Twitter sentiment analysis

☆ Stefano Huber, Orazio Rillo, Catinca Mujdei ☆

***Abstract*—In this day and age, the amount of data shared by users on social platforms, by reviewers of certain products, etc. is ever increasing. This multitude accounts for a great demand of efficient machine learning models that can classify the data into predefined categories. In our project, the dataset is a large collection of tweets that are labeled by either a positive or a negative sentiment. We considered several learning methods, from simple naive Bayes to recurrent neural networks, and looked at some extensions in hopes of improving our results.**

I. Introduction

For this machine learning project we studied several approaches to the problem of binary sentiment classification of tweets. Each of these models are listed in this report, in an ascending order corresponding to our impression of their divergence from the standards. The structure of this report is as follows: section II deals with the general notation used throughout the report, in section III the cleaning methods are mentioned, section IV deals with the different considered models, and the final section V provides a summary of our findings.

II. Notation

The general problem of text classification can be formulated as follows [1]: given a set of J classes $\mathcal{C} = \{c_1, \dots, c_J\}$, a document space $\mathbf{D}$, a training set $\mathcal{D} \subseteq \mathbf{D}$ of which the elements d are labeled as $(d, c) \in \mathcal{D} \times \mathcal{C}$, construct a learning method that learns a classifier γ that accurately maps documents to their corresponding class, i.e. $\gamma : \mathbf{D} \to \mathcal{C}$. We further call the vocabulary $\mathcal{V}$ the set consisting of all terms occurring in the training set $\mathcal{D}$. In our specific case, $\mathcal{D}$ is a set of tweets and $J = 2$, such that c_1 represents a positive smiley/sentiment and c_2 a negative one.

III. Data cleaning

In order to feed the data to each learning model, some cleaning is required so as to not consider unnecessary words or characters, and similar concerns. We therefore implemented some cleaning functions such as cleaning tags, cleaning punctuation, cleaning spelling, transforming to lowercase, removing Saxon genitive, removing stop words, removing numbers, removing URLs, applying translation, and lemmatizing. Some cleaning methods might contribute more to the learning task than others, therefore a combination of an arbitrary subset of these can also be considered.

IV. Models and methods

A. Naive Bayes[1]

1) Model: The introduction of this model is in line with the specifications in [1]. In the context of text classification, one wishes for a document $d \in \mathbf{D}$ to find $\arg\max_{i \in \{1,\dots,J\}} P(c_i|d)$. The document d can be represented as a tuple of consecutive terms as $d = (t_1, \dots, t_K)$ for a given length K, so that the computation of $P(d \mid c_i)$ is not necessarily trivial, since the space of all possible configurations of $(t_1, \dots, t_K)$ can be unimaginably large. Therefore, the naive Bayes method suggests a conditional independence of all terms t_j, $j \in \{1, \dots, K\}$, on c_i. That is, $P(d \,|\, c_i) = \prod_{j=1}^{K} P(t_j \,|\, c_i)$, thus simplifying

$$\begin{aligned} \arg \max_{i \in \{1,\dots,J\}} P(c_i \mid d) &= \arg \max_{i \in \{1,\dots,J\}} \frac{P(c_i) \cdot P(d \mid c_i)}{P(d)} \\ &= \arg \max_{i \in \{1,\dots,J\}} \frac{P(c_i) \cdot \prod_{j=1}^{K} P(t_j \mid c_i)}{P(d)} \\ &= \arg \max_{i \in \{1,\dots,J\}} P(c_i) \cdot \prod_{j=1}^{K} P(t_j \mid c_i), \end{aligned} \tag{1}$$

wherein the last step is accounted for by the fact that $P(d)$ has the same constant value for every class c_i. In this last term, a lot of conditional probabilities are multiplied, each of which might be very small, possibly resulting in a floating point underflow. It is therefore more convenient to restate the problem by, instead of multiplying probabilities, just adding their logarithms. The class c_i with the highest $\log$ score will still be the most probable one, since $\log$ is a monotonically increasing function. Important to note is that the assumption of conditional independence still leaves us with too many parameters to estimate for the model if every position in the document has its own probability distribution. Therefore we also assume that the occurrence of a term in a document is independent of the position it has taken in this document, given the class. I.e., we state positional independence, an approach which is also adopted by the bag of words model: $P(d_{k_1} = t \,|\, c) = P(d_{k_2} = t \,|\, c)$ where k_1 and k_2 are arbitrary positions in the document d.

The terms in expression (1) can be estimated by their maximum likelihood estimates, i.e., $\hat{P}(c) = \frac{N_c}{N}$, where N_c

[1]In our task of text classification, we assumed the multinomial naive Bayes model.

denotes the number of documents in $\mathcal{D}$ that belong to class c and $N = |\mathcal{D}|$, and

$$\hat{P}(t \mid c) = \frac{T_{ct}}{\sum_{t' \in \mathcal{V}} T_{ct'}}, \tag{2}$$

where T_{ct} denotes the number of times a term t appears in a document belonging to class c, taking into account multiple occurrences in a same document. To handle the anomaly of (2) being zero when the given term t does not appear in any training document belonging to class c, an *add-one smoothing* (also called *Laplace smoothing*) can be considered. That is, $\hat{P}(t \mid c) = \frac{T_{ct}+1}{\sum_{t' \in \mathcal{V}} (T_{ct'}+1)}$. Other smoothing methods exist as well as proposed in [7], however, it was not feasible to apply them manually in Python due to difficulties introduced by the sparsity of the count matrix (see next paragraph).

In our project we implemented the naive Bayes classifier with some additional specifications, inspired by [6]. Each tweet is represented by a feature vector according to the bag of words model. That is, each term in the vocabulary $\mathcal{V}$ is assigned a fixed index in the feature vector, and the feature vector of each tweet contains at each index the number of times that the corresponding word appears in this tweet. In this way a count matrix is constructed with a number of $|\mathcal{D}|$ rows, each row being a feature vector of a certain tweet. Afterwards, the term counts are transformed to term frequencies by dividing every row by its sum of counts, which avoids longer documents to have larger term counts on average than shorter documents, since this is of no use if they belong to the same class. A second transform, called *Term Frequency times Inverse Term Frequency* (tf-idf), is applied by down-scaling terms that appear in many documents as opposed to less occurring words that might be more informative about the class.

One way of improving the performance of the naive Bayes classifier is by also considering n-grams, see [4]. In this way we are sort of relaxing the local conditional independence assumption, by allowing terms to be associated with the previous $n-1$ terms in the same document. From the training set, all instances of sequences of a maximum number of n words are now represented by indices in the feature vector, thus also increasing the number of columns of the count matrix.

Another attempt at enhancing our results is by feature selection [1], with which one wishes to extract a certain subset from the training set so that only this subset will be used to construct features in the classification task. One considers for a class c the utility measure $A(t, c)$ for every $t \in \mathcal{V}$ and we only use these terms of $\mathcal{V}$ that have the highest utility measure value. The rest of the terms is neglected in the classification. In our project, we looked at the χ^2-test as a utility measure, inspired by [3].

2) Results: The resulting accuracy graph for increasing n is shown in Figure 1. For n up until 4 we can observe an initial strong increase, which settles down for larger values of n. The accuracy does not improve much anymore by the time $n = 10$ is allowed; it is then at about 81.4%. When n is too small, our model will not be able to capture sufficient context, whereas for a too large value of n our count matrix will turn out to be very sparse.

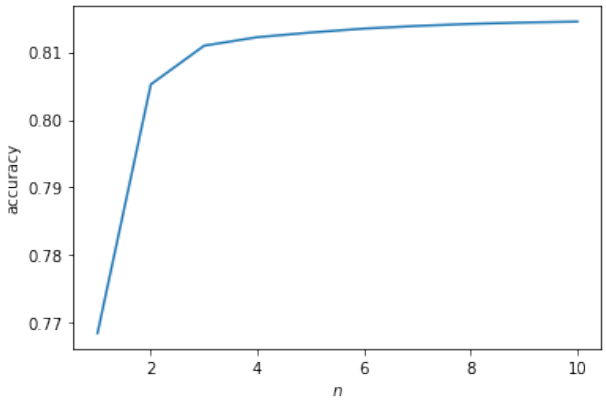

Fig. 1: Accuracy for the naive Bayes classifier for increasing maximum length of the n-grams

We tried out the feature selection using the χ^2-test, for uni- as well as for digrams. In both cases it is clear[2] that retaining a subset of the features will not yield a strictly higher accuracy than in the case where all features are used; however, almost the same accuracy can be obtained with less features, which would enable us to use a smaller count matrix and also offers us faster prediction.

3) Discussion: We can conclude that our best performance will not be attained with naive Bayes. Important to note is that it is not uncommon to come across n-grams in the document space $\mathbf{D}$ that do not even occur during training, explained by the heavy-tailed nature of language. Moreover, the time cost of classifying with naive Bayes also increases for increasing n, another reason because of which we conclude that we must find some other way to improve our results.

The results hint us that, however unrealistic the assumption of conditional independence between terms, the naive Bayes approach does yield a surprisingly high accuracy. An explanation for this seemingly unjust observation is that, although the estimated probabilities of the naive Bayes model are usually far from being accurate, the decisions it makes are quite the opposite, and only the latter objective matters in the objective of text classification. One must be aware that better classification methods exist, although naive Bayes does have some valuable advantages, such as robustness, handling of concept drift, and the fact that training and predicting only require one pass through the data. These favourable properties are interesting to look into, but for this project we preferred to focus on improving our results with other methods.

B. Support vector machine

1) Model: The introduction of this model is also in line with the specifications in [1]. In the context of a support vector machine (SVM), our objective is to maximize the margin of a

[2]The results are available in the `README.md` file.

decision surface with respect to any data point. Logically this implies that the choice of this decision surface is motivated by only a small subset of the data points, which are called the support vectors. A reason to prefer this classification model of maximizing the distance, is that it provides us with a certain classification safety margin. That is, if a small error is inferred upon a sample, it will most probably not affect the classification decision.

Assuming the given data to be linearly separable, the decision surface will therefore be a hyperplane. This hyperplane can be specified by a weight vector $\vec{w}$ orthogonal to the hyperplane and an intercept b such that a point $\vec{x}$ belongs to the hyperplane if and only if $\vec{w}^T\vec{x} = -b$. For an arbitrary feature vector $\vec{x}$ the linear classifier will decide on its class by the function $f(\vec{x}) = \text{sign}(\vec{w}^T\vec{x} + b)$.

For the SVM we also made use of a bag of words model, as was done for the naive Bayes classifier.

2) Results: The feature selection mentioned in the section on naive Bayes was also performed on the SVM with the χ^2 method, but only with unigrams. From our findings[3] we can conclude that SVM performs best when all features are used. The highest achieved accuracy is 80.71%.

3) Discussion: From the results one might conclude that SVM needs all the features of the model in order to yield the highest accuracy.

In the lectures we have studied the kernel trick, which could theoretically also be applied to our case of text classification in order to improve accuracy. That is, as with the n-gram study for the naive Bayes classification, we might induce some local dependencies between tokens. For increasing value of n, we could apply a polynomial kernel trick of degree n. However, since storing a kernel scales quadratically with the number of data points in the data set, this seems like an infeasible task in our case. Another reason to look for other classification models, is that SVM training scales superlinearly with the size of the data set. However, there exist ways to work around these difficulties such as kernel approximation by reformulating the kernelized SVM as a linear SVM.

One last thing to note is that our assumption of linearly separable data might be too hopeful. After some careful observation of the data, one could decide on an appropriate kernel to use to yield a more favourable decision surface. However, as was mentioned earlier, applying a kernel trick on our large amount of data would not be very efficient.

C. Word Embedding Averaging

1) Model: The so called Word Embedding Averaging approach is actually one part of a bigger structure which consists in three steps: Word representation learning, Sentence embedding, Sentiment classification.

In order to classify tweets as positive or negative we may want γ to be a neural network, which maximizes the probability to correctly classify the right label $P(\gamma(d_i) = c_i)$ for each document $d_i \in \mathcal{D}$. The next task which remains to be solved is to embed every document in a representation space $\mathbb{R}^r$, so that r is constant for every document. Neural networks, in fact, need the input space to be constant. One trivial solution would be to set $r = \mathcal{V}$, the size of the vocabulary itself, and use as input x_i for the neural network the bag of words related to each document d_i. Although this method may work in theory, training a neural network in such a big representational space $\mathbb{R}^{\mathcal{V}}$, with $\mathcal{V}$ in the order of hundreds of thousands, would require ages. Therefore, a more clever embedding for sentences must be found which would allow us to have r in the order of the hundreds. One alternative approach could be then to learn the representation of each word in a chosen space $\mathbb{R}^r$, and then compute the sentence embedding by just averaging the word vectors that appear in the sentence. If we call $w_{i,k}$ the word representation for any token t_k appearing in document d_i, then the document representation for d_i would just be $\sum_{w_{i,k} \in d_i} \frac{w_{i,k}}{|d_i|}$. Smarter approaches to compute document vectors can be used, like weighting the embedding of words by the χ^2 value of every word, and use its score as weight when computing the weighted average. When doing so, in fact, we are assuming that words with a high χ^2 are likely to be dependent on the presence of a certain label (positive smile or negative), therefore we may want to give more importance to those words when computing the average of the word embeddings.

In order to compute the word embeddings, we decided to use the *Word2Vec*-algorithm [2]: this method (in its easier version, called continuous bags of words, or *cbow*) tries to predict for every word in every document, which other words will be likely to appear in the same context (the context, set with the hyperparameter $window$, is just defined as the set of $window$ words appearing to the left and to the right of our target word in any document). In order to do so, a shallow neural network is used: the input layer of size $|\mathcal{V}|$ is the one hot encoding representation of the target word. The hidden layer is of size $|r|$ and the output layer (again of size $|\mathcal{V}|$) is the probability of having any word in our vocabulary to appear in the same context as the target word (the probability distribuition is computed using the softmax function). After the training phase, the word embedding is given by the hidden layer of size r (and therefore the model are the weights among the input layer and the hidden layer).

2) Results: Word Embedding Averaging reached an accuracy of 82%, when using the Word2Vec pre-trained model from Google (with word embeddings of size 300), a neural network of 2 hidden layers of size 95, trained for 30 epochs with a batch size of 128. In contrast to our expectations, the χ^2 weight applied when taking the average of the word vectors, didn't work as expected, with the accuracy dropping to 78% when performing cross validation for the hyper tuning of the parameters (two hidden layers of size 60).

3) Discussion: Word embedding by averaging word vectors has proved to work well in practice, although being a very simple model. In contrast, the weighted averaging by the χ^2 value, even though it seemed promising, did not actually perform well. In order to improve the χ^2 weighting, we may

[3] which can also be found in the `README.md` file

try to scale the weights in a nonlinear fashion: χ^2 values live in a wide range (from one to several thousands), and therefore, by applying a nonlinear function like $\log(\chi^2(w_{i,k}))$, we could preserve the information of the dependency among words and labels, but in a more softened fashion.

D. Convolutional neural network

1) Model: Convolutional neural networks (CNNs) entail a sequence of differently sized and shaped filters that reduce the given data into matrices of smaller dimensions. By using these filters, a CNN explores the surroundings of locations in the data. CNNs are primarily known to be robust in image classification, but it turns out that their properties can also be of use in the light of text classification. For text data, a one-dimensional kernel is used: a sentence is represented by a matrix of fixed size, where each row represents a word of the sentence according to the preserved order (possibly followed by some padding) and the columns (in our implementation) represent the features as determined by the *GloVe* word embeddings. (These embeddings were opted for because of online availability of word vectors pre-trained on tweet datasets. Moreover, we wanted to see if these embeddings would yield a higher accuracy than when using *Word2Vec*.)

2) Discussion: We do not deepen into the structure of CNNs in the Model-subsection; we tried several implementations of CNNs as can be seen in the code, but were not able to yield any satisfying results. The network always overfits after some epochs and does not yield a better testing accuracy than when using an LSTM (see next section). Therefore, we preferred to focus on the latter type of network.

E. LSTM

1) Model: We achieved the best results so far by using a Long Short-Term Memory (LSTM). LSTMs were developed in order to overcome issues of recurrent neural networks linked to the vanishing gradient and they have always been performing NLP tasks incredibly well. They can keep memory of informations even for long periods of time. As shown in Figure 2 their structure is built in a way that takes as input both the output and the state of the node that logically 'precedes' it. The sigmoid function σ is applied to the input to decide how much of each component should be let through and the $\tanh$ activation function is used to update the cell state.

In our case this method seemed to be very attractive since our goal, that was to classify sentences with respect to their sentiment, was about catching the different contexts in which words are used.

Therefore we started with data cleaning as mentioned in section III and then we focused on arranging the sentences in such a way that is most suited for automatic computation. We thus built a vocabulary $\mathcal{V}$ of unique words (actually tokens) and we mapped each of them to an integer index. Then, keeping in mind that the input of a neural network should be homogenous in its dimensions, we padded all the sentences that were shorter (in terms of number of tokens) than a certain value that we chose to be the maximum length among all the sentences in the training set. We then ended up with a matrix of $\mathcal{N}$ rows and $\mathcal{F}$ columns, $\mathcal{N}$ being the number of inputs and $\mathcal{F}$ the number of features we extracted so far from each sentence. Here is where the LSTM comes into action. We wanted to perform a transformation that is the following (from a dimensional perspective):

$$\mathcal{N} \times \mathcal{F} \to \mathcal{N}.$$

To do this we built an LSTM using the `keras` library for the implementation with the following layer composition:

1) Embedding Layer: converting our word tokens (previously encoded into integers) into embeddings of size 256,
2) LSTM Layer: using the $\tanh$ activation function within layers and two dropout coefficients on the inputs and on the hidden states to avoid overfitting,
3) Softmax Activation Layer: transforming the outputs into the desired form of a $\mathcal{N} \times 2$ result.

Eventually, the predictions were calculated by taking for each row of our output the maximum value in the row, that according to its position, represents a prediction of a positive or negative sentiment.

2) Results: The best achieved result with the LSTM is an accuracy of 84.6%.

3) Discussion: We tried to improve our results in many ways. The one in which we were more confident was by enlarging our dataset in order to avoid a fast overfitting of the neural network. To this end, we found on the web a big dataset of nearly 1.5M labeled entries. We cleaned it in such a way that the final result was exactly equivalent to the one that had originally been given to us. But surprisingly the results did not improve, actually in some cases they got worse.

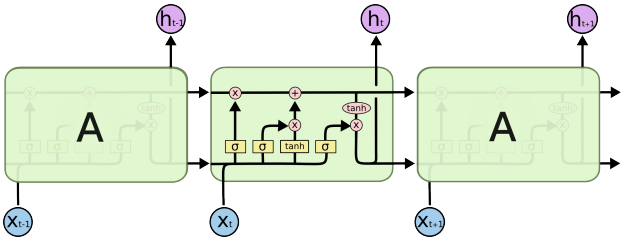

Fig. 2: General structure of an LSTM [5]

V. Summary

In our search for an effective learning function to binary classify the given emotionally tagged tweets, our best result was obtained when using LSTMs. Other neural nets such as CNNs using word embeddings such as the ones provided by *Word2Vec* and *GloVe* also yield satisfiable results, however not as good as the LSTMs. Simpler models such as naive Bayes and SVM do not score much worse than neural nets, and might therefore be preferred when one has less time available.

VI. Note

Mentions of the used libraries can be found in the `README.md` file of our code (see the installation instructions).

REFERENCES

[1] C. D. Manning, P. Raghavan, and H. Schütze. "An Introduction to Information Retrieval". Cambridge University Press. 2009.

[2] Tomas Mikolov et al. "Distributed Representations of Words and Phrases and their Compo sitionality". In: *Advances in Neural Information Processing Systems 26*. Ed. by C. J. C. Burges et al. Curran Associates, Inc., 2013, pp. 3111–3119. URL: http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf.

[3] *ML-Fundamentals - Exercise - Mutual Information for Feature Selection*. https://gitlab.com/deep.TEACHING/educational-materials/blob/master/notebooks/natural-language-processing/text-classification/exercise-mutual-information.ipynb. Accessed: 19/12/2019.

[4] F. Peng. "Augmenting Naive Bayes Classifiers with Statistical Language Models". Computer Science Department Faculty Publication Series, 91. 2003.

[5] *Understanding LSTM Networks*. http://colah.github.io/posts/2015-08-Understanding-LSTMs/. Accessed: 19/12/2019.

[6] *Working With Text Data*. https://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html. Accessed: 19/12/2919.

[7] Q. Yuan, G. Cong, and N. M. Thalmann. "Enhancing Naive Bayes with Various Smoothing Methods for Short Text Classification". 2012.