CS-433 Machine Learning - Twitter Sentiment Analysis

Department of Computer Science, EPFL, Switzerland
Arman Ehsasi, Aleksandar Hrusanov, Thomas Stegmüller

Abstract—**Machine Learning has become increasingly important in the last decades and has revolutionized many industrial and scientific sectors. A very common application of Machine Learning is in the field of Natural Language Processing (NLP). Indeed, it is the mechanism that allows a user to find documents related to the textual query they entered on their favourite search engine. This task is called document retrieval and is one of the fundamental tasks handled by NLP. Another typical text driven task is sentiment analysis, which is the objective of this paper. We propose an approach to perform sentiment analysis on a compilation of *tweets*, which are messages published on the *microblogging* platform Twitter. More specifically, we want to predict the positive or negative emotions (e.g. use of positive or negative emoticons) in a tweet based on its textual context. Our best method achieved an accuracy score of 0.843 on the online test set in the AIcrowd competition platform.**

I. Introduction

Sentiment analysis aims to, more generally, classify the polarity of a given document (e.g. word, sentence, text, etc.) and decide whether it has a *positive*, *negative*, or *neutral* tone thereby deducing an emotional expression. Sentiment analysis has a wide range of applications, for example when it comes to analyzing customer's opinions, preferences, and aversions (presented on e.g. social media, review sections of websites, blogs).

Our project analyses a set of Twitter data (the *corpus*), which contains tweets (the *documents*) limited to a maximum of 140 characters. We aim towards predicting whether a given tweet used to contain a positive or negative smiley emoticon after it was stripped from these emoticons. In comparison to other binary classification problems the raw data regarding text classification is not numerical and thus features relating to each document (a sentence of words in this case) must be first created in order use machine learning algorithms. Due to its own disciplinary complexities, this task extends beyond the scope of pre-processing and is known as "feature extraction" or "word vectorization" , and represents the underpinning part of any successful text driven application.

This paper summarizes our efforts at cleaning the corpus, producing a meaningful set of features per document, applying various classification algorithms and discussing their respective results.

II. Exploratory Data Analysis and Pre-Processing

The data we worked with consisted of two annotated training sets of size $2.5 \cdot 10^6$ and $2 \cdot 10^5$ tweets and an unannotated test set of size 10^4 tweets. The annotations of the training data indicate whether a tweet used to contain a *positive smiley ":)"* or a *negative smiley ":("*.

Many tweets are characterized by the use of informal language and some Twitter-related terminology resulting in the data containing, amongst other things, incorrect spelling, word concatenation, common usage of hashtags, username mentions, URLs, contractions and emoticons. During our data pre-processing stage we focused on reformatting the data to take some of those specific text features into account. Before further explaining the pre-processing steps it is important to specify that the decision was made to use the pre-trained *GloVe* [1] embedding as it strongly influences some of our pre-processing choices. The motivation behind the usage of a pre-trained embedding will be made explicit later in the report. Many of the pre-processing steps were inspired by the methods used in *GloVe*.

A. *Tokenization*

Tokenization is the process of mapping a string to a sequence of tokens/words. This task is not as trivial as it might appear especially when dealing with special characters. For that purpose, we used an existing tool, `TweetTokenizer`, provided by the library `NLTK` [2].

B. *General Text Cleaning*

1) Lower-casing: Widely regarded as a standard routine the provided raw data was already lower-cased for us. However, it is worth mentioning that changing the first letter of words can change the meaning of the word. For example the name *"Rose"* changes to the noun *"rose"* [3].

2) Characters and Numbers: Numbers tend to not carry any specific sentiment, we therefore simply removed stand alone numbers from each document. Further, if there is a numerical character concatenated with with a word, we remove those as well. Punctuation and special characters were also removed from the corpus. Nonetheless, some punctuation such as exclamation or interrogation marks are likely to be used when emphasizing an emotion and should probably be included. The special character *"#"* (hashtag) found in our documents is used to link tweets with similar thematic content together. Although, it is considered as uninformative in many text classifications, its use may hold a lot of information for our case as indicated in [3].

3) Misspelled Words: Tweets contain a lot of informal language. It is common that people emphasise certain words by repeating a character multiple times (e.g. *"I am so* *saaaaaad"* or *"I am so happyyyyyy"*). Although, we implemented a function that could handle such expressions we kept them as is, since most of the words containing repeated letters had a pre-trained embedding in *GloVe*. Additionally, this choice had the added effect to make the pre-processing task faster.

4) Stop Words: The term *stop words* refers to some of the most commonly used words in a language (e.g., in English, e.g. articles such as *"a"* and *"the"*). Depending on the classification task, such words carry little to no predictive power and thus removing them significantly reduces the size of the corpus without harming the model. In the specific case of sentiment analysis *stop words* might not be harmless. Indeed the two sentences *"I did get the job"* and *"I did not get the job"* are unlikely to have the same associated sentiment if a word such as *"not"* were to be removed, which is commonly included in the *stop words* list. For that reason *stop words* were not removed from the corpus.

5) Rare words: Some words appear very seldomly in the corpus. They are thus unlikely to have an existing pre-trained embedding and therefore would not have a significant impact on the accuracy of the model. Additionally, removing such words can greatly reduce the sparsity in the embedding space when using *Bag of Words* or *TF-IDF* as word representations. Words appearing less than 5 times in the complete corpus (large train set + test set) were removed. A more flexible approach would be to use a probability of appearing of the treated word w as the decision criterion. Since the probability of appearing is unknow it can be empirically estimated by $P_{D,W}(w) = docfreq(w)/N_W$, where N_W refers to the total number of words in the corpus. This is indeed a better criterion as it is applicable regardless of the corpus. When this step was applied to the small set of tweets with a threshold of at least two apparitions, the number of different words was reduced by 50% hereby confirming the intuition that the *Zipf's law* is a likely distribution for the words' occurrences.

6) Expansion of Contractions: We used a dictionary of commonly used contractions and transformed them into their multi-word equivalent. In this way, no unnecessary words were added to the vocabulary and possible negations were revealed to model *don't* as *do not* for example.

C. *Stemming*

Stemming refers to the process of mapping a word to its root, base or normalized form by stripping commonly used prefixes and/or suffixes [3]. The major benefit of this trick is to reduce sparsity in the embedding space. However, this technique is regarded as fairly "aggressive" since the method does not always return a word found in the dictionary. This can lead to a significant loss of information, where in the particular case of sentiment analysis mapping the word *"unhappy"* to *"happy"*, for example, falsifies the data.

D. *Lemmatization*

Lemmatization can be seen as an improved version of *Stemming* as it maps a word to its *lemma*, by stripping commonly used prefixes/suffixes with the additional constraint that the *lemma* must be a valid entry of the dictionary used by the algorithm. Moreover, this process uses the context in which the word appears in and its category to deduce the appropriate *lemma*. We decided to implement this process in our pre-processing.

E. *Hashtag Handling*

Hashtags are very commonly used in tweets. They are usually followed by two or more concatenated words (e.g. *"#ilovemyjob"*). Treating the hashtags is not a straight forward task. Indeed, the concatenation of words sometimes bears a completely different meaning as opposed to processing them separately, as it is with the case of *#metoo*, referencing the *Me Too* movement. Splitting the hashtag might not be the most appropriate method. On the other side, keeping the hashtag as a whole significantly increases the sparsity of the embedding matrix with *Bag of Words* or *TF-IDF* as representation. During the cleaning process the hashtags were completely dropped from the corpus. A posteriori, a better approach would have been to treat the concatenated words as normal words and only keep those who had an appearing probability larger than a fixed threshold.

F. *Before and After Comparison*

As an example of the pre-processing step, *"<user> no way he's mine ! ! !"* gets mapped to *"no way he is mine"*. To better assess the performance of the overall pre-processing, we compared the accuracy of the *Logistic Regression*, using *TF-IDF* for our word vectorization, between the cleaned and raw corpus. The accuracy obtained under the two setups was identical and equal to 80%. The latter indicates that at least the cleaning does not remove essential information from the data. When assessing the pre-processing step one also has to consider the number of distinct words in the corpus. The cleaned corpus only contains 12% of the original number of words while maintaining the same predictive power, which indicates that the selected words are meaningful. Furthermore, with such a reduction in the vocabulary we'll be able to fine tune a pre-trained embedding matrix during the training stage. Finally, since the final model will be a *Neural Network* it is preferable to clean the data set a little too much than too little as these models are well know to easily overfit due to their large number of parameters.

III. Word Vectorization

After concluding the pre-processing steps, we must choose an appropriate method to convert the documents into numerical features in order apply various machine learning algorithms. We tried the following three word embedding techniques: *Bag of Words*, *Term Frequency-Inverse Document Frequency (TF-IDF)*, and *Word Embeddings*.

A. *Bag of Words*

Bag of Words (BoW) is one of the most basic text representation models in which a given text is transformed into the multiset of its words - e.g. disregarding word-ordering and grammar but only taking multiplicity into account. A common application of *BoW* is word vectorization, where term frequencies $TF(w, d)$ (number of occurrences of a given word in a text) are computed for each word per document, mapping the corpus to a matrix. Thus each document is represented by a vector of the same length, which can be used as features in a Machine Learning model. We used the *scikit-learn* library for implementing this method [4].

B. *Term Frequency-Inverse Document Frequency (TF-IDF)*

The *Term Frequency-Inverse Document Frequency* model is based on the idea of *BoW* using the same term frequency computation, however includes a term to discount words with less predictive power. To account for words that are more frequently used in general rather than only in a specific document (e.g. *the*), the term frequency is offset by the number of documents in the corpus in which the word is present. To calculate the word matrix we use $TF(w, d) * IDF(w)$ with $IDF(w) = log(N/docfreq(w))$, where N is the total number of documents in the corpus and $docfreq(w)$ the number of documents containing w. Again, we used the *scikit-learn* library for implementing this method [4].

C. *Word Embeddings*

Word embeddings uses fixed size high dimensional real-number vectors to represent words. This representation has the advantage that it maps similar words used in similar contexts to vectors which are also close to each other. Another example of such relationship between word vectors can be seen if one performs some linear operations on two or more such vectors - e.g., the commonly quoted example of using the vector representations of the words *King, Man, Woman, Queen* as follows: $King - Man + Woman \approx Queen$.

D. *Comparison of Vectorization Methods*

Word Vectorization Method	Accuracy	Feature Dimension
Bag of Words	80%	5000
TF-IDF	80%	5000
GloVe	77%	200

Table I: Vectorization Methods Comparison

In order to determine the optimal feature extraction method, we compare the performance of the different vectorization methods on a *Logistic Regression* model in Table I. All the training was done on the smaller set of tweets, using 3-fold cross-validation, l2 penalization and without *stop-words* removal. Even though the score obtained with the *GloVe* embedding is slightly lower, this representation retains more information about the words as the corresponding score was achieved with eight times less number of parameters.

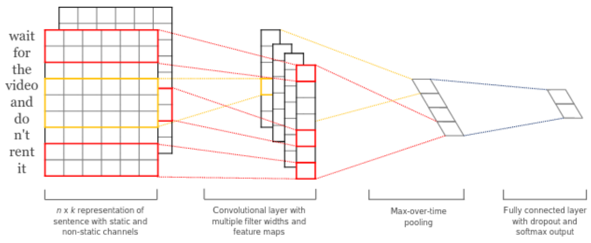
wait
for
the
video
and
do
n't
rent
it
n x k representation of sentence with static and non-static channels
Convolutional layer with multiple filter widths and feature maps
Max-over-time pooling
Fully connected layer with dropout and softmax output

Figure 1: CNN Architecture with Textual Input [5]

IV. Classification Methods

As a result of the observations made in section III we decided to use the pre-trained *GloVe* embeddings for our word vectorization model. This is indeed a reasonable choice as the embedding dimension is greatly reduced, independent of the corpus and finally the embedding matrix is not sparse, which is needed as sparse representation can be a source of overfitting. Two types of models were implemented *Recurrent Neural Networks (RNN)* and *Convolutional Neural Networks (CNN)*, which have recently been proven to be effective with text classification and even improve current state of the art results for *Sentiment Analysis* tasks [5]. Both models inherit properties and methods from an abstract class *TextNet* using a batch of tweets as the input data, converting the documents to a numerical representation on the fly. There are two major improvements brought by these types of models in comparison to the *Logistic Regression* that was previously used:

- The tweet representation retains the property that tweets are ordered sequences of words.
- The embedding matrix can be finely tuned during training stage, which allows new words to be added to the vocabulary.

Effectively, the representation of a single tweet is a matrix $M \in \mathbb{R}^{|W_{doc}| \times D}$ where D refers to the dimension of the embedding space and $|W_{doc}|$ to the number of words in the tweet. When using batches of tweets, some padding occurs in order to compensate for different the number of words in a tweet.

A. *Convolutional Neural Network (TextLeNetCNN)*

This model is inspired by the architecture described in [5] and treats a tweet as a standard *CNN* would treat a single channel image, with the additional constraint that the horizontal dimension of all filters is fixed to be D. On the other hand, the vertical dimension fixes the number of words that are convoluted with the kernel. When dealing with differently sized kernels, a pooling operation along the word dimension is needed to make the dimensions match as depicted in Figure 1. This model uses the pre-trained *GloVe* embedding with 25 dimensions. Moreover, additional entries in the embedding matrix were created for words that did not have an existing embedding representation. The total number of parameters in the model was 1842081 among which 1829775 were attributed to the embedding matrix. This number corresponds to a vocabulary size of 73191 words.

B. Recurrent Neural Network (GRUNet)

This model is based on the *Gated Recurrent Unit*, which is specially tailored to deal with sequential data as it treats words one after the other and further, creates a temporary output which can be reused together with the next word to create a new temporary output. We observed a significant improvement in the predictive power of the model when the inputs were processed in a bidirectional way. The total number of parameters in the model was 2396817 among which 1829775 were attributed to the embedding matrix. This number corresponds to a vocabulary size of 73191 words.

V. Results

The two previously described models were trained on the large set of tweets that had previously been split in train, validation and test sets of respective size $2 \cdot 10^6$, $2.5 \cdot 10^5$ and $2.5 \cdot 10^5$ tweets. To improve the speed of convergence and regularize the training, *Batch-Normalization* and *Dropout* layers were used extensively and were observed to help significantly. The two models were first trained with a similar number of parameters ($\approx 1.9 \cdot 10^6$) and obtained comparable results of (accuracy ≈ 0.83). As the *GRUNet* seemed to converge faster, its capacity was thus increased to get a final accuracy score of 0.843 on the online test set. A *GRUNet* was also trained using $D = 200$, to obtain similar accuracy indicating that the embedding dimension was not the bottleneck for this task.

VI. Post-Processing

The *GloVe* embedding is well known for its interesting latent space structure. In order to gain some insight about this representation and to verify that fine tuning the embedding matrix did not break the structure of the latent space we performed some post-processing on the trained model. In particular, we explored the possibility of applying a clustering algorithm in the latent space. The author of *GloVe* suggests that semantically similar words are separated by a small euclidean distance suggesting the *K-Means* algorithm is an appropriate choice. Performing the *K-Means* algorithm revealed a significant amount of highly interpretable clusters such as foreign languages, food aliments, medical terms, movies names, locations, etc. It was remarked that increasing the number of clusters improved the interpretability of the different assignments but not necessarily the separability of the data.

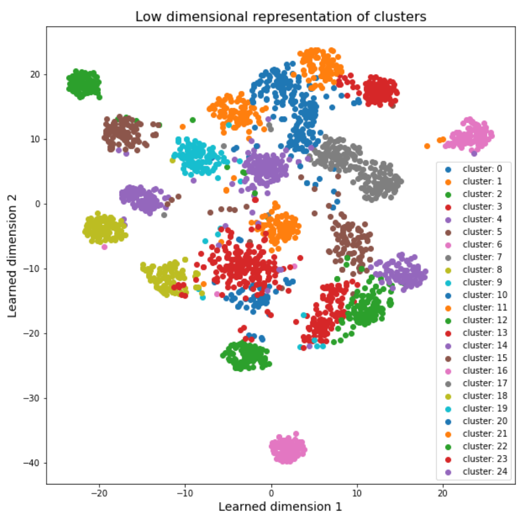

Figure 2: 2D Representation of the Clustering on the Embedding Space

Figure 2 displays the result obtained by running the K-Means algorithm on the 25- dimensional embedding space with 25 different clusters. The *TSNE* algorithm allowed us to observe this result. The latter aims at learning a specified number of new dimensions such that two close points in the original space are close to each other in the learned latent space and reciprocally. In Figure 2 we can clearly see some clusters appearing, nonetheless this plot should be taken with a grain of salt as it only shows the closest points to each centroid. If the complete set of words were to be used, the plot would be much more cluttered and wouldn't show such neat separations between different clusters. This observation is actually coherent with our intuition that some words can belong to more than a single cluster and that a soft assignment might be more suitable in this case.

VII. Outlook

This paper demonstrated the complete pipeline that allowed us to get from raw textual data to sentiment predictions at a document level. It was observed that with regards to accuracy the pre-processing step bears the most importance and particularly when the capacity of the models used increases. We came to the conclusion that a more cautious cleaning, notably with the hashtags handling could further improve the overall accuracy score.

References

[1] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543. [Online]. Available: http://www.aclweb.org/anthology/D14-1162

[2] E. Loper and S. Bird, "Nltk: The natural language toolkit," in *In Proceedings of the ACL Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics. Philadelphia: Association for Computational Linguistics*, 2002.

[3] . S. A. Denny, M., "Text preprocessing for unsupervised learning: Why it matters, when it misleads, and what to do about it," *Political Analysis*, vol. 26(2), p. 168–189, 2018.

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[5] Y. Kim, "Convolutional neural networks for sentence classification," pp. 1746–1751, Oct. 2014. [Online]. Available: https://www.aclweb.org/anthology/D14-1181