Twitter Sentiment Analysis

Emna Fendri - Ahmed Ezzo - Mohamed ELasfoury
Department of Computer Science, EPFL, Switzerland

***Abstract*—In this paper we aim to train a model that predicts if a tweet message used to contain a positive :) or negative :(smiley, by considering only the remaining text. We discuss different Machine Learning, Natural language processing and Deep Learning techniques to perform this task and find out which model gives the most accurate results.**

I. Introduction

Twitter is a popular online news and social networking site with more than 1.86 billion active users, allowing them to communicate and express themselves in short and informal messages called tweets. With the use of machine learning, we will be able to assign a positive or a negative sentiment to an entire tweet. This task refers to polarity classification of tweets which can be sometimes challenging due to terseness and informality of the text in addition to the wide variety and rapid evolution of language in Twitter. To this end, we use a data set of 2.5 million tweets with the same number of positive and negative tweets. We pre-process the data, study various feature representations schemes that we apply on several models.

II. Data Pre-processing

Feeding the machine learning model the best data possible is an important step. In fact, making the tweets more structured will help in representing the text and therefore, in selecting the most convenient feature representation.
In this section, we focus on the precise steps we followed for the cleaning.

A. Unpacking hashtags

Hashtags are user-generated tags to label the content of posts. Thus, they hold important key words that could help to detect the polarity of a given tweet. To this end, we use the *ekphrasis* library that performs on top of word segmentation a spell correction using word statistics (unigrams and bigrams) from 2 big corpora (the English Wikipedia and a collection of 330 million English Twitter messages). E.g., `#happyHolidays`→ `hashtag happy holiday hashtag`.

B. Handling repeating characters

We handle repeating characters by normalizing to 2 repetitions.E.g., *"goood"→"good"*.

C. Handling vocabulary

We also unpack grammatical contractions e.g., `can't`→`can not` and popular English slang contractions e.g., `2mro`→`tomorrow`.

D. Handling special symbols

We replace `!` and `?` symbols by `exclamation` and `question` respectively.
We replace emoticons by their explicit meaning e.g., `:)`→`happy`, `<3`→`heart`.
We normalize terms like numbers, time, dates and money using *ekphrasis*. Expressions such as:

- `10/17/94` or `15 December 2021` become `date`
- `5:30` or `5:45pm` become `time`
- `$2B` or `€10` become `money`

We then replace any other number occurrence by `number`. We also keep the tags `user` and `url`. This being done, we then remove what is left of the punctuation.

E. Lemmatizing

We lemmatize each word to reduce it to its lemma i.e. root word. E.g., *"better"→"good"*, *"walking"→"walk"*.

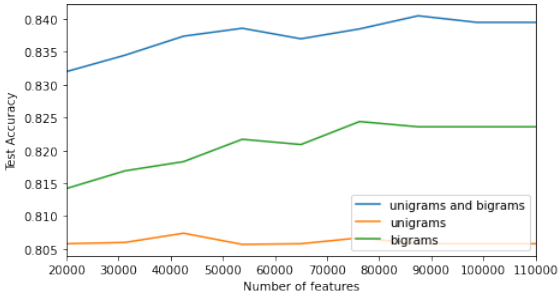
III. Feature representation of the text

A. Bag of words retrieval

Natural language text carries a lot of meaning, which still cannot fully be captured computationally. A basic approach to perform information retrieval is to use the words that occur in a text as *features* for the interpretation of the content. This method ignores most of the grammatical structure of text and reduces texts essentially to the terms they contain. This approach is called "full text" or "bag of words" retrieval and is a simplification that has proven to be very successful.
We first define a vocabulary V of size M by gathering the M most frequent words encountered in the training set where each word has its own index. Each tweet is modeled by a vector of size M where position i represents the number of occurrences of word $w_i \in V$ in that particular tweet. Our feature matrix will result in a sparse matrix of size $N \times M$ where N is the size of the training set. This first representation is known as **count vectorisation**.
Another variant, which led to better results is to give a **TF-IDF** weight to each token. This weight takes into account how frequent a term occurs within a tweet (measure for similarity) and how frequent a term occurs within the entire corpus of size N (measure for distinctiveness). For instance, stop words normally occur in every tweet, their weight will normally be close to 0.
Moreover, for both of the vectorizations it is possible to select the n-gram range. We see from Fig. 1 that using both uni-grams and bi-grams improves considerably the accuracy. Which is not surprising as each word in a textual data does

depend on the surrounding words. For instance, the word "like" alone has an opposite meaning compared to the bi-gram "don't like".

Fig. 1. Accuracy as a function of the number of features using different combinations of uni-gram and bi-grams with logistic regression



B. Word embedding

Word embedding is another approach for representing words and documents using a dense vector representation as apposed to the traditional bag-of-word model seen in the previous section. In an embedding, words are represented by dense vectors where a vector represents the projection of the word into a continuous vector space. The position of a word within the vector space is learned from text and is based on the words that surround the word when it is used. The position of a word in the learned vector space is referred to as its **embedding**. Their aim is to map semantic meaning into a geometric space. This geometric space is then called the embedding space.

1) GloVe:

The GloVe (Global Vectors) blends the advantages of global matrix factorization and local context window methods [1]. Thanks to the co-occurrence matrix, the GloVe method comprises global statistics in its embeddings. To create the co-occurrence matrix from a set of tweets, we need to:

- Build a vocabulary of all the words that appear in the tweets. We can reduce this vocabulary by removing words that don't appear in the tweets more than five times.
- Create a matrix with the size $N \times N$, where N is the number of words in the vocabulary.
- Populate the matrix such that the i^{th} row and the j^{th} column contain how many times the i^{th} word in the vocabulary and the j^{th} word in the vocabulary appear together in the same tweet.

The idea is to create an embedding space where how close two vectors are to each other relate to how similar these words are. The paper uses the famous example of ice and steam. We define $P(i|k)$ as the probability that i and k appear together. We can quickly get this number from the co-occurrence matrix by looking at the element at the ith row and kth column and dividing it by the total number of tweets. Now we say that $P(k|ice)$ is how many times the word k appears with the word ice. Similarly, with $P(k|steam)$.
We now look at an interesting number: $P(k|ice)/P(k|steam)$. In this quantity, we call k "the probe word". We see that if $P(k|ice)$ is more significant than $P(k|steam)$, this quantity will be greater than one. Hence, the word k is similar to ice but irrelevant to steam. Conversely, if it is smaller than one, it is more related to steam. If it is unrelated to both, they will have almost the same probabilities, and this quantity will be close to 1.
Therefore, this quantity is essential to compute our word embeddings since it encompasses the essence of global statistics.

Using this as a starting point the paper derives the final cost function:

$$J(w_i, w_j) = \sum_{i,j=1}^{V} f(X_{ij})(w_i^T w_j - b_i + b_j - log(X_{ij})^2$$

Where:
V is the total number of words in the vocabulary.
w_i is the embedding of the word i.
w_j is the embedding of the word j.
f A function with the following properties:

- $f(0) = 0$
- $f(x)$ should be non-decreaseing
- $f(x)$ should be relatively small for large values of x.

We use the function proposed by the paper: $f(n) = min(1.0, n/nmax)^\alpha$. Where $nmax$ is a hyper-parameter.
X_{ij} the entry at the i^{th} row and j^{th} column in the co-occurrence matrix.
For the purpose of simplicity we ignore the bias variables b_i and b_j.
The derivative of this cost function with respect to w_i is:

$$2f(X_{ij})(w_i^T w_j - log(X_{ij}))w_j$$

Similarly when getting the partial derivative with respect to w_j.
Finally, using gradient descent, we get an embedding for all words w_i such that it minimizes this cost function.
It is important to note that we have used a pre-processed GloVe embedding. The creators of GloVe provided this embedding, and they trained it on a large dataset of tweets. We used the **pre-processed Glove** because it was outperforming our **custom GloVe**. We believe that the reason is that we trained our embedding on a smaller dataset.

2) CBOW:

Another model we tried to get the word embeddings is an extended version of the CBOW model that has been trained using the fastText library.
The model has been trained using CBOW with position-weights, in dimension 300, with character n-grams of length 5, a window of size 5 and 10 negatives as described in section 3 of the original paper. [2].
A brief explanation of how the model was trained is that they added position dependent weights to capture positional information and the word embeddings for a word is obtained by summing the vectors of the character ngrams appearing in the word.
This model has been trained on a dataset that contains a

mixture of Wikipedia and Common Crawl. One of the benefits of using the fastText library is that it is possible to get a word vector for a word that was not in the training set that was used to train the model, as opposed to the GloVe model.
We also used the fastText library to create our own CBOW model, which gave us better results than the pretrained one (0.80 accuracy vs 0.78).
To obtain the training data we go over all the tweets in our training set and to get one training sample we get the word embedding of each word in the tweet and add it to a matrix and then add the classification of this matrix.

IV. Basic Machine Learning Models

Now that we have a valid input representation we start to solve this classification problem with basic machine learning models from $sklearn$ library. We use cross validation to select the hyper parameters and summarize the results in the table below. Note that we use an $(1, 2)$ n-gram range as it gives us better results.

Method	Accuracy	AUC	Number of features
Naive Bayes + count	0.768	0.857	110 000
Naive Bayes + TF-IDF	0.797	0.887	100 000
Logistic regression + TF-IDF	0.840	0.920	90 000
SVC + TF-IDF	0.81	N/A	90 000

TABLE I
Models and Accuracies

We see from this table that logistic regression resulted in a **0.84** accuracy on our clean data on AIcrowd. We noticed a significant improvement when we kept stop words in our text. This basic base line actually helped us to adjust our cleaning steps, where the final version was described in the previous section. However, the multinomial Naive Bayes classifier gave us less better results, which is probably due to the strong assumption made during calculations, that the features are independent which is surely not true for text data.

V. Deep Learning Models

We now look at deep learning models that will train on top of an **embedding layer**. Note that we'll use word embedding to represent our data as deep learning models converge easier with dense vectors than with sparse ones. We report our results in Table II.

A. Embedding layer

$Keras$ library offers an embedding layer that can be used for neural nets on text data. The Embedding layer is initialized with either random weights or with a predefined embedding matrix. If we allow training, it will learn an embedding for all of the words in the training data set. However, if training is not allowed, the embedding matrix is kept constant during training and this generally lead to less better results. We therefore try with different embedding techniques and report the results in Table II.
We first need to define some parameters:

- `input_length`: This is the maximum number of tokens in a tweet. We use post-zero-padding so that all the tweets have the same length. In this project we set it to 100.
- `output_dim`: This is the size of the vector space in which words will be embedded.
- `input_dim`: The size of the vocabulary containing the tokens. For this task we take the 150000 most frequent words that occurred in the training set.

With these settings, the **embedding matrix** will have a shape of `input_dim` $\times$ `output_dim`, where row i represents the embedding of the word of index i.
Each tweet will be encoded so that each word is represented by a unique integer i.e. with `input_length` = 150000, this integer index takes values between 1 and 150000, as value 0 is reserved for padding (Note that during training, 0 values will be ignored).
Finally, the **embedding layer** takes as input the **encoded tweet**, use the **embedding layer** as a look up table to map each token with its embedding and output an `input_length` $\times$ `output_dim` matrix to represent the tweet as a whole.

B. Implementation Choices

We chose to set the commonly used Adam optimizer which is an SGD method that is based on adaptive estimation of first-order and second-order moments. We use the ReLU function as the activation function inside our neural net. The sigmoid function is used at the output layer (1 node) to compute the probability of a tweet is positive. We finally set our threshold to be 0.5. Before every dense layer note that we use a 0.4 dropout layer.

Method	(Embedding,Trainable)	Accuracy	Hyper-parameters
Feedforward NN	(GloVe,True)	0.838	epochs=5, batch=32, output_dim=200
Feedforward NN	(CBOW, False)	0.801	epochs=13, batch= 64, output_dim=100
CNN	(GloVe,True)	0.845	epoch=4,batch=32, output_dim=200
Multi-channel CNN	(GloVe,True)	0.861	epoch=15,batch=1024, output_dim=200
Multi-channel CNN	(GloVe,False)	0.851	epoch=15,batch=1024, output_dim=200
Bidir-LSTM	(GloVe, True)	0.861	epochs = 2, batch = 1024, output_dim=200
Ensemble Learning	N/A	0.853	Using Basic NN, Multi-channel CNN and CNN

TABLE II
Deep learning models and Accuracies from AIcrowd

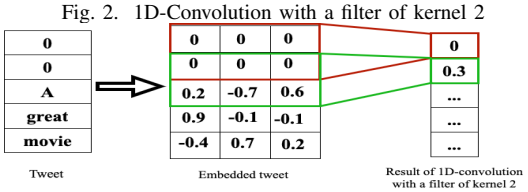
C. Models

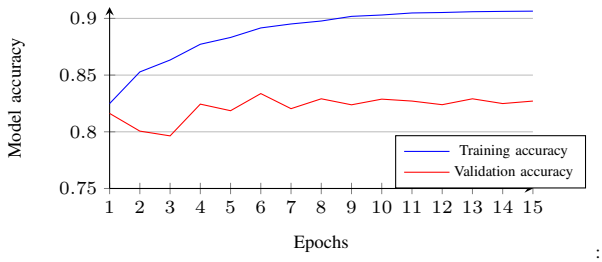
1) Forward Neural Network: We first use a basic neural network implementation to give us some insight on how it could perform on our clean training set. Glove embedding led to better results than CBOW embedding.

2) Convolution Neural Network: Convolutional networks were originally developed for computer vision tasks, however they have been recently applied in NLP problems and results are promisisng [3]. A typical convolutional model for texts usually consists of a convolutional layer applied to word embedding, which is followed by a non-linearity (ReLU) and a pooling operation.

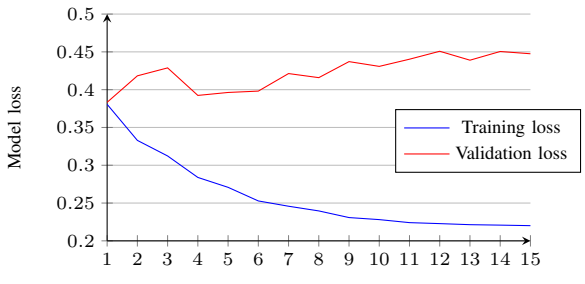
For images, filters capture local visual patterns. For text, such local patterns are word **n-grams**. 1D-Convolving filters (sliding vertically) are therefore used as n-gram detectors. In fact, the embedding of two similar words are similar in terms of cosine distance (i.e. close in the embedding space), and applying the same filter will result in a similar result (Fig. 2). In addition, max-pooling induces a thresholding behavior, where values below a given threshold are considered to be irrelevant to make a prediction. The main advantage is that this technique can be easily extended to several n-grams and as opposed to the bag of words representation our feature matrix won't explode. We try a first CNN with 128 filter of kernel 5, we then adopt a Multi-channel CNN model which gave us the best accuracy, which architecture is described in 3, we also monitor the accuracy and loss evolution in order to fine-tune our model.

Fig. 2. 1D-Convolution with a filter of kernel 2



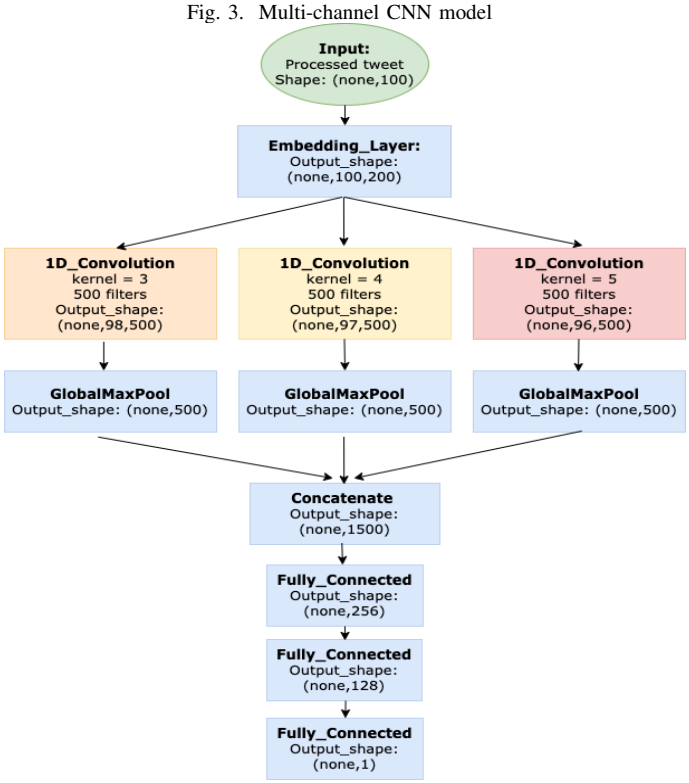


Accuracy evolution for Multi-channel CNN model



Loss-evolution for Multi-channel CNN model

Fig. 3. Multi-channel CNN model



3) Long Short Term Memory: A LSTM network is a type of RNN capable to seize long term dependencies in one direction (either forward or backward).The bidirectional LSTM processes the data on both sides to persist the information and indeed gave us better results than LSTM.

4) Ensemble Learning: In machine learning, ensemble methods is a general approach that seeks better predictive performance by combining the predictions from multiple models. In our case we'll take the three best models, train them and use the obtained predictions, setting them at equal weights for the final prediction. This method didn't gave us better results.

VI. Conclusion and Discussion

We finally conclude that deep learning models led to consistent results in particular, with the Bidir-LSTM and the Multi-channel CNN with 0.861 accuracy on AIcrowd. It is hard to reach an 1.0 accuracy in this particular task due to the informal twitter language style and the possible inconsistency between a text and a smiley (use of sarcasm). Nevertheless, we still believe that with more computational resources, a better accuracy could've been reached. In fact there is a significant amount of techniques we could have used, each with even more parameters to optimize. However we still managed to tackle various interesting techniques in this project and to learn more about NLP tasks.

Acknowledgements

We would like to thank all the Machine Learning class collaborators for their help and for making this project possible.

REFERENCES

[1] S. R. . M. C. D. Pennington, J., "Glove: Global vectors for word representation," *In Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014.

[2] P. G. A. J. T. M. Edouard Grave, Piotr Bojanowski, "Learning word vectors for 157 languages," 2018.

[3] N. N. Muhammad Zain Amin, "Convolutional neural network: Text classification model for open domain question answering system," 2018.