Emojify - Sentiment Analysis of Text

Mincato Emanuele
emanuele.mincato@studenti.unipd.it

Joi Berberi
joi.berberi@studenti.unipd.it

1. Introduction

The aim of this work is to find the most related sentiment of a written sequence and map it into an emoji, this is a problem of text classification. In this case there are only five emoji: heart, baseball, smile, disappointed, fork-and-knife. Our project is divided into two main parts. In the first one we implement most of the model see during the course (Logistic Regression, Random Forest, Support Vector Machine, K-Nearest Neighbors) while in the second one we build a Neural Network, a Deep NN and also we implement a Long short-term memory (LSTM) NN because it can keep track of the dependencies between sequence elements. We decided to divide the project into these two parts because we use different working methods.

2. Dataset

First we need to look at the data that are provide to us. There are two dataset containing 132 phrases, 100 from train-set and 32 from validation-set, and each sentence is labeled with the respective emoji class. We checked the phrases and we find out that are 'clean' data, in fact there are no grammatical errors and also we observe that there are no punctuation marks. In these sentences there are 260 different words and we count them to see the word frequency. As we can see from the figure below we are dealing with very skewed data, so the accuracy may be not the only metrics we need to look for to choose the best model.

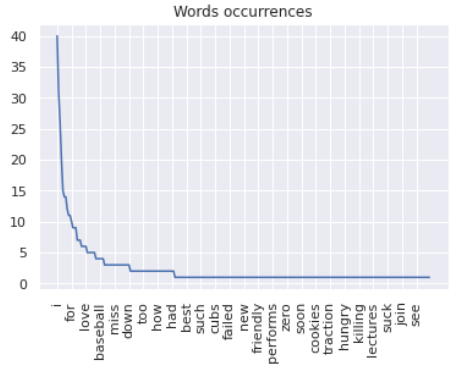

Figure 1. Word frequency

Also are provide to us two nunpy array containing a vector representation of the phrases, pre-computed from the GloVe model. The total dimension of the pooled array is 132x250 because each word is represented by a 25 dimension vector and each phrase is padded to the phrase of maximum length, in this case 10.

2.1. Preprocessing the data

In addition to the GloVe representation of the words we decide to use two other text representation. The first one is *bag-of-words*; when we use this methods we discard most of the structure of the input text, we create a vocabulary of all words that appear and finally we count how often each word appears in each sentence. To do this we transform the sentences using the method *CounterVectorizer*, already implemented in sklearn, which gives us the representation stored in a sparse matrix (it only stored non-zero entries). The second representation we use is the *term frequency-inverse document frequency(tf-idf)*. The intuition of this method is to give high weight to any term that appears often in a particular sentence but not in many sentences. If a word appear often in a "group" of phrases it is likely to be very descriptive for a particular class(emoji). We implement this methods using *TfidVectorizer* which takes in the text data and do both the *bag-of-words* feature extraction and the tf-idf transformation. We have to specify that tf-idf scaling is a purely unsupervised technique. To improve these two text representation we can discard uninformative words that are too frequent in English language to be informative. To do this we remove from the vocabulary, made by *bag-of-words*, a specific list of stopwords already implement in sklearn. Furthermore to get better results we try lemmatization, the purpose of this type of text-processing is to 'normalize' words, avoiding adding words with the same semantics to the vocabulary, for example lemmatization can normalize "worse" to "bad". Thank to stopwords we discard 75 features (words) while using lemmatization we discard 92 features. Instead for the GloVe representation we don't use the previous preprocessing steps. The problem is that we are working with pre-trained embeds and therefore, by using standard preprocessing, we could lose valuable information which would help our models to classify correctly the sentences.

3. Models and Experiments

For the first part we use a very greedy approach, in fact we tested different models using all the three previous text representation. We know this may not be the best way to work, but we observed that the models are fast to compute and so we could run a lot of simulation without spend to much time.

We started building the four models (Log. Regression, Rand. Forest, SVM, k-NN) using the already implemented algorithm in scikit learn and for each of these algorithms we used a similar approach. To choose the best hyperparameters we implemented a pipeline thanks to which we could make a GridSearchCV. We used pipelines because is a way to simplify the process of building chains of transformations and models. The GridSearchCV implements two different process, GridSearch and cross-validation, in the same time. GridSearch is a method that allow us to try all possible combinations of the parameters of interest, in other words the possible hyperparameters. Cross-validation, on the other hand, is a statistical method of evaluating generalization performance that is more stable and thorough than using only one training-set and one validation-set. We used a five fold cross-validation so, for each possible combination of parameters, the pooled data of train and validation are divided in five folds, four for training and one for testing, and are used to train multiple models. To summarize, GridSearchCV allows us to choose the hyperparameters that maximize the average accuracy of five different models that are trained using different parts of pooled data. We found the best parameters for each type of models and we trained them using the pooled data, after we tested them in a made-up test set built by us. The test set consist of 34 phrases with the respective label (index of emoji) and also the target classes are balanced (6 sentences for the first emoji and 7 for each of the others).

In the table below are reported the results of our experiments considering only the best model for each type. In the second column are stored the values of the average accuracy score of cross-validation while in the last column are stored the accuracy in the test set.

Method	cross-val (%)	test-set (%)
Log. Reg.	68.06	70.59
Rnd For.	62.67	61.76
K-NN	59.91	61.76
SVM	68.06	70.59

Table 1. Accuracy scores for the best models

From the table we can see that, in most of the cases, the models get an higher accuracy in the test set, this is due to the fact that we choose 'text' phrases that are easy to classify in the right class. The algorithms that give the worst results are K-NN and Random Forest. This two methods get also the same test accuracy but if we look at the confusion matrix, reported only on the notebook, we can see that they made different mistakes. Random Forest is a time consuming method because we decide to average 500 decision tree to obtain a more robust algorithm by reducing overfitting, a big issue of decision tree. The K-NN instead works badly because we are dealing with dataset with many features (number of words) and especially, when we use bag-of-words or tf-idf representations, the features are 0 most of the time (*sparse dataset*). Thanks to the GridSearchCV we found out that the best numbers of neighbors for K-NN is three. On the opposite side Logistic Regression and Support Vector Machine are the best algorithms among these four. From Table 1 we can see that GridSearchCV lead the two models to the same accuracy in cross-validation and test set. To check if the models made different mistakes we decide to compute the confusion matrix (Figure 2). We reported here just one matrix because they are identical.

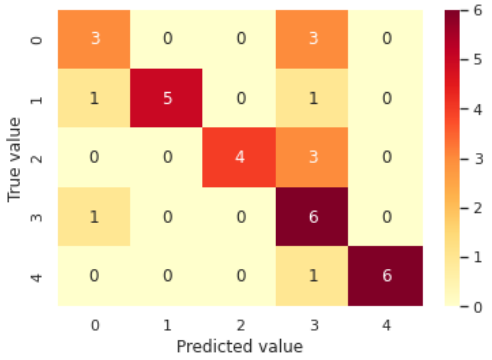

Figure 2. Confusion matrix

The only main difference between these two algorithms is in the text representation. Both of them are obtained using TfidVectorizer but with a different value for the parameter n-gram. This variable regulate the range of tokens that are considered as features by *bag-of-words* feature extraction. For Logistic Regression this parameters is equal to (1,1) instead for SVM is equal to (1,2); so in the first case we just consider single word as features while in the second case we consider also pairwise words.

3.1. Neural Networks

Because training the neural networks take longer time than the other algorithms, in this second part, we had to make *a priori* decisions. We decided in fact to use only the GloVe text representation to train our models. We also need to provide the desired output in the correct form, so we use the *one-hot-encoding* representation, such that we replace a categorical variable with five features that can have the values 0 or 1. We first start training a simple neural network with only one hidden layer. To tune the hyperparameters,

like the validation function and the dropout, we decided to choose them one by one. We try different values for one hyperparameters while the others remain constant and after we plot the accuracy and loss function, in function of the epochs, so we can choose the best ones. To better evaluate the generalization performance of our model we pooled together the GloVe representation of training and validation set and, instead of using the cross-validation (used previously), we use the stratified k-fold cross-validation with seven folds. In stratified k-fold cross-validation the data are divided such that the proportions between classes are the same in each fold as they are in the whole dataset. We decided to use this technique because we are dealing with unbalanced datasets and so we don't want to introduce some kind of bias from data. The results of the experiments in this second part are reported in Table 2. In the second column is stored the values of the average accuracy score of stratified cross-validation while in the last one we reported the respective values of standard deviation.

Method	strat. cv (%)	$\pm$ sd (%)
Naive NN	62.78	10.93
Deep NN	63.58	6.30
LSTM	68.17	9.76

Table 2. Accuracy score and standard deviation

To increase the average accuracy we built a deep neural network adding another hidden layer to the previous model. We tuned the hyperparameters as we did previously but, in this case, we also need to find the best architecture for the model. After several trials we found that the best architecture is composed of 128 neurons in the first hidden layer and 32 in the second. Below, in Table 3, are reported the best hyperparameters for the "naive" neural network and for the deep neural network.

Method	Architecture	Activation fun.	Dropout
Naive NN	[64]	relu	0.35
Deep NN	[128, 32]	sigmoid	0.25

Table 3. Hyperparameters of neural networks

Finally as last model we trained a Long-Short Term Memory networks. The LSTM is a very powerful type of recurrent neural network, the model in fact don't use only the words as input but it also encodes the dependencies between them. This can be helpful in understanding the meaning of a sentence better which could lead to higher accuracy. A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell can keep track of the dependencies between sequence elements and the three gates regulate the flow of information into and out of the cell. To improve the model performance we also define two Bidirectional LSTM layers, with 64 hidden nodes in the first layer and 32 in the second. Bidirectional LSTMs train two LSTMs instead of one. The first on the input sequence as is and the second on an inverted copy of the input sequence. This can provide additional context to the network and lead to faster and more complete learning of the problem. Compared to previous models this is the most complicated and the slowest to train but it is also the one that leads to the best accuracy results, as we can see from Table 2.

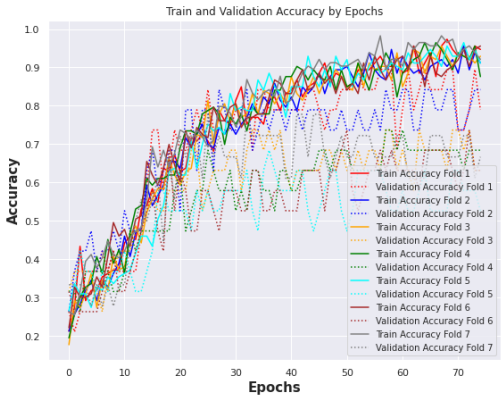

Figure 3. stratified k-fold cross-validation of LSTM

4. Conclusion

Finally we decide to use the best models of the two parts of our works to make the prediction on Kaggle test sentences. Since there is no difference in results between the logistic regression and the support vector machine we decided to use the first one. We trained both models, logistic regression and LSTM, using the sentences from training and validation set. Using the logistic regression model, Kaggle gives us 71.43% of accuracy in private score while using the LSTM model we reach a surprisingly high accuracy of 75%, in both private and public score. To conclude our work we want to highlight the problems we have encountered during our experiments. All the model suffer in different ways of overfitting, as we can see also from Figure 3 where the train accuracy continue to increase while the validation accuracy stop increasing after 30/40 epochs. Another problem is related to the way models deal with words never seen before, that may appears in test sentences. This is a huge problem especially in *bag-of-words* and *tf-idf* text representation where the unseen words are discarded, because are not included in the vocabulary during the training phase. Definitely a possible solution to solve these problems is to increase the number of labeled sentences in the training set.