learning Models on validation set

and it adds an output classification layer to predict the sentiment of a text. This last layer makes use of the [CLS] token, which encapsulates the global meaning of the sentence, and performs an optimization for the classification task using a Cross Entropy loss. The latter includes 134 billion of trainable parameters.

We finetuned the « bertweet-base-sentiment-analysis » on our dataset, using a PyTorch implementation. First, the data is tokenized with the provided tokenizer with padding and truncation, and encapsulated in a Tweet class that we wrote, in order to feed a PyTorch dataset to the model per batch. Again we use 80% of the data for training and 20% for validation. The training is performed using the Trainer class provided by Hugging Face in PyTorch, with a learning rate of e^{-5} for Adam optimizer.

The training was performed on Google Colab, running a Tesla P100 GPU with 52 Go of RAM, and took 30 hours for 4 epochs on the full dataset. We tried to train for one more epoch, but we got a decrease of 0.003 in accuracy, probably because we end up over-fitting. Thus we stood with the 4 epochs trained model.

VII Results & Discussion

Baseline Models. Table 1 presents the validation accuracy of the baselines Machine Learning models. We are immediately struck by the performance of the "naive" text representation techniques. Indeed, the best result is obtained with TF-IDF, which gives an accuracy of 0.850. Similarly, BoW, whose implementation is the simplest, also does very well with regard to dense word embeddings. Even though BoW and TF-IDF don't provide vector representation for a single word, they are good and simple methods for classifying documents as a whole.

Moreover, it is interesting to notice that the accuracy of the word embeddings for the Naive Bayes classifier are lower; this is because Naive Bayes makes the assumption of conditional independence between every pair of features, which is not the case when constructing word embeddings. Note that Random Forest for BoW and Fasttext perform better because a depth of 8 instead of 3 was used.

Advanced models. Table 2 shows the results obtained with LSTM and the fine-tuned BERT model. The models that achieved the best scores overall were the transformers based one with an accuracy of **0.905** on the test set and **0.930** on validation set.

Due to the pretraining on tweets background of the BERT model used and the high complexity level that allows full contextualization consideration, transformers seem to confirm their superiority in sentiment analysis tasks. As expected, LSTM has a lower performance because of the smallness of the dataset used for training.

Model	Validation	AiCrowd
LSTM	0.844	0.835
BERT	**0.930**	**0.905**

Table 2: Validation and test Accuracy of DL Models

General Observation. It is important to remark that, despite its simplicity, TF-IDF performs very well for text classification tasks, thanks to an adapted preprocessing pipeline.

One main problem we often encounter in machine learning when tuning the hyper parameters with grid search is the exponential number of operations as a function of number of parameters to tune. The Bayesian optimization is a strategy for global optimization of black box function. With a bit of work, our hyper-parameters tuning can be seen as a black box optimization problem and one could use Bayesian optimization to explore a wider range of parameters and their values.

VIII Conclusion

Handling textual data from social media, often full of peculiarities and errors, is not straightforward. In this work, we have presented different preprocessing techniques, text representations learning, machine learning and deep learning models, and the more recent transformers based architecture BERT [10] that gives the best result for this text classification task. BERT models have impressive performances and they become state-of-the-art models for Natural Language Understanding, paving the way for many new possibilities. We understood that the type of word representations learning is crucial for performance.

As a future work, one can continue fine tuning our BERT model, that has been made available on the Hugging Face platform [12] or analyzing the effect of an ensemble learning method aggregation our LSTM and BERT models, and why not adding a GRU neural network to improve our performance.

References

[1] Slides lectures, https://github.com/epfml/ML_course/tree/master/lectures

[2] Jeffrey Pennington,Richard Socher,Christopher D. Manning, "GloVe: Global Vectors for Word Representation", 2014, https://nlp.stanford.edu/projects/glove/

[3] Zi Yin, Yuanyuan Shen, "On the Dimensionality of Word Embedding", 2018, http://proceedings.neurips.cc/paper/2018/file/b534ba68236ba543ae44b22bd110a1d6-Paper.pdf

[4] Lingyan Zhou, "Review Sentiment / FastText, LSTM vs CNN", 2018, https://www.kaggle.com/zhoulingyan0228/review-sentiment-fasttext-lstm-vs-cnn

[5] Dwarampudi Mahidhar Reddy, N V Subba Reddy, "Effects of padding on LSTMS and CNNS", 2019, https://arxiv.org/pdf/1903.07288.pdf

[6] Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean, "Efficient Estimation of Word Representations in Vector Space", 2013, https://arxiv.org/pdf/1301.3781.pdf

[7] Piotr Bojanowski, Edouard Grave, Tomas Mikolov, Armand Joulin, "Bag of Tricks for Efficient Text Classification", 2016, https://arxiv.org/pdf/1607.01759.pdf

[8] Juan Manuel Pérez, Juan Carlos Giudici, Franco Luque, "pysentimiento: A Python Toolkit for Sentiment Analysis and SocialNLP tasks", 2021, https://arxiv.org/abs/2106.09462

[9] Dat Quoc Nguyen, Thanh Vu, Anh Tuan Nguyen, "BERTweet: A pre-trained language model for English Tweets", 2020, https://aclanthology.org/2020.emnlp-demos.2.pdf

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", 2018, https://arxiv.org/abs/1810.04805

[11] Lei Shen, Junlin Zhang, "Empirical Evaluation of RNN Architectures on Sentence Classification Task", https://arxiv.org/pdf/1609.09171.pdf

[12] Adam Chellaoui, "tweet-sentiment-analyzer" https://huggingface.co/adam-chell/tweet-sentiment-analyzer

[13] AI-Crowd platform, https://www.aicrowd.com/challenges/epfl-ml-text-classification

[14] Saif Hassan, Fernández Miriam, He Yulan, Alani Harith (2014), "On stopwords, filtering and data sparsity for sentiment analysis of Twitter", http://oro.open.ac.uk/40666/