Project 2, Machine Learning

Tom Suter, Matteo Battilana, Hedi Driss
Pink Panthers, EPFL Lausanne, Switzerland

***Abstract*—The aim of this project is to build a model that can predict if a tweet has a positive or negative connotation, a task that goes under the name of *Twitter sentiment classification*. In the following we will present the methods we use for treating the input tweets and preparing them for the classification.**

I. Introduction

The tweet sentiment analysis is an interesting problem of text classification whose intent is to determine if a tweet is carrying a positive or negative message. This task has several applications like obtaining an overall feedback of a product launched in the market or examining the social impact after a particular event has occurred (like the USA presidential election). Tweets are convenient to deal with as thanks to the *hashtags* one can easily filter them by topics.
The analysis is made of two parts, the *features engineering* in which we prepare at best the data for the classification, i.e. we get the tweets in input and provide them of a meaningful representation in a vector space. The second part consists in building and training a model for classifying these points.

II. Features Engineering

This is an essential part of the analysis. We start by saving each tweet in a *list* of strings, whose elements are the words that forms the sentence. We also use the *nltk* library to separate words in a better way, so that words as *'yes!'* are split as *'yes'* , *'!'*. We also remove words and punctuation that are not giving any sentiment or utility to the tweet. Once we have those simplified tweets, we count the occurrence of each word and delete those with less than 5 occurrences since words appearing rarely don't give relevant information and would just make the computation heavier. The list of all the words together with their occurrence, make our **vocabulary**. At this point we are ready to embed these words in a vector space. In order to do so we consider two methods: *GloVe* and *Word2vec*.

A. GloVe

The main idea of GloVe algorithm [1] is that the scalar product of word embedding vectors should be related to the probability of these words occurring together. Geometrically speaking the higher this probability is the more parallel the two embedded vector will be. After some mathematical assumptions based on:

- linearity of the space
- symmetry by words interchange

one can write the scalar product within the embedding vectors $\vec{w}_i, \vec{w}_j \in \mathbb{R}^D$ of the words i and j as:

$$\vec{w}_i \cdot \vec{w}_j = \log C_{ij} + B \tag{1}$$

Here B is a constant while C_{ij} is the (i, j) entry of the *co-occurrence* matrix which counts in how many tweets the two words i and j occur together. Clearly it's a symmetric matrix and the diagonal represents the number of times that the same word appears in a tweet at least twice. We know the matrix C_{ij} and we want to estimate the embeddings then the natural thing to do is to minimize a cost function $J(\vec{w})$ for obtaining the best estimate for the vectors:

$$J\left(\vec{w}\right) = \sum_{i,j} f\left(C_{ij}\right)\left(\vec{w}_i \cdot \vec{w}_j - \log C_{ij}\right)^2 \tag{2}$$

in which we take $B = 0$. The f is a weight function necessary for avoiding divergences when $C_{ij} = 0$ and for penalizing small entries of the matrix (which carry small information). A good choice for it is:

$$f\left(x\right) = \min\left\{\left(\frac{x}{100}\right)^{3/4}, 1\right\} \tag{3}$$

Then by building the co-occurrence matrix C_{ij} and minimizing (2) we get the embeddings for the words.
We set the features space dimension to $D = 20$.

B. Word2vec

This word embedder makes use of shallow neural networks models made of only one hidden layer. One can adopt two specular architectures, the *CBOW* (Continuous Bag Of Words) and the *Skip-Gram*. The first predicts the most likely word given a set of context words while the second does the opposite, i.e. it predicts the most likely set of context words given a word. We used the *CBOW* architecture as from [2] it results to be the fastest one.
As already mentioned this model is a three-layer neural net (see fig (1)). If we let V be the size of the vocabolary, then the input words must be represented as one-hot encoded vectors i.e. vectors who are zero everywhere except in a dimension in which they are one. Then a tweet of C words is build as the sum of its words vectors; this representation falls under the name of *bag of words*[1].

[1]This expression comes from the fact that we don't care of the actual order of the words in a sentence but just on their frequency.

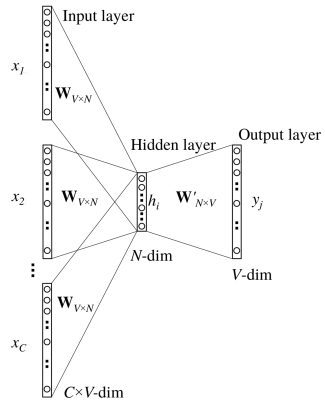

Figure 1. Graphic representation of the CBOW embedder [3].

Consequently the input tweet is a sparse vector $\vec{t} = \sum_{n\in C} \vec{x}_n$ of dimension V. The entries of the hidden layer, whose dimension is N, are computed thanks to the weight matrix $\underline{\underline{W}}$, of size $V \times N$:

$$\vec{h} = \underline{\underline{W}}\vec{t} \tag{4}$$

Notice that no activation function is taken. The output layer has the same dimension (V) as the input layer and it provides, for each word of the vocabulary, the probability to be linked to the context words (i.e. the tweet). These conditional probabilities are obtained from:

$$y_i = P\left(\omega_i|\vec{t}\right) = \phi\left(\vec{W}'_i \cdot \vec{h}\right) = \frac{\exp\left(\vec{W}'_i \cdot \vec{h}\right)}{\sum_{k=1}^{V} \exp\left(\vec{W}'_k \cdot \vec{h}\right)} \tag{5}$$

where $\underline{\underline{W'}}$ is a $V \times N$ weight matrix and the activation function is (in our case) the **softmax function**.
The best weights will be the ones maximizing this probability, hence we can define a cost function as minus the logarithm of the probability:

$$E\left(\underline{\underline{W}}, \underline{\underline{W'}}\right) = -\vec{W}'_i \cdot \vec{h} + \log\left[\sum_{k=1}^{V} \exp\left(\vec{W}'_k \cdot \vec{h}\right)\right] \tag{6}$$

For this word embedder we use the *gensim* library and we set the feature space dimension to $D = 300$. For minimizing the cost function we use the stochastic gradient descent whose learning parameter η decreases linearly during the training process.

III. Classification

For the classification we try first some linear techniques as *logistic regression* and *support vector machine*, lastly *neural networks*.

A. Logistic regression

In the logistic regression the cost function to minimize is:

$$\mathcal{L}\left(\vec{\omega}\right) = \sum_{n=1}^{N} \log\left[1 + \exp\left(\vec{\phi}_n \cdot \vec{\omega}\right)\right] - y_n \vec{\phi}_n \cdot \vec{\omega} + \frac{\lambda}{2}|\vec{\omega}|^2 \tag{7}$$

where $\vec{\omega}$ are the weights, $\vec{\phi}_n$ is the n-th observation expressed in some linear basis and y_n its label.
First we do a grid-search for looking at the best regularization parameter λ. We find that the score is almost insensitive on λ so we set $\lambda = 1$.

We use a polynomial basis up to degree four and the stochastic gradient descent for minimizing (7). Around ~ 200000 tweets are available for training and testing the model and the dataset is folded in five parts for the cross-validation. In I the scores we get using the *sklearn* library.

Polynomial degree	Score GloVe	Score Word2vec
1	0.597 ± 0.008	0.695 ± 0.003
2	0.598 ± 0.008	0.712 ± 0.002
3	0.600 ± 0.008	0.714 ± 0.002
4	0.599 ± 0.008	0.715 ± 0.002

Table I
Results for logistic regression classifier with l^2 penalization. We train and test in a 5-folded dataset of ~ 200000 samples.

B. Support Vector Machine

In the SVM the cost function to minimize is:

$$\begin{aligned}\mathcal{L}\left(\vec{\omega}\right) &= \sum_{n=1}^{N} \max\left\{0, 1 - y_n \vec{X}_n \cdot \vec{\omega}\right\} + \frac{1}{2}|\vec{\omega}|^2 \\ &= \sum_{n=1}^{N} \max\left\{0, 1 - y_n \vec{\kappa}_n \cdot \vec{v}\right\} + \frac{1}{2}|\vec{\omega}|^2\end{aligned} \tag{8}$$

where as before $\vec{\omega}$ are the weights, $\vec{X}_n$ is the n-th observation and y_n its label. In the second row we explicitly use the *kernel trick* where $\vec{\kappa}_n$ is the n-th row of the kernel matrix $\underline{\underline{\kappa}}$ and $\vec{v}$ is a vector such that $\vec{\omega} = \underline{\underline{X}}\vec{v}$. We consider a l^2 regularization for penalizing overfit and used *stochastic gradient descent* for minimizing (8). As the SVM is computationally expensive, we select 10% of the available tweets (~ 200000) for training and testing the model. Lately the dataset is folded in five parts for the cross-validation. Below the results obtained with methods from *sklearn* using a radial kernel $\kappa_{ij} = \exp\left(-\gamma|\vec{X}_i - \vec{X}_j|^2\right)$

Kernel parameter γ	Score GloVe	Score Word2vec
0.00005	0.613 ± 0.003	0.691 ± 0.007
0.001	0.604 ± 0.003	0.707 ± 0.008
0.1	0.616 ± 0.004	0.700 ± 0.007

Table II
Results for SVM with radial basis kernel. We train and test in a 5-folded dataset of ~ 20000 samples.

Here the results we obtain using a polynomial kernel $\kappa_{ij} = \left(\vec{X}_i \cdot \vec{X}_j\right)^d$, up to degree four.

Polynomial degree	Score GloVe	Score Word2vec
1	0.605 ± 0.003	0.699 ± 0.008
2	0.597 ± 0.005	0.706 ± 0.009
3	0.564 ± 0.006	0.72 ± 0.01
4	0.507 ± 0.003	0.67 ± 0.01

Table III
Results for SVM with polynomial basis kernel. We train and test in a 5-folded dataset of ~ 20000 samples.

C. Neural Networks

A neural network is a non-linear model for regression or classification problems. It consists of an *input layer*, an *output layer* and a set of $K \geq 0$ *hidden layers*. Each layer consists of N_l neurons to which we will refer in the following as $x_i^{(l)}$ where $i = 1, \ldots, N_l$ and $l = 0, \ldots, K+1$. All these neurons are linked by a non-linear function applied to a standard "weights summation" and a bias term:

$$x_i^{(l)} = \phi\left(\sum_{j=1}^{N_l} W_{i,j}^{(l)} x_i^{(l-1)} + b_i^{(l)}\right) \tag{9}$$

This type of neural network is called **dense** because neurons of adjacent layers are fully connected.
The strength of a neural network is that not only we can learn the best weights for the classification (K-th hidden $\rightarrow$ output), but also the best weights for preparing the data for this task (input $\rightarrow$ 1st hidden $\rightarrow \cdots \rightarrow K-1$-th hidden). So a neural network model has several tunable parameter summarized in table IV.

Parameters
Number of hidden layers (K)
Number of neurons in each layer (N_l)
Activation function (ϕ)
Learning rate of the SGD (α)
Batch size of the SGD (B)
Regularizer parameter for the l^2 penalizer (λ)

Table IV
Main dofs of a neural network.

Formally it had been proven that a single hidden layer is enough for 1-dimensional data and two are enough for data in $D \geq 2$ dimensions. So far there are no other formal rules for the choice of the parameters so everything relies in a grid-search.

For our analysis we consider $K = 2$ hidden layers as we have multidimensional data. The number of neurons are estimated after a grid search; first we explore a wide range of sizes and then a more precise search around the best candidate.

1) GloVe: For the GloVe representation we find the following parameters after a grid-search.

Neurons number	$(N_1, N_2) = (15, 5)$
Activation function	$\phi \rightarrow$ sigmoid
Learning rate	$\alpha = 0.1$
Batch size	$B = 20$
Regularizer parameter	$\lambda = 10^{-4}$
Score	61.22%

Table V
Neural network parameters for GloVe representation.

In fig(2) and fig(3) two graphs exhibiting the grid-search done on the number of neurons in the first hidden layer and regularization parameter λ.

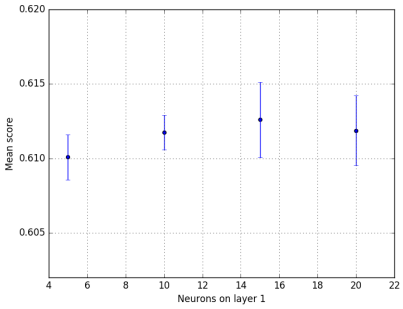

Figure 2. Score versus the number of neurons of the first hidden layer for GloVe embedding, using 50% of the data with a 3-fold cross validation.

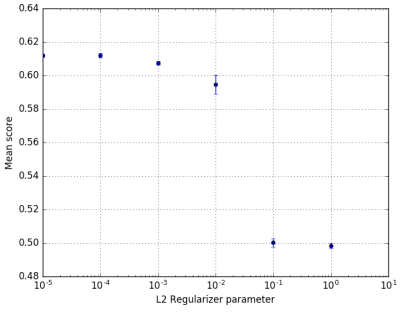

Figure 3. Score versus the value of the regularizer parameter for a GloVe embedding, using 50% of the data with a 3-fold cross validation.

2) Word2vec: We apply similar techniques for a *word2vec* representation but we don't obtain satisfactory results.

D. Convolutional Neural Networks

In order to apply convolutional neural nets (CNN), we need to build a matrix for each tweet, as illustrated in fig 4. In order to do so we consider the embedding to each word (the rows of the matrix). Since the tweets don't have the same length, we simply add a token (called PAD) in the end to have uniform tweet length (map on the longest). Then we are ready to apply CNN.

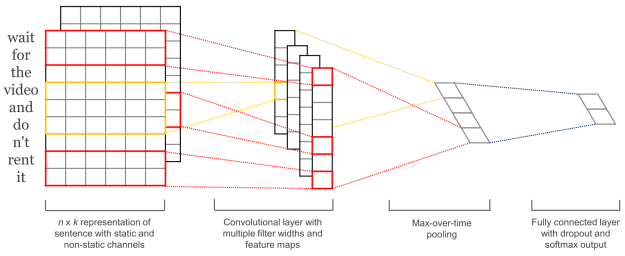

Figure 4. Graphic representation CNN similar to our model [4]

Here we follow the procedure of [4]. This particular setting contain an embedding layer. Indeed starting from the GloVe embedding, this layer is able to return a better word representation. It is followed by a drop out which deactivate randomly a percentage of the neurons. The third layer is essentially a filter that takes in input the data, groups them and return the max of these groups. As a consequence we have a reduction of the dimensionality. This process is applied a second time between the third and fourth layers.

This model is said to achieve great performance for small sentences sentiment recognition [4]. With it we obtain our best result, using only the small training set and 50 epochs.

IV. Discussion

- In the text preprocessing part we remove punctuation and neutral words like 'the', 'and', 'is'. Certainly this makes the vocabulary lighter and the computation slightly faster but on the other side we don't see significant improvements on the embedding quality. We believe that extending the vocabulary to some *n-gram* would increase the quality (despite spending more time for the computations). Indeed already *2-grams* like 'not good' encode a negative meaning that it is not well captured by the embedders we use.
- The logistic regression seems to be too poor for such a task. Indeed independently on the penalty parameter, on the degree of the polynomial basis and on the embedder we obtain the modest score of $\sim 60\%$.
  On the other side in the SVM we find an interesting correlation between the score and the embedder used. Indeed, under the same conditions, *word2vec* guarantees in average a score of ten percentual points higher than GloVe.
- Nonlinear classifier as neural networks are much more qualified for this kind of tasks. Indeed, if well trained, they can add optimal features that make easier the classification in the output layer. Neural networks have several degrees of freedom that one can play with and this makes them a very elastic model. Conversely this freedom is a bit frustrating since there isn't a background theory driving on an optimal choice of the parameters. We believe this is the main reason we don't get good results with neural networks ($\sim 60\%$), i.e. we haven't found the optimal parameters for this task.
  On the other side following the advices of [4] we can build an efficient convolutional neural network that provides us the best score ($\sim 76\%$), even training on the small dataset.

V. Conclusions

In this project we tested many different techniques (both in the processing and classification parts) for doing a *tweets sentiment analysis*. From our analysis it turns out the best method to use is a convolutional neural network since it improves the word embedding in its hidden layers. Overall we don't obtain exciting results and we think we should have done a better text preprocessing, maybe using the *skip-gram* architecture of word2vec and a broader grid-search on the tunable parameters.

References

[1] C. D. M. Jeffrey Pennington, Richard Socher, "Glove: Global vectors for word representation," 2014.

[2] G. C. J. D. Tomas Mikolov, Kai Chen, "Efficient estimation of word representations in vector space," 2013.

[3] X. Rong, "word2vec parameter learning explained," 2016.

[4] K. Yoon, "Convolutionnal neural networks for sentence classification," 2014.