Machine Learning Project: Tweet Classification

Jacopo Ferro, Majdouline Ait Yahia, Victor Carles
Machine Learning CS-433, EPFL, Switzerland

***Abstract*—The goal of this project was to build a fine-tuned Machine Learning model for binary classification of tweets based on their sentiment. The model is designed to recognize if a tweet would have contained a ":)" smiley or a ":(" smiley, which is a proxy for classifying whether the tweet is positive or negative. Five dataset were provided: two full training datasets containing roughly 1 250 000 tweets each, with respectively positive and negative tweets, two samples of the training tweets containing about 10% of the full data, and a test dataset of 10 000 unlabeled tweets to submit our predictions with label encoding as +1 and -1 for ':)' (positive) and ':(' (negative) respectively. To evaluate our models we used randomized splitting on the partial and full training data as well as cross-validation, then final testing with submission on AICrowd (www.aicrowd.com) which would then compare the obtained labels with the expected ones and rank based on the accuracy and F1-Score.**

I. Introduction

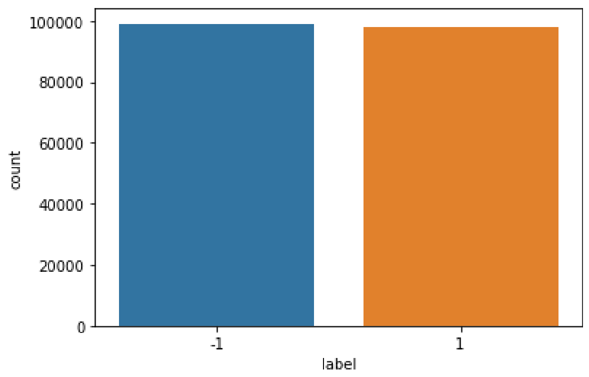
The paper is divided into four sections. The first section consists in the dataset analysis and the pre-processing. Since the full training dataset contains more than one million entries, it is essential to make sure that the dataset is well balanced and contains only useful words for feature embedding. This is why we consecrated the first part of our project in understanding the dataset and pre-processing it, removing useless words and applying transformations. The second section presents the different embeddings we used throughout the project. Finding a relevant embedding for our sentiment classification is primordial to obtain exploitable features for our models. We tested multiple pre-trained embeddings and designed our own to yield an optimal training dataframe from tweets. The third part presents the models that we used and the accuracy that resulted from them.

II. Dataset analysis and Pre-Processing

The first step of our pipeline is preprocessing of the text. Tweets are fundamentally different in their semantic meaning compared to more formal corpora because slang, acronyms and punctuation can hold information that would be considered noise in another context. Preprocessing and tokenization is necessary to obtain a vectorized representation of our data that can be used by our classifier.

Balance the dataset

The data was balanced, so we did not have to manually account for very large differences in the number of negative versus positive tweets. We did however notice that some training samples did not match what a human would consider necessarily as positive or negative and some tweets contain sequences of tweets and answers, as well as 'spam' or duplicated tweets, so further research should be made on having a better raw training set. This was not the main problem in our analysis and we decided to improve our model on top of the raw data with preprocessing done on each tweet, trusting the provided labels.



Remove tags and stop words

Every username mention and URL is represented in the form of tags in the dataset for simplicity, i.e $< user >$ or $< url >$. As these tags don't contain any information on the sentiment of the tweet, we removed them. Moreover, we removed stop words based on the english dictionary ("is", "the", "and" etc) because they usually don't carry any useful information for a training set as well.

–insert plot of proportion of stop words and tags –

Stemming, lemmatization and acronyms

Tweets contain a huge amount of acronyms and slang expressions, i.e words that are not in the english dictionary and that might be hard to use for our models. That is why use a non-exhaustive list of acronyms-to-words tuple in order to replace acronyms by their equivalent in the english language, e.g "idc" is replaced by "i don't care". This is particularly useful when we will embed words based on their context, aswell as when using n-grams. We also truncate verbs and tense to its prefix by using Stemming and Lemmatization to avoid over-representing features with similar semantics and syntax.

– insert plot of proportion of non-english words –

Elongated words, repetitions and emoticons

Tweets also contain misspelled words, or repetition of characters. This can be particularly problematic as words such as "alriiiighttt" and "alright" can be interpreted as being different because of their spelling. That is why we reduce elongated words. We also uncased all words, which would not necessarily always be a good idea as capital letters can carry information about potential negative sentiment, but we figured that most embeddings models we use are treating words as uncased anyway. Finally, we converted most common smileys to words, e.g ":)" becomes "happy". This is done because it can be very hard for a machine to understand the meaning of a smiley compared to a human being, whereas a word immediately carries a semantic definition that can be represented as a vector.

III. Embedding vectors

In order to train our model, we need to embed each tweet. Depending on the types of algorithm we use, the dimension of the vector and its values might vary. It was an important part of the project to find an optimal embedding, and see how each of these embeddings would fit with each model.

TF-IDF

TF-IDF vectorization is a more refined successor to the BOW matrix representation of texts, in that not only it counts the occurrances of each token/word but it assigns more weight to rare words and normalizes the matrix in a way that naturally filters out noise. TF-IDF is a relatively simple technique and is usually superpassed by more complex and nuanced word embeddings that attempt to extrapolate the semantic meanining of words, based on their spacial locality in the corpus and even their position in the text for advanced deep learning techniques. TF-IDF is somewhat discarded for complicated tasks in big language models, text-to-text generation and precise sentiment analysis, but we dove deeper and we found that for our purpose it performed not only well, but was also much faster than other techniques. Indeed we want to draw attention to an aspect that we find important: the cost of training and inference, both in terms of computational resources, of time and of human resources (complexity of implementation itself). TF-IDF when paired with n-grams and some preprocessing becomes very powerful for a binary classification on a relatively big dataset of short texts. The sparsity of a huge feature matrix was never a problem because modern libraries like Numpy can handle very well keeping track of the non-zero values of each data point, so even with an over-parametrized model consisting of millions of features, this mapping into higher-dimensional space leads to extremely fast classification training and prediction with classifiers like Multinomial Naive Bayes and Linear SVM. It does lead to overfitting which can be reduced with regularization or boosting as well as finer preprocessing. In particular, TF-IDF can be paired with a more precise definition of N-Grams. We created a list of 'phrases' consisting of bigrams, trigrams, quadrigrams that are present in at least 10 different tweets. This eliminates the need to project every n-gram and greatly reduces the number of features while still keeping a high enough dimension to make the classification easy for a linear classifier. Surprisingly this preprocessing did not change much the test results but we think this approach is more robust than the blind inclusion of all n-grams in each tweet. We reached 0.848 accuracy and 0.850 F1 score with the classical 1 to 4-grams and TF-IDF with Linear SVM classifier with no regularization. We reached the same acuracy but 0.849 F1 score with the phrases-grams.

1) Parameters: Randomized Grid Search with 5-fold Cross Validation was performed in a pipeline created with TF-IDF as vectorizer and MultinomialNB as classifier:

- N grams: results improve up to 4-grams, add useless complexity with 5-grams (no or significant improvement) and worsen the results with 6-grams and above.
- Max Number of features: no cap on number of feature is best.
- Minimum frequency of co-occurrence in corpus: does not change drastically the results, using 1, 5, 10, 15 we found that 5 is best.
- Ignoring over-represented words: ignoring tokens present in 90%, 95%, 98% of corpus or no cap, we found that no cap is best.
- Tokenizer: we found that a tweet tokenizer which is case sensitive and only shrinks repeating characters to a maximum of 3 is a best. That is, 'Heyyyyy' becomes 'Heyyy', not 'hey'.
- Stop-words: no stop words is best at this stage, using 'english' corpus cleans out too many that in tweets retain information.

GloVe

GloVe is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space. Words are hence represented as vectors in an Euclidean space, where semantic similarity between words is interpreted as distance between vectors. Instead of training an algorithm from scratch, we imported some pre-trained word vectors from GloVe, e.g glove.twitter.27B.- in either 50, 100 or 200 dimensions. The larger the dimension of the vector is, the more subtleties between words the vector can enhance. Although some larger vectors exist, e.g 300d, we realized that 200d was sufficient for our models, as giving more complexity to the vector space would add too much of a computational trade-off compared to the benefit in accuracy. We tokenized each tweet, firstly by our own means and later on using Tokenizers from libraries such as Keras. Tokenizing a tweet means that the tweet will be fragmented into an array of words. Each word would then be translated to its embedding vector using the GloVe dataset. This is why the pre-processing part is very important when using GloVe, because some words that are mispelled, acronyms, smileys, emoticons or even elongated words might not have corresponding vectors in the dataset of embeddings, which could result in having a poor representation of the tweet. Moreover, since each tweet must be represented by only one vector, there stood the question of aggregating the vectorized tokens. One easy way to do that was to average vectors, resulting in a dataframe of only one feature-vector per tweet, but as we will state later in this paper, we found some better ways to represent tweets.

Word2Vec

Word2Vec is different than GloVe in the sense that it leverages co-occurrences within local context, instead of a whole dictionary. Word2Vec is based on two main architectures, Continuons Bag of Words (CBOW) and Skip-Gram model. CBOW essentially tries to predict a target word from a list of context words, whereas Skip-Gram does the opposite by taking the current word as an input and tries to accurately predict the words before and after this current word. Using the gensim library, representing a tweet as a feature vector is similar to GloVe, i.e using the Word2Vec pre-trained model to compute every token vectors and then averaging them. In practice, we found out that Word2Vec and GloVe were approximately giving the same results.

Fasttext

FastText is an open-source, free library from Facebook AI Research for learning word embeddings and word classifications. It can be used to find semantic similarities, just like the other embedding algorithms we have seen, but has the advantage of also being a classifier that can be trained on large datasets in an efficient way. The Fasttext classification model relies on two possible embedding algorithms, CBOW and Skip-gram, similarly to Word2Vec. Instead of using a Linear Classifier like most Deep Learning algorithms that are slow and expensive on huge datasets, Fasttext uses a Hierarchical Classifier, where every node in a binary tree represents a probability. A label is represented by the probability along the path to that given label. This means that the leaf nodes of the binary tree represent the labels. As being both an embedding algorithm and a classifier, Fasttext has the advantage of being already fully implemented and very straightforward to use.

IV. Models

Naive Bayes

Naive Bayes (Multinomial Naive Bayes Classifier) produced by far the fastest results on TF-IDF vectoriazion. This is a probabilistic method so it's not surprising as it does not need to actually 'learn' and uses the matrix to find the best guess on the label based on the TF-IDF vectorization. Note however that the simplicity of the model can be seen also in the training data. Indeed in constrast to the Linear SVM which overfits a very high dimensional space (over-parametrization), Naive Bayes is not complex enough to even fit the training data. Nonetheless it achieved only 0.03 Accuracy and F1-Score less than the over-fitting, unregularized Linear SVM classifier. On test set NB achieved Accuracy of 0.822 and F1-Score of 0.834. The training and predictions took a total of 1min and 50s on a laptop, accounting also for the vectorization of all the datapoints.

Logistic Regression

A simple classifier we firstly used is Logistic Regression, a model consisting of a logistic function (sigmoid) that maps the inputs to values between 0 and 1. Translating labels from -1 to 0 outcame as being the best way of making use of the logic behind our models, especially Logistic Regression. We would then translate the prediction back from 0 to -1 for submission. Such re-labeling would always be done after preprocessing. We tried Logistic Regression both with GloVe and Word2Vec embeddings. We used the method from Scikit-learn, which implements a binomial logistic regression and where regularization is applied by default to prevent overfitting. With a test-train splitting of the full training dataset, we obtained an average accuracy of 0.748 for the validation data. On the test dataset, we obtained an accuracy of 0.729 and an average F1-Score of 0.729 with GloVe.

Support-Vector machine

Another simple and interesting classifier we used is Support-Vector machine. SVM maps training examples to points in space so as to maximise the width of the gap between the two categories. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall. SVM from Scikit-learn uses kernel trick for non-linear classification, but since text classification is often linear, we use LinearSVC for optimization of the running time. Using Word2Vec the accuracy is O.74 on the test-train split, and 0.729 on AICrowd with a F1-Score of 0.730, which is close to the results obtained with logistic regression. Some hyperparameters could be tuned, but we decided to use more advanced algorithms in order to find a better accuracy. As mentioned above we obtained 0.848 accuracy and 0.850 F1-Score by using Linear SVM on TF-IDF vectorization. Randomized Grid Search with Cross Validation showed no major improvements on using different regularization parameters with TF-IDF but it did allow for faster (and correct) computation on more dense representation with word embeddings of 100 and 300 dimension. The training + vectorization + predictions of test data with LinearSVM (optimized in sklearn using LinearSVC) took a total of 2min 20s on a laptop.

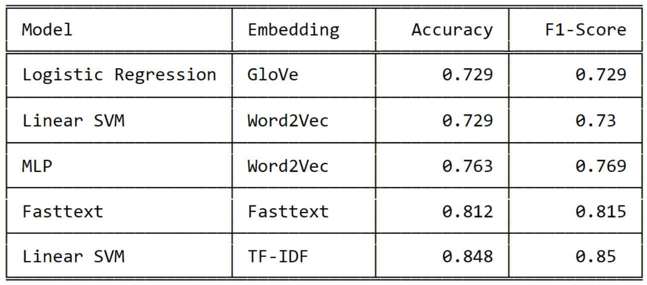
A. Multi-layer Perceptron classifier

More advanced models consist in using Neural Networks. A MLP consists of at least three layers of nodes: an input layer, a hidden layer and an output layer. Except for the input nodes, each node is a neuron that uses a nonlinear activation function. MLP utilizes a supervised learning technique called back propagation for training. The more we add hidden layers, the more accurate the model becomes, but as a trade-off, becomes very slow to train. This is why we used Google Colab to run our neural networks, which fasten greatly the process by running the computations on a Google GPU. We tried conduction a Grid Search on the MLP Classifier to find optimal parameters, but it was too expensive in terms of computational time, so we rather opted for testing parameters manually. Using the MLP Classifier from Scikit-Learn, the best accuracy we found for the MLP Classifier with Word2Vec embedding was 0.763 with a F1-Score of 0.769 on AICrowd, using a hidden layer of shape (10,2), a lbfgs solver and an alpha of value 1e-5.

Fasttext

As stated in the first section, Fasttext is both an embedding algorithm and a classifier. It has the advantage of being pre-trained and very fast. Instead of tokenizing, we can directly input lines of text (tweets) containing the labels in the form ”__label__0” or ”__label__1” as a prefix of the sentence. Fasttext classifier yielded very good results for almost no computational efforts, as we obtained an average of 0.812 for the accuracy 0.812 and 0.815 for the F1-score on AICrowd. We figured out that in this case, pre-processing would reduce the accuracy. This can be explained by the fact that stemming and removing stop words can strip some informations on sentiment from the tweets. That is why for most models, we used cleaned datasets that had no stemming, no lemmatization and no stop words.

Results



Model	Embedding	Accuracy	F1-Score
Logistic Regression	GloVe	0.729	0.729
Linear SVM	Word2Vec	0.729	0.73
MLP	Word2Vec	0.763	0.769
Fasttext	Fasttext	0.812	0.815
Linear SVM	TF-IDF	0.848	0.85

LSTM

Some more advanced Neural Networks Architecture are Recurrent Neural Networks. Long Short Term Memory (LSTM), which we found to be a good Deep Learning algorithm for text classification. LSTM is effective in memorizing important information using a FORGET gate layer. Moreover, LSTM works on the elimination of unused information, and as we know the text data consists of a lot of unused information which can be eliminated by the LSTM so that the calculation timing and cost can be reduced. We used the library Keras to import the Sequential model we would use as base to build our LSTM neural network, and the layers such as Dropouts to reduce the effect of bias, common Dense layers and other useful layers. Running on Google colab a simple LSTM with 128 neurons, relu activation function and a 0.2 dropout parameter yielded an accuracy of 0.819 and a F2-Score of 0.826 on AICrowd.

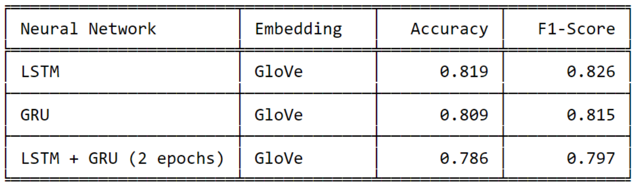
GRU

Another interesting type of Recurrent Neural Networks Architecture we used is Gated recurrent units (GRU). It is similar to LSTM in the sens where it also has a Long short term memory, but with fewer parameters as it has no output gate. It's performance on text classification is similar to LSTM. We ran a GRU with 128 neurons, this time using a combination of relu and sigmoid activation functions, and obtained an accuracy of on AICrowd.

Combination of both

Finally, we can try combining both LSTM and GRU to complexify our Neural Network. We use 100 neurons for the LSTM layers and 128 neurons for the GRU layers, and sigmoid for the activation function. This yields an accuracy of 0.786 and a F1-Score of 0.797 on AICrowd, which doesn't seem very accurate but was run for only 2 epochs. Running Recurrent Neural Networks for more should yield better results, but takes time and resources.

Results



Neural Network	Embedding	Accuracy	F1-Score
LSTM	GloVe	0.819	0.826
GRU	GloVe	0.809	0.815
LSTM + GRU (2 epochs)	GloVe	0.786	0.797

B. *Conclusion*

The rise of deep learning and the release of complex models pre-trained on large corpora (BERT and every fine-tuning and iteration that followed) shifted the landscape towards less manual tweaking and more ’freedom’ to the model in learning the best representation of text (embeddings) based only on the final result. In this project we explored various NLP techniques, and we decided to emphasize how some older, simpler pipelines can reach competitive results with a fraction of the costs. This somewhat diverges from the general trends that is going towards ’black-box’ models that can be called in a single line of code and are trained by state-of-the-art hardware for long periods of time. We saw how the problem of sparse matrices that quickly discards vectorizers lik BOW and TF-IDF is handled elegantly by standard libraries like Numpy, leading to fast, interpretable and competitive results.

C. *References*

We used a variety of references, Scientific Papers, Practical examples in the documentation of the libraries, CS-433 lectures and Videos.

- Comparing Methods of Text Categorization
- Simple Noun Phrase Extraction for Text Analysis
- Analysis of Twitter Specific Preprocessing Technique for Tweets
- BERTweet: A pre-trained language model for English Tweets