Text classification for sentiment analysis

Alexis Dewaele, Robin Leurent, Alexis Mermet

***Abstract*— The analysis of sentiments in texts is proven to be challenging for computers, especially since it can depend on the kind of text one wishes to analyze. In this project, we classified tweets based on the sentiment expressed in them through the use of Machine Learning. After the rigorous pre-processing of the data, we implemented several kinds of models including different sorts of neural networks. Finally, the best result was obtained with the use of a Recurrent Neural Network.**

INTRODUCTION

Sentiment analysis is common and often easy to achieve as humans. The difficulty for computers to understand the sentiments behind given sentences lies in the fact that words have different meanings depending on the context. Words occurring together can have the opposite meaning of these words taken separately. Machine Learning is the answer to this. We will first go through the data pre-processing we did, then we'll explain how the different models we tried work, and then we'll describe our results.

I. Data Processing

Text processing and in this particular case, tweet processing can be quite complicated. Indeed, tweets are messy: people use homonyms, slang, abbreviations or even extend words to convey deeper meaning, e.g. *"I loooved this movie!"*. Therefore, we must do some heavy pre-processing before feeding our data into our model. We used several techniques but not all of them were kept in our final work.

A. Unused data processing

- **Stop-words Removal**
  Using the Natural Language Toolkit (NLTK) data-set [1], we removed all the stop-words in the tweets since in theory they appear extremely often in the English language but they don't bring any meaning to the sentence. Example of stop-words: *a* or *the*.
- **Slang**
  The twitter language is full of slang words having the same meaning as an existing English word. We tried to convert those words into the real English ones using a slang dictionary [2]. Unfortunately this did not work well since slang words often have several meanings, and some slang words exist in the English dictionary as well but with completely different signification. For example, *blood* can also mean *close friend* as shown in [2].
- **Lemmatizing**
  To lemmatize a word means to reduce it to its root form, e.g. the word *fishing* would be transformed in *fish*. We used the NLTK Lemmatizer [1] to lemmatize all the words in our tweets.

We discovered that these techniques work well in theory and in small data sets, but can lead to malfunction for large data sets. Thus we decided to drop these features and to implement the following ones.

B. Data processing used

- **Punctuation and numbers**
  The punctuation in a sentence doesn't bring any information. We remove it, as well as all the single numbers. We also remove the beacons $< user >$ and $< url >$.
- **Sets of words**
  Instead of lemmatizing each word in order to reduce the number of words in the vocabulary, we decided to create sets of words having a close meaning. For example we grouped in a single set $laugh = \{lol, laugh, haha, funny, fun, lmao, laughing, ...\}$. We created sets for the sentiments *bad, angry, love, happy* and *good*. To decide if a word belongs to the set good or the set bad, we created a function that retrieves the 1000 words that occur the most respectively in the positive and the negative training sets. Then, we compared the occurrences of these words in the opposite set, and if the ratio was high enough (70%), we placed them in the corresponding set.

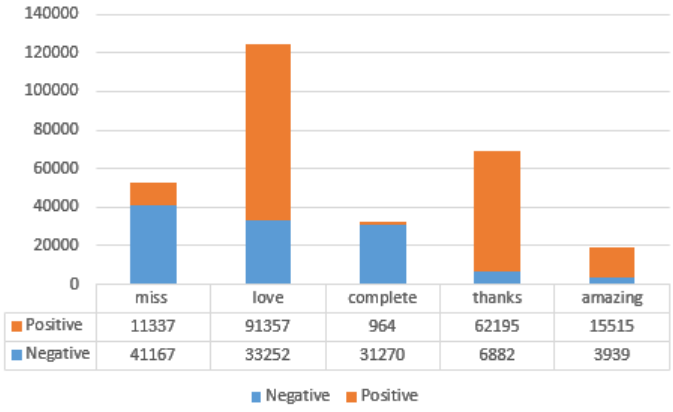

Fig. 1. Histogram of some high occurring words and their distribution.

For example, as we can see on Figure 1, we classified *miss* and *complete* as bad words, and *love, thanks, amazing* as good words. Then, we replaced all the occurrences of these words by the referent word of their set. This allows to reduce the vocabulary size, and to emphasize the meaning of critical words.

- **Filter out rare words**
  Having extremely specific words that appear once or

twice in one million tweets does not give a relevant information for a machine learning model. The probability that this word will help to predict a new tweet is close to zero, and it increases the vocabulary size. Hence we decided to drop the words occurring less than 5 times in the joint training sets.

II. Models

We used the library Keras [7] to implement all of our neural networks. For most of the hyper-parameters stated in this section, we used a simple grid search on the small data-set in order to find the best parameters.

A. Logistic Regression with GloVe solution

First, we used the solution given in the assignment to do the matrix factorization of the co-occurrence matrix. Now this matrix has the form $X = WZ^T$. We can use the matrix W to recover the representation of any given word in the vocabulary (W[0] contains the most common words of the vocabulary, etc...). When we have found the embedding of each word in a tweet, we find the mean vector representing this tweet: $\frac{1}{n}\sum_{w \in tweet} w_{rep}$.Note that if the tweets doesn't contain any word in the vocabulary, its representation is the global mean vector of the embedding matrix. Then we plug the newly obtained data into a logistic regression to obtain the classification into the positive and negative classes. This regularized logistic regression comes from the Python library Scikit-learn [4]. The aim of penalized logistic regression is to find W such that it minimizes:

$$\underset{w}{\operatorname{argmin}} \sum_{n=1}^{N} ln(1+\exp\left(\mathbf{x_n}^T\mathbf{w}\right)) - y_n \mathbf{x_n}^T\mathbf{w} + \lambda \mid\mid \mathbf{w} \mid\mid^2 \quad (1)$$

The parameters we used are a tolerance of 0.001 , a regularization strength of 0.05 and a solver newton-cg.

B. Logistic Regression with Word2Vec

Having a look at the code of the GloVe solution provided, we realized that it was simply doing a scaling without any regularization. At this time, we had not seen the matrix factorization in the lectures, hence we decided to create the embedding matrix using Word2Vec. We used the version from Gensim [3]. This framework implements Word2Vec family of algorithms, using highly optimized C routines, data streaming and Pythonic interfaces. In our case, we used the skip-gram implementation. This method uses a sliding window around the input word, and outputs a vector that represents this word. Basically, multiplying two of these vectors together computes the probability x that given the word_1, taking a random word in its entourage has probability x to be the word_2. Two words having similar contexts will have a very close representation. We applied Word2Vec on the preprocessed training sets to obtain each word representation. Then we plug the data into the logistic regression as before (II-A) and obtain the classification of our tweets. Here again we used a tolerance of 0.001 , a regularization strength of 0.05 and a solver newton-cg.

C. Convolutional Neural Network (CNN)

A logistic regression model is a linear model. Hence, it cannot find non linear relationships between the inputs. But a convolutional neural network (CNN) has the ability with its deep network to learn more complex non linear functions between the inputs. Since we have a substantial data-set, we implemented and trained a CNN. We tried two different embedding algorithms.

First, we used Word2vec as in section (II-B). As we will see in part III, our results was quite low with it.

We hence decided to implement the matrix factorization using SGD algorithm as we saw during the lecture since it is supposed to be one of the best. Using the given scripts, we create the co-occurrence matrix X as in II-A. Then we use a stochastic gradient descent to obtain a new representation of X: $X = WZ^T$. The SGD is such that, after randomly setting up W and Y:

$$W_i^{(n+1)} = \gamma * (error * Z_j^{(n)} - \lambda_W * W_i^{(n)}) \quad (2)$$

$$Z_j^{(n+1)} = \gamma * (error * W_i^{(n)} - \lambda_Z * Z_j^{(n)}) \quad (3)$$

With $error = log(X_{ij}) - W_i Z_j^T$, $\gamma = 0.000001$ and decreasing at each steps by a factor 1.2 and $\lambda_W = \lambda_Z = 0.01$. We use 300 dimensions or the embedding.

These embeddings are such that given a word (the key) we can recover its vector representation. Since we can't use this form of embedding in the CNN, we had to create a new embedding matrix that associates an index to a vector representation. To do this, we create first the vocabulary from our processed tweets. This vocabulary associated to each word an index, following the order of the embedding obtained from Gensim. We then add in both the vocab and the new embedding matrix a padding word and vector '<PAD>' represented by a null vector. Once done we pad all our tweets such that they all have the same size.

Now our data is ready to be given to our CNN. We have tried 2 different models that you can see below:

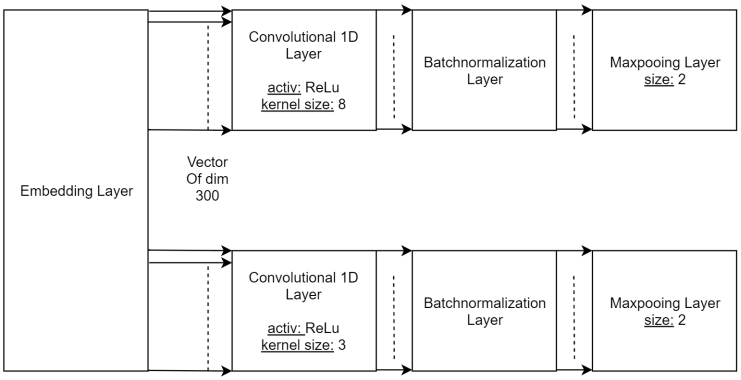

Fig. 2. Embedding Layer and **one** Convolutional Block of the CNN (this block is repeated a second time.

1) non-static CNN using convolutional blocks [5]: The CNN using convolutional blocks consist of an embedding layer creating the sentence representations followed by **two** serial convolutional blocks as the one described in Figure 2.

Then, following the two serial convolutional blocks, we implemented a flatten layer to merge the parallel buses followed by a dense layer using the ReLu (Rectifier linear unit function) activation function and a dense layer using a sigmoid. This lead to an array having at each index i the probability for the ith tweet to be positive. We are using the optimizer Adam to do the computations, with a learning rate of 0.001.

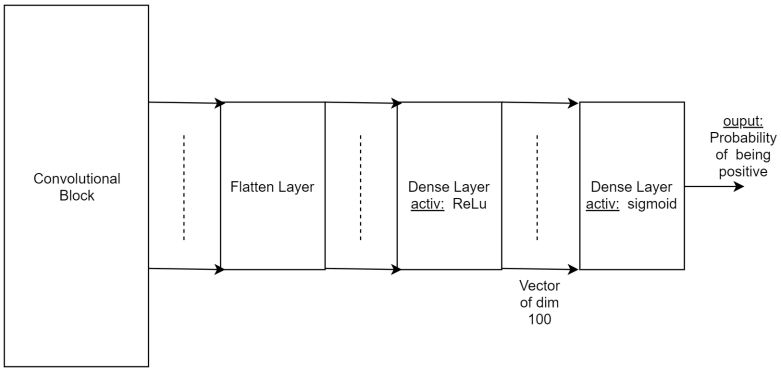

Fig. 3. Flatten layer and two dense layer.

2) sequential CNN: The CNN consists of an embedding layer, followed by two 1D convolutional layers with respective Kernel size of 8 and 3. They are followed by a maxpooling layer of size 2 as showed in the Figure 4.

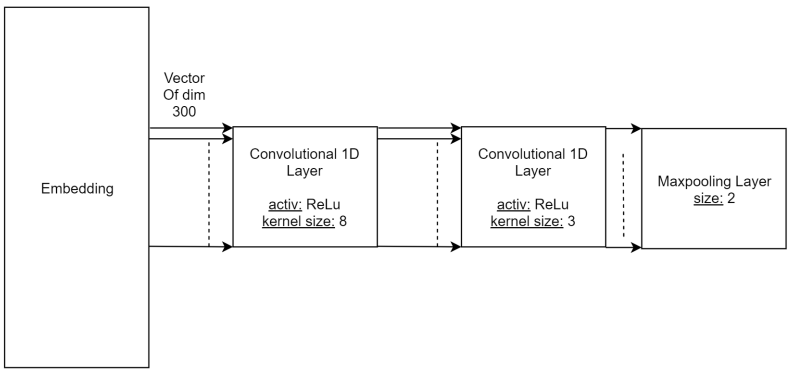

Fig. 4. Embedding, Convolutional Layers.

We then have a second time this scheme of convolutional layers followed by a maxpooling layer of size 1. The output of this layer is then processed through two dense layers using the ReLu activation function and one using the sigmoid as shown in Figure 5. This leads to an array having at each index i the probability for the ith tweet to be positive. We are using the optimizer Adam to do the computations, with a learning rate of 0.001.

In both of the models, we are using dropout (we drop some features with a given probability) so that we can avoid overfitting.

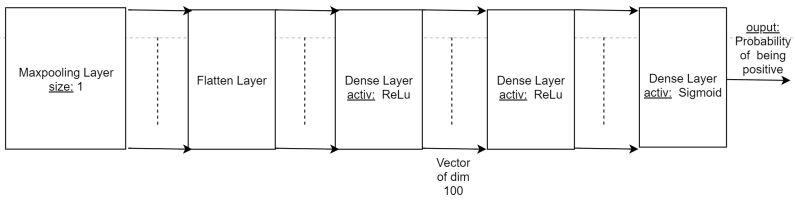

Fig. 5. Dense layers of the CNN.

D. Recurrent Neural Network (RNN)

As CNN learns to find patterns in space, a different kind of neural networks called the Recurrent Neural Network is often said to recognize patterns in time. For example, a RNN will easily learn that the word 'dog' has a completely different meaning if preceded by 'hot'. This memory is implemented using Long Short Term Memory (LSTM) cells in the neural network. They allow the neural network to remember information for a long period of time. They are particularly suited for speech and text analysis [6]. We implemented such a model using Keras. It is composed of an embedding layer followed by an LSTM layer of 250 LSTM units. A single 2 dimensional dense layer is added at the end. We used a spatial dropout of rate 40% in order to regularized our neural network by dropping an entire one dimensional feature. This allows us again to avoid overfitting.

E. Combined CNN and RNN

Combining a the spatial properties of the CNN and the time properties of the RNN seems to be the optimal sentiment analysis model. We combined the sequential CNN explained in part II-C.2 and the RNN described in II-D in a sequential way. As we will see on part III, this model was efficient, but not the best we implemented.

III. Results

In this part, we are going to present the results we obtain for the different methods described in part II.

A. Logistic Regression Result

With cross-validation and using Word2Vec we obtained a score of approximately 75%.
We did not try this method with the matrix factorization since when we implemented this embedding method we had already achieved better scores using neural networks instead of logistic regression.

B. Non-Static CNN using convolutional blocks

Using the CNN we presented in part II-C.1 we obtained much better results. But, when iterating too much our validation accuracy dropped because of overfitting, even though we used dropout.
Using Word2Vec we obtained a score of 72.03% (less than with logistic because we overffited a lot).

Then using the matrix factorization we achieved a score of 80.59% which was a big improvement.

C. Sequential CNN

Using the Sequential CNN presented in part II-C.2, we obtained better results than the previous one even though they are really similar. The huge drawback of this implementation is the computation time.
Using Word2Vec, we, this time, obtained an accuracy of 78.23%.
Then using the matrix factorization approach we reached an accuracy of 83.6%.

D. RNN

We did not train the RNN presented in part II-D using the Word2Vec embedding as it always gave us worse results than the matrix factorization approach for the CNN.
With the latter, we obtained a score of 84.7%.

E. Combined CNN and RNN

Finally we tried to combine both CNN and RNN ([8]).
Using matrix factorization we scored an accuracy of 84.44%.

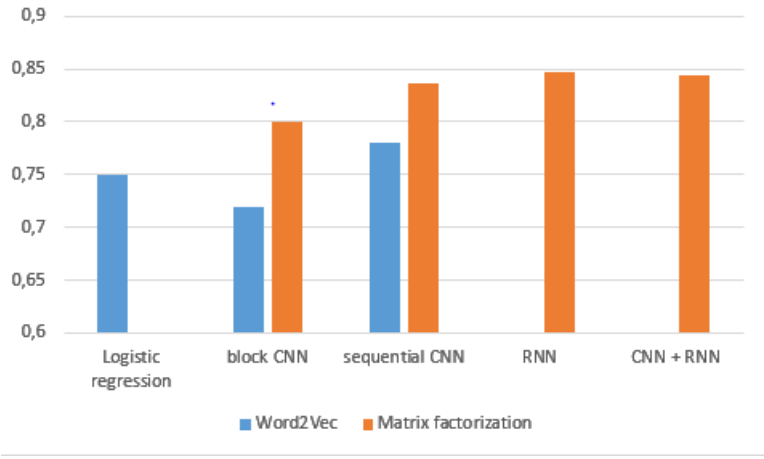

Fig. 6. Accuracy obtained on the validation data for the different models.

IV. Conclusion

While we knew from the lecture that Matrix factorization should achieve high results, the first accuracy we had was 0.55 consistently. After running a grid search on the lambdas we realized we were underfitting our embedding. Our best model was the Recurrent Neural Network using LSTM cells. We think that we did not get higher results with the combined CNN and RNN because of our embedding method which is highly time-consuming and quite simple.

References

[1] Edward Loper and Steven Bird, *NLTK: The Natural Language Toolkit.*, https://www.nltk.org/

[2] Hitanshu Tiwari, *Slang dictionnary*, https://floatcode.wordpress.com/

[3] Radim Řehůřek and Petr Sojka, *Software Framework for Topic Modelling with Large Corpora* https://radimrehurek.com/gensim/index.html

[4] Pedregosa, F. and Varoquaux, G. and Gramfort, A. and Michel, V. and Thirion, B. and Grisel, O. and Blondel, M. and Prettenhofer, P. and Weiss, R. and Dubourg, V. and Vanderplas, J. and Passos, A. and Cournapeau, D. and Brucher, M. and Perrot, M. and Duchesnay, E. *Scikit-learn: Machine Learning in Python*,

[5] Yoon Kim. Inspired by Denny Britz article "Implementing a CNN for Text Classification in TensorFlow". *Convolutional Neural Networks for Sentence Classification*, https://github.com/alexander-rakhlin/CNN-for-Sentence-Classification-in-Keras

[6] Christopher Olah, Understanding LSTMs http://colah.github.io/posts/2015-08-Understanding-LSTMs/ Sepp Hochreiter and Jurgen Schmidhuber, Long short term memory (1997) https://www.bioinf.jku.at/publications/older/2604.pdf

[7] Franois Chollet, Keras https://github.com/fchollet/keras

[8] Pedro M Sosa, "Twitter sentiment analysis using combined LSTM-CNN models", February 19, 2018, http://konukoii.com/blog/2018/02/19/twitter-sentiment-analysis-using-combined-lstm-cnn-models/