Machine Learning Project 2
Text Classification

Franck Dessimoz, Paul Jeha, Pierre Latécoère
CS-433 Machine Learning, EPFL, Switzerland

***Abstract*—Text classification is a core topic of Machine Learning. It has a broad range of applications such as sentiment analysis of social media posts, email spam detection or text classification with respect to specific sentiments.**

I. Introduction

In this paper we are going to investigate different techniques to classify tweets. Our goal is, given a set of tweets, to train a model to predict whether a tweet reflects a positive or a negative emotion. The data we are given are tweets which originally ended with either a ":)" smiley for positive tweets or a ":(" smiley for negative ones.

II. Methods

A critical point in machine learning applications, and thus in text classification, is the preprocessing step. In our case, we must map sequence of words to their vector representation, without losing the essence of the information.

The preprocessing intervenes at two different steps: firstly on the text by working on the words themselves, and secondly on the vectorization process, i.e. how to map those sequences and words to their vector representation. Our vectors must capture the relations between the words and thus reflects the meaning of the words or sentences.

A. First method - Preprocessing on words

The first preprocessing step on words consists of getting all the different chain of characters existing in the tweets. Then we only kept the ones that appeared at least five times to create a vocabulary. This allows us to get rid of the words that will bring no information as they don't appear frequently enough, and will generate useless computation.

B. Second method - From tweets to vectors using GloVe

In this method we map each word to a vector using the GloVe algorithm provided with the project. It creates a vector with an embedding dimension of $N = 20$ for each word. It gives us the vector representation of each word.
Now that we have the representation of each word, we need to obtain the vector representation of each tweet. In this method we create the tweet vector by averaging the words vector of the words present in the tweet.

C. Third method - Words' weight with TF-IDF

In this method we improved our strategy in constructing the vector representation of a tweet. Indeed the previous method did not take into account the weight of a word. Of course all words in a tweet do not provide the same amount of information and thus its weight is of high importance. As an example words such as 'a', 'the' or 'this' do not provide the same amount of information as 'happy' or 'sad' as they occur very often irrespective of the document. Therefore they should not be treated as having the same importance. Hence we want now to attribute weights to words that reflects their importance. To do so we use the $TfidfVectorizer$ function of the Scikit learn library.

The $TfidfVectorizer$ function converts a collection of raw documents (a document represents a tweet in our case) to a TF-IDF matrix. Each document is directly embedded in a vector, meaning that we do not do a word embedding first.

In a TF-IDF matrix each document is represented by a row, while the columns correspond to the features. Each row is built taking into account the importance of the words. The metric to measure the importance of a word is the frequency of this word in a given document. The more frequent the more important. However if a word appears often in other documents, its importance is reduced as he brings less relevant information.

D. Fourth method - Customized tokenizer and stopwords

Taking into account the weight of the words is good idea but is not sufficient in our case. We must now tune two main parameters of $TfidfVectorizer$, the tokenizer and the stop words list.

Firstly, we customized the stop word list. Indeed till now we used the default stop words list of $TfidfVectorizer$, which contains the stop words of the english language. However this list is not specific to the context of tweets, since they do not follow the usual convention of the english grammar and vocabulary. Therefore we must use a more appropriate stop words list. We found three lists[1] of words specifically designed for twitter text analysis, that we combined in order to obtain our final list of stop words.

Secondly, we tuned the tokenizer parameter of the $TfidfVectorizer$. The tokenizer is the function that split a sentence into a list of words. $TfidfVectorizer$ has a default tokenizer but once again this default tokenizer doesn't take

[1]https://sites.google.com/site/iamgongwei/home/sw

into account the fact that we are in a social media context. To handle this issue we defined our own tokenizer, specifically designed for our needs. We make use of $TweetTokenizer()$, a tweet-aware tokenizer, i.e. a function that splits tweets into lists of words taking into consideration that we are working with tweets. For example, smileys as ':-)' or words containing a hashtag '#friend' won't be split, while the default tokenizer would have returned [':', '-', ')'] and ['#', 'friend']. After that, we applied a lemmatizer function to the tokens. The lemmatizing step is the lexical analysis, which groups words of the same family. This means that each word will be taken back to its root. It groups the different forms a word can take: a name, a plural a verb or an infinitive. For example, 'taking' will be taken back to 'take', 'dogs' will be taken back to 'dog'.

E. Fifth method: Tuning the hyperparameters of the vectorizer

The main issue of our previous method is that the $TfidfVectorizer$ created too many features. Indeed in the small dataset, the number of features is roughly 100'000 while in the full dataset, the number of features is around 600'000. To decrease the number of features we can tune the threshold parameter *min_df* of $TfidfVectorizer$, which removes the words which number of appearance is less than the threshold. Below is a table showing the numbers of features remaining for different values of *min_df*.

TABLE I
NUMBER OF FEATURE AS A FUNCTION OF MIN_DF

	0	5	10	50
Small dataset	100'000	20'000	10'000	3'500
Full dataset	600'000	90'000	50'000	20'000

To find the optimal parameter, we computed the accuracy obtained for different values of *min_df*:

TABLE II
TUNED TFIDFVECTORIZER - MIN_DF

	5	10	50
LogisticRegression	82.66%	82.77%	82.34%
SGDClassifier	81.60%	81.75%	81.66%
LinearSVC	82.20%	82.58%	82.13%

We see that setting *min_df* to 10 is the best compromise between the number of features and the accuracy.

The last parameter we considered is *ngram_range*. This parameter represents the size of the window sliding over each word. Indeed, $TfidfVectorizer$ can, if stated through this parameter, consider the context of each word. This context is represented by the window. Consider the following example : "Big Black Cat" with *ngram_range=2*. Then $TfidfVectorizer$ will consider : {*Big; Black; Cat; Big Black; Black Cat*}. Now instead of having the information of one-word context, we gain the information provided by tuples of words.

Setting *min_df* to 10, we have the following results for different values of the upper bound of *ngram_range*:

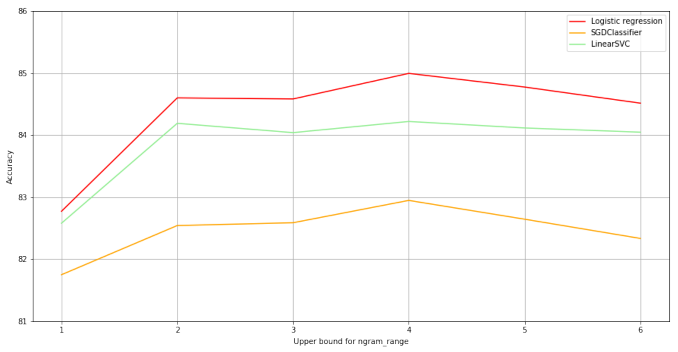

Fig. 1. Accuracy as a function of $ngram\_range$

III. MODELS AND RESULTS

In this study we used three different classifiers: $LogisticRegression$, $SGDClassifier$ and $LinearSVC$. $LogisticRegression$ is a standard model and is the simplest model that we tried, but not the least accurate. $SGDClassifier$ is a classification model that is often used in the context of text classification, while dealing with large datasets. $LinearSVC$ is a less computationally intensive model than SVC, the support vector machine classifier implemented in the Scikit learn library.

Note that we were given two data sets: the full one (2.5 millions samples) and a smaller one (200k samples). When stated explicitly (when our hardware allowed it) we ran our classifiers on the full dataset, otherwise we used the smaller dataset.

Also the sentiment associated with each tweet is a number encoded as 1 for a positive sentiment and 0 for a negative one[2].
To compute the accuracies of our models, we split our dataset into a train set, always according to the following proportions: train set 80% and test set 20%. Note that this has been done for all models in order to compute the accuracy locally.

A. First model

In this first model we used the preprocessing methods for words and tweets to vectors mapping described in sections II-A and II-B. Thus, our training set consists of vectors of length 20, each representing a tweet.

The following table contains the results of running $LogisticRegression$, $SGDClassifier$ and $LinearSVC$ without polynomial degree expansion and with polynomial degree expansions of degree 2 and 3:

[2]When submitting on CrowdAi, the negative tweets are encoded as -1 but most of the classifiers we used ask for prediction values to be binary.

TABLE III
COMPARISON OF CLASSIFIERS

	d = 1	d = 2	d = 3
LogisticRegression	61.43%	63.75%	64.91%
SGDClassifier	61.42%	63.08%	64.26%
LinearSVC	61.49%	61.71%	64.53%

By observing the above tables, we can see that the accuracy of our model is bounded by 65%. This means that we have to improve the preprocessing step in order to get a better accuracy.

B. Second model

The previous model highlight the issue we stated in the previous section, i.e the words can not be considered as having the same weight. In this model we start from the raw dataset of tweets. We process them through the second method of preprocessing, $TfidfVectorizer$, presented in section II-C. Thus our training set is a matrix where each row consists of the embedding of a tweet.
We run the three same classifiers to obtain the following accuracies:

TABLE IV
BASIC TFIDFVECTORIZER

	Small dataset	Full dataset
LogisticRegression	80.81%	82.20%
SGDClassifier	78.98%	78.80%
LinearSVC	80.56%	82.10%

We clearly see an improvement of almost 20% in the accuracies when taking into account the weight of each word.

Note that the $TfidfVectorizer$ returns a matrix, whose rows correspond to the vector representation of the tweets. This matrix is sparse, which prevents us to make many transformations on it but allows the computation to be much faster.

C. Third model

For this model we used the third method of preprocessing, $TfidfVectorizer$ with tuned tokenizer and stopwords, described in section II-D. Again our training set is a matrix where each row embeds a tweet. We used the same three classifiers to obtain the following accuracies:

TABLE V
TFIDFVECTORIZER WITH TUNED TOKENIZER AND STOPWORDS

	small dataset	full dataset
LogisticRegression	82.61%	83.72%
SGDClassifier	81.49%	81.26%
LinearSVC	82.45%	83.55%

With this third model, we see an improvement of 2% compared to the previous one.

D. Fourth model

In this model, we used the preprocessing method of section II-E, which is $TfidfVectorizer$ with tuned hyperparameters.

We used the parameter values for $TfidfVectorizer$ which gave the best accuracies: *ngram_range = (1,4)* and *min_df = 10*. See Table II and Fig. 1.

The accuracy results produced by the three classifiers are the following:

TABLE VI
FINAL TFIDFVECTORIZER

	Small dataset	Full dataset
LogisticRegression	84.80%	87.16%
SGDClassifier	81.75%	82.54%
LinearSVC	82.58%	85.09%

Again we can see a improvement of about 2% compared to the model with the default *ngram_range* and *min_df* parameters.

E. Fifth model

In the previous models we focused on tuning the parameters of the vectorizer, and test accuracies using 3 classifiers.
Now we will focus on finding the best classifier and finding its optimal parameters. We decided to focus only on $LinearSVC$. Indeed after some research [3], we found out that support vector machine classifier is considered one of the best classifier for text classification. The Scikit learn library provides many support vector machine classifiers. Among them, we kept two: SVC ans $LinearSVC$. $LinearSVC$ is a faster classifier for SVC with the kernel parameter set to be linear.
Then we focused on finding the optimal parameters for $LinearSVC$. To do so, we ran a 5-fold cross-validation grid search implemented in the Scikit learn library, $GridSearchCV$. We defined the grid parameters to be the following:

losses = ['hinge', 'squared_hinge']
tols = [10^{-5}, 10^{-4}, 10^{-3}]
Cs = [0.1, 1, 10]

We restricted ourselves to a few parameters, and applied $GridSearchCV$ on the small dataset in order to keep the computation runnable on our computers.

The grid search results in the following optimal parameters: losses = 'hinge', tols = 10^{-5}, Cs = 1

Applying the optimal parameters to $LinearSVC$, we obtain the following accuracies, which are the final ones:

[3]https://towardsdatascience.com/multi-class-text-classification-model-comparison-and-selection-5eb066197568

TABLE VII
LINEARSVC WITH OPTIMIZED PARAMETERS

	Small dataset	Full dataset	CrowdAI
LinearSVC	84.89%	87.73%	87.1%

We were surprised to see that running two times $LinearSVC$ on the same input could lead to different outputs, event with the $random\_state$ parameters set to the same value. After investigating, we found out that the underlying C implementation of $LinearSVC$ uses its own random number generator to select features when fitting the model. Thus, it is not uncommon to have slightly different results for the same input data[4]. Therefore, our final accuracy vary between 86.6% and 87.1%.

IV. DISCUSSION

Let's discuss about what are the next steps that could be taken in order to increase even more the predictions accuracy. First of all, we quickly noticed that our computers are not able to run more complex models than the ones we used. Indeed, as described above, we used $LinearSVC$ as the final classifier since it is a quicker implementation of SVC with kernel parameter set to be linear. Not surprisingly, a faster convergence implies a loss of accuracy. This means that if we were able to run the SVC classifier, we would certainly have obtained a better accuracy. To illustrate this statement, we plotted the accuracies of the 4 classifiers - $LogisticRegression$, $SGDClassifier$, $LinearSVC$ and SVC - as a function of the number of samples, up to 100000, which is the maximum number of samples we can use to be able to execute SVC on our machines.

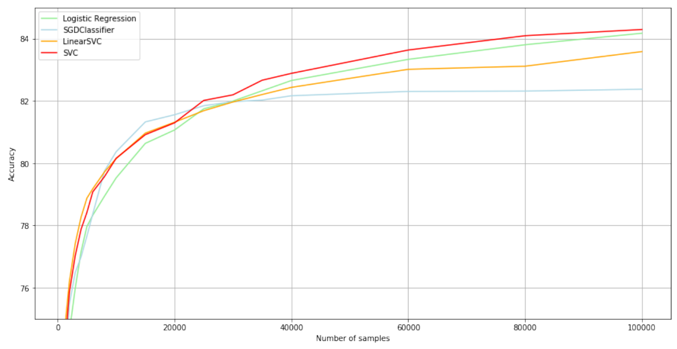

Fig. 2. Accuracy as a function of the number of samples

We can see that SVC classifier performs slightly better the other ones. Note that we ran the four classifiers with their standard parameters. Therefore, by tuning the parameters of SVC classifier the same way we did for $LinearSVC$ (using $GridSearchCV$), we would have achieved an higher score.

[4]https://kite.com/python/docs/sklearn.svm.classes.linearsvc

Secondly, the same issue as the one described above appears when we apply the grid search algorithm to find the classifier's optimal parameters. We are limited in the number of parameters to test by the computation complexity of the classifiers. To solve this problem, it would be helpful to run the scripts on a cluster.

Finally, there exists many more ways to transform words or tweets into vectors, tune the stop words, customize the way of tokenizing text or classify data using machine learning libraries. This leads to millions of parameters to tune and it certainly exists a better combinations that the ones we presented in this paper, even thought we think that $TfidfVectorizer$ is one of the most prevalent text vectorizer.

V. CONCLUSION

As final results, we achieved a 87.1% accuracy using a $LinearSVC$ classifier, a TF-IDF tweet embedding and some preprocessing and parameters tuning.

Now, let's state all what this project has taught us.
Firstly, to become acquainted with a practical text classification problem. We learned different techniques of Natural Language Processing that drastically improved the accuracy of our results. We had to understand the dataset by analyzing and manipulating it to be able to get a better intuition of what were the next steps to take.

Having to work with a large dataset, we experienced a common problem in machine learning projects: models are very limited by the computational power of hardware.

Finally, we can clearly say that text classification is a very interesting research area and we really enjoyed working on this project.