Machine Learning: Project 2
Text classification

Benjamin Pedrotti - benjamin.pedrotti@epfl.ch
Damien Ronssin - damien.ronssin@epfl.ch
Jérémie Canoni–Meynet - jeremie.canoni-meynet@epfl.ch

***Abstract*—For the second Machine Learning (CS-433, EPFL) [1] project, we decided to work on a binary text classification problem based on a twitter dataset. The main goal of this report is to elaborate a model predicting the overall sentiment in a tweet. Throughout this report, one can see the different steps needed to build a model enabling us to obtain an accuracy of 86.6%. This accuracy was obtained using a convolutional neural network architecture.**

I. Introduction

The goal of this project is to predict the overall sentiment contained in a tweet using machine learning algorithms. The dataset contains more than 2 millions raw tweets which included two types of smileys, either :) or :(. These smileys carried respectively a positive and a negative sentiment. They were removed and now serve as labels for the overall sentiment of the tweet. Two main approaches have been used here, one based on ”classical” machine learning, and a second one focusing on deep learning. First, let’s start with the pre-processing of the dataset.

II. Explanatory data analysis and data pre-processing

A. Processing the different types of features

We tried different preprocessing strategies on this dataset. Here we will describe each step we tried and we will discuss their effect on the performance later in this report.

We first started by removing all the non-unique tweets in order to avoid a bias in the dataset. Then, we proceeded with the following steps.

a) Punctuation and number removal: All the numbers, comas, question marks and final points were removed from the dataset. The hashtag symbols were also removed but their contents were kept. This choice have been made because we considered that these symbols did not carry relevant information.

b) Spelling correction: The dataset comes from Twitter and is therefore prone to a lot of spelling mistakes. This can lead to a different embedding for the same word. To overcome this problem, we used the **SymSpell** library [2] to correct spelling mistakes. This library also performs word segmentation, it is notably very useful to get back the multiple words composing a hashtag.

c) Stopwords: A lot of common words do not carry much information. This is the case for instance for all the subjects, prepositions, etc. Hence, in order to reduce the dataset’s dimensionality, we removed a custom list of stopwords. The models have shown to be very sensitive to the removal of stopwords. Indeed, the trade-off between the gain in dimensionality reduction and the loss of information needs to be carefully set.

d) Lemmatizing: Suffixes do not carry a lot of information. It appears natural to produce the same embedding for words with the same root. The **nltk** library [3] have been used to produce these transformations, e.g.

$$\text{learning} \rightarrow \text{learn}, \quad \text{algorithms} \rightarrow \text{algorithm}.$$

For both spelling correction and lemmatization algorithms, some unwanted artifacts can appear. For example, the expression ’lol’ is corrected as ’low’. We tried to avoid this as much as possible by adding some words in the SymSpell dictionary. Still, some particular words will be corrected or lemmatized in a wrong way. However, this is not so important since all the expressions will be corrected or lemmatized the same way. They will therefore be treated as the same token by the classification algorithm.

e) Negation processing: The dataset contains several negative words, e.g. ”won’t”, ”don’t”, etc. In order to highlight the negation we replaced all the words ending with ”n’t” by ”not”. Hence, the words ”won’t” and ”don’t” become ”not”. We also decided to apply this process to a specific list of tokens (e.g. ”wont”, ”wasnt”, etc.).

III. Classical machine learning approach

A. Encoding - Theoretical explanation

As stated before, one of the main challenge of NLP is to encode the information. For doing this, we firstly used the so called **Term Frequency - Inverse Document Frequency** (TF-IDF) matrix approach. In order to define how this works, let us take a step back and return to our initial setup. In our dataset, we possess tweets, and each of them contains several ”words”. To obtain the TF-IDF matrix, we first compute the **Bag-Of-Words** (BOW) matrix. For each tweet, we compute the BOW, which is simply a dictionary of each word contained in a tweet and the number of time they appear in this tweet. Then, we transformed this dictionary in a high-dimensional (and very sparse) vector.

For instance, if the tweet is ”I love apples and I love ML”, then its respective bag of words is {I: 2, and:1, love:2, ML:1, apples:1}, and its feature vector is $[0, \ldots 1, 0, \ldots 2, \ldots 1, 0, \ldots 2, \ldots 1]$. The ”0” corresponds to words that do not appear in this tweet, but do appear in other tweets.
To create the BOW matrix, we just concatenate row-wisely these feature vectors. The obtained matrix has N rows, where N represents the number of tweets, and K columns, where K represents the number of different words in the whole set. With this information, we can now compute the **Inverse Document Frequency** (IDF) of each words, which is defined as: $idf(w) = -\log(f_{tweet}(w)/N)$, where w is a word, $f_{tweet}(w)$ is the number of tweets that contain the word w, and N defined as before.
Finally, by defining the term frequency of word w in tweet t as $tf(w, t)$, we obtained the TF-IDF matrix X such that $X_{i,j} = tw(i, j) \times idf(i)$, where $i = 1, \ldots, N$ is the tweet number, and $j = 1, \ldots, K$ is the word number. We now have our feature matrix X! Note that for this part, the encoding is done directly on the original dataset as using a pre-process dataset here decreased the accuracy.

B. *Encoding - Practical approach*

For the practical implementation of the TF-IDF encoding, we used the package **scikit-learn**. From the module `feature_extraction.text`, we used the function `CountVectorizer`, which transforms the text into the BOW matrix, and the function `TfidfTransformer`, which transforms the BOW into its TF-IDF matrix representation. Finally, we used the function `make_pipeline`, which enabled us to re-apply the same transformation on the training and testing set.
During the encoding part, we had optimized several hyperparameters while doing the `CountVectorizer`. Namely, we optimize, via cross-validation, the parameters $max\_features$ and $ngram\_range$. We did this using the function called `hyperparameter_tuning` in our code, which enables us to tune hyperparameters on the encoding and on any machine learning models that we will use after. In a nutshell, we used cross validation in this function, which notably takes as arguments a list of kwargs. Typically, this latter will be the list of different hyperparemeters to optimize.

C. *Hyperparameters tuning and model selection*

Now that we have our feature matrix X, the rest of the problem is a classical machine learning problem: we need to decide which model we want to use and to tune their respective hyperparemeters. This is done in the Jupyter Notebook **classical_ml**. In this code, we used the following approach: first, we optimize the parameters for each models, using the same function as before (`hyperparameter_tuning`). And then, we compare each model train with their respective best parameters (note that due to computational constraint, this part has been done using 10% of the dataset). Here is the list of the models used with their hyperparameters:

- **Logistic Regression** (LR): try different solver and penalty function.
- **Random Forest** (RF): optimize $n\_estimators$ (number of trees), best is $n\_estimators = 108$.
- **Naïve Bayes** (NB): optimize the smoothing parameter $alpha$, best is $alpha = 0.88$.
- **K-Nearest Neighbors** (KNN): optimize the numbers of neighbors to consider, best is $n\_neighbors = 5$.
- **Support Vector Machines** (SVM): optimize the regularization parameter C, best is $C = 0.78$

Note that due to computational constraint, we couldn’t run the SVM classifier on the all dataset, and thus, it won’t be present in the rest of our analysis. KNN also gives us very bad results which is due to the curse of dimensionality as seen in class.

The figure obtained for the comparison of all the models can be seen on figure 1 (note that as we are ranked by accuracy on AIcrowd, we used the accuracy as a proxy for the performance. Note also that as KNN gave us a very low accuracy, we didn’t put it on the figure for scaling purposes). As we can see in the

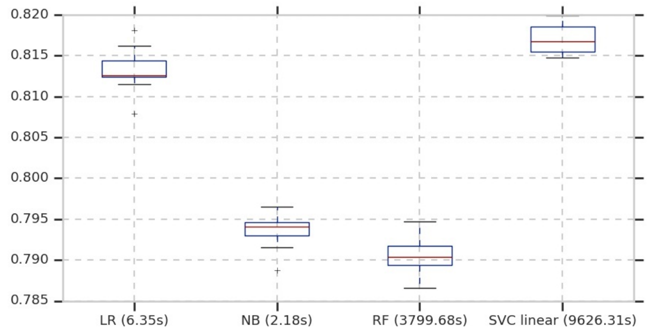

Fig. 1: Accuracy comparison between the different models

figure 1, it seems that SVM gives the best results. While the difference between SVM and LR is very small the time needed to fit the two models is very different. Indeed, it takes more that 1000 times longer to train SVM than LR. Consequently and due to the fact that we couldn’t run SVM on the whole dataset, we will keep LR as a best model.

We also plotted the **receiver operating characteristic** (ROC) curve on figure 2. Note that the ROC curve 2 confort us in our choice of LR for being the best classifier to use.

D. *Conclusion of the classical machine Learning*

To conclude with the classical machine learning part, we have seen that one challenging aspect of NLP is how do we extract the information from the text, and transform it to a classical feature matrix X. We did this using the TF-IDF matrix approach. After that, we are faced with a classical classification problem. We firstly optimized some hyperparameters among a set of chosen models, and then compared the different models using cross-validation. For this problem, the best model is LR. Not only it gave us the best accuracy, but it is also one of the fastest model. With this model, we were able to make a submission on AIcrowd (ID:28202), and achieve an accuracy of 84.8%, which is quite astonishing regarding the simplicity of the model!

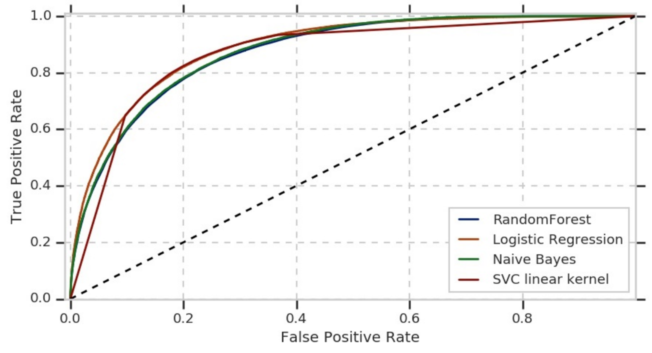

Fig. 2: ROC curves

IV. Deep learning approach

An approach based on deep learning algorithms is presented here.

A. *Word embedding*

As for the classical machine learning part, representing words in a vector space is required to train algorithms. Two models have been considered to proceed word embedding.

a) ***Word2Vec****:* The first model is the so-called Word2Vec model with a Skip-gram model architecture. The aim of the Skip-gram model is to, given a sequence of T words $w_1, \ldots, w_T$, maximize an approximation of the following average log probability:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c,\ j\neq 0}\log p(w_{t+j}|w_t), \tag{1}$$

where c is the surrounding window of the word w_t [4]. Therefore, it gives a representation that takes into account the spaciality in the sequence and predicts with a certain accuracy the surrounding words in this sequence. In this project, we used the negative sampling method to approximate equation (1). Word2Vec was used thanks to the **gensim** library implementation [5].

b) ***GloVe****:* The second model is GloVe, a variant of Word2Vec. While Word2Vec is based on a neural network architecture, GloVe finds a matrix factorization of the word-dataset co-occurence matrix. Then, an update is done on the embedding matrix by minimizing a specific loss (in this case the difference between the products of word embeddings and the log of the probability of co-occurrences) [6].

B. *Architectures*

Two main architectures have been considered for deep neural networks. In each case, the embedding is considered as a layer of the network.

a) ***Long-Short Term Memory*** *(LSTM [7]):* LSTM is a specific type of Recurrent Neural Network (RNN). Using LSTM models can be useful for processing sequences as they are able to learn long-term dependencies. Indeed, LSTM is composed of different mechanisms such as gates, hidden state and cell state. The role of the different gates is to regulate the flow of information before encoding it into the states. The states, which are updated at each iteration over the processed sequence, allow to keep in memory selected information throughout the processing. In other words, they will act as the memory of the network. We see that this architecture can be helpful for our problem as it would be able to build a relation between the words in a tweet (which is a word sequence) [8]. In our architecture, we used a single layer of LSTM as follows:

- GloVe embedding
- LSTM layer
- Fully connected layer (to 1 neuron) with sigmoid as activation function.

b) ***Convolutional Neural Network*** *(CNN):* The structure of our CNN is the following:

- Word2Vec embedding (skipgram)
- 1D convolution + GlobalMaxPooling (takes maximum of each channel)
- Fully Connected + dropout + ReLU
- Fully Connected (to 1 neuron) + Sigmoid

This 1D convolutional structure is well known in binary sentiment text classification and yields good results on different datasets [9]. The binary crossentropy loss was chosen as it is designed for binary classification. Also, the Adam optimizer was used [10] with its default learning rate and other parameters in Keras [11].

Concerning the training of the embedding layer, we explored two possibilities. For the first one, we trained the Word2Vec model once with gensim and then kept constant the embedding's parameters. For the second one, we continued the training of embedding's parameters during the training of the full convolutionnal network. For computational limitation reasons, the hyperparameters tuning was done with the embedding parameters fixed. A comparison between these two possibilities will be done later in this report using the same network structure.

C. *Hyper-parameters tuning*

We decided to tune only certain parameters because of the important computation time it requires. A choice have been made to set specific parameters to default values known to be reasonable and working well in most cases. For instance, all our deep learning algorithms use the Adam optimizer, which is an adaptive learning rate method. Therefore, we didn't tune this parameter.

Also for computational reasons, all the parameters were optimized via kfold cross-validation on 10% of the dataset, with $k_{fold} = 5$.

a) ***Long-Short Term Memory****:* We used a pre-trained GloVe model that has been trained on a dataset containing 2 billions tweets. This allows to have a great variety of words and increase our chance to have a corresponding embedding for any word in our dataset. Out-of-vocabulary words are just ignored when producing the co-occurrence count. There are different embedding dimensions available for this pre-trained

GloVe model : $n_{dim} = 25, 50, 100, 200$. For this architecture, the following parameters have been tuned:

- Embedding dimension ($n_{dim} = 200$)
- Output dimension ($d_{out} = 400$)
- Batch size used for training ($b_s = 64$)
- Number of epochs ($e = 4$)

As previously, we performed the hyper-parameter tuning using k-fold cross validation and $k_{fold} = 5$. Concerning the pre-processing, the best accuracy for this model have been obtained with a particular combination of pre-processing methods listed in Section.II. We obtained 86.2% on Aicrowd with the following pre-processing : Punctuation and number removal + Spelling correction + Negation processing.

*b) **Convolutional Neural Network:*** For training Word2Vec model, we chose a window size of 5 words, a number of iterations of 5 and a 6 as the minimum appearance threshold for a word to be considered. The kernel size of the convolutional layer was set to 5 just like the window size of Word2Vec. The dropout probability was set to 0.2.

The different hyper-parameters optimized and their respective results are the following:

- Embedding dimension ($n_{dim} = 400$)
- Number of filters of the convolutional layer ($n_f = 300$)
- Dimension of the first fully connected layer ($FC_d = 250$)
- Batch size ($b = 150$)

The best number of epochs depends on the dataset used and if we keep training the embedding with the network and will be detailed in next section.

D. Results

In figure 3, one can see the results of the deep learning models on 10% of the dataset. Let us now explicit the legend keywords: 'raw' means that the dataset used to train the model is almost unprocessed (punctuation and numbers removal), 'p-pr' means that the dataset has been partially processed (raw processing + spelling correction + negation processing), 'pr' means that the dataset has been processed: (raw processing + spelling correction + lemmatization + removal of stop words). Finally 'tr_e' means that the embedding layer has been trained also during the network training.

The best result in Fig 3 ($82,37\%$) was achieved by a convolutional neural network on 10% of the raw dataset, with embedding trained also during the network training with 2 epochs. On the full dataset this configuration gave us a score on Aicrowd of 86.6% (ID:31579). Note that due to stochastic processes, the accuracy may differ from the one we obtained. To overcome this we fixed as much seeds as possible but it may still remains some uncertainties.

V. Conclusion

To conclude with, on 10% of the dataset, both classical ML and deep learning algorithms gave us similar accuracies around $80\% - 82\%$. When the full dataset was used, we observed an important increase of the accuracies of all models. However one cannot guarantee that our hyper-parameter tuning is optimal for the full dataset, thus it might be possible to increase the performances by performing the tuning on the full dataset. As said before, this would be a really computationally demanding step.

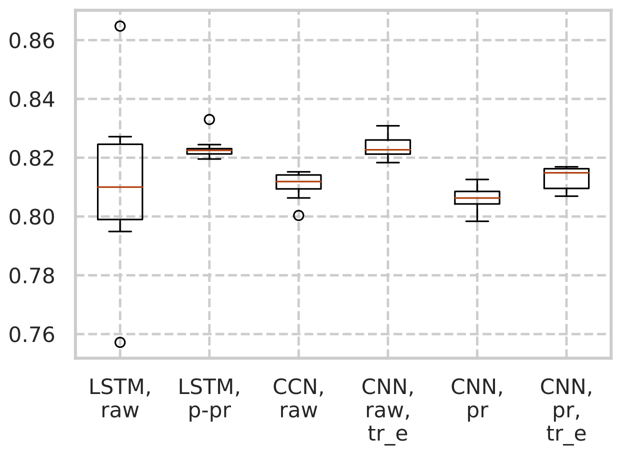

Fig. 3: Accuracy of the deep learning models (kfold crossvalidation with k=10)

We also observed the important influence of the pre-processing step and some of its counter-intuitive aspects. For example, lemmatizing the tweets resulted in worse performances for different models.

The best score was obtained with a deep learning architecture. These types of architectures allow to obtain very good results. However, they are quite demanding in terms of computational resources for the training part. In contrast, we saw that classical machine learning algorithms, such as the logistic regression for instance, yields also good results in a very short period of time.

References

[1] M. Jaggi and R. Urbanke, "Machine learning-cs-433," 2019.
[2] W. Garbe, "Symspell," 2019.
[3] E. Loper and S. Bird, "Nltk: the natural language toolkit," *arXiv preprint cs/0205028*, 2002.
[4] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.
[5] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Valletta, Malta: ELRA, May 2010, pp. 45–50, http://is.muni.cz/publication/884893/en.
[6] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.
[7] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[8] C. Zhou, C. Sun, Z. Liu, and F. Lau, "A c-lstm neural network for text classification," *arXiv preprint arXiv:1511.08630*, 2015.
[9] H. Kim and Y.-S. Jeong, "Sentiment classification using convolutional neural networks," *Applied Sciences*, vol. 9, no. 11, p. 2347, 2019.
[10] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.
[11] F. Chollet *et al.*, "Keras," https://keras.io, 2015.