Data Science Lab: Process and methods
Politecnico di Torino

Project report
Student ID: s275782

Exam session: Winter 2020

1 Data exploration (max. 400 words)

In the phase of Data exploration using the command *df = pd.read_csv('./development.csv')* of the Pandas package I loaded the development dataset and then I looked at its characteristics.

With the command *df.info()* I discovered that there are *28754 entries* and *2 columns*. One column contains the text of the review and the other contains the class, which can be positive or negative. For each column I saw that there are *28754 non-null objects* and therefore consequently there are no *NaN*.

I did a similar analysis for the *evaluation.csv* file. I loaded the evaluation dataset in *df_eval*. I saw that there are *12323 entries* and *12323 non-null objects* and therefore there are no *NaN*.

With the command *df['class'].unique()* I have verified that in development dataset there are only two classes, one for positive elements and one for negative elements. 67.93% of the samples are in the "*pos*" class (19532 elements), while 32.07% are in the "*neg*" class (9222 elements).

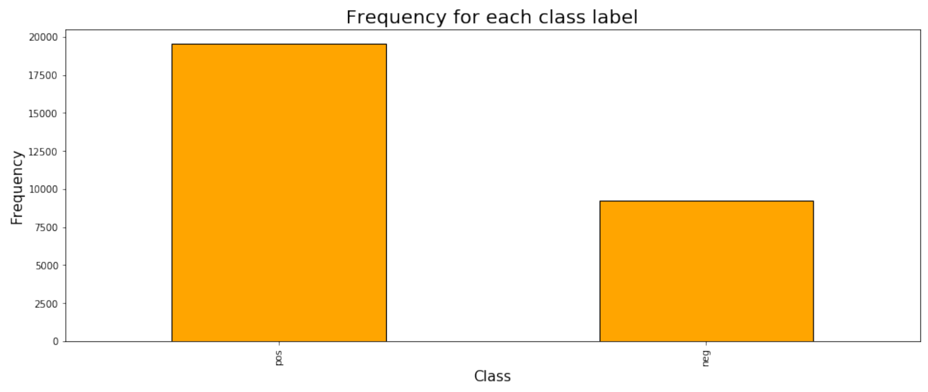

Figure 1: Frequency for each class label

Analyzing the number of characters in each review, I observe that the average in *df* is 701 characters and the average in *df_eval* is 699 characters.

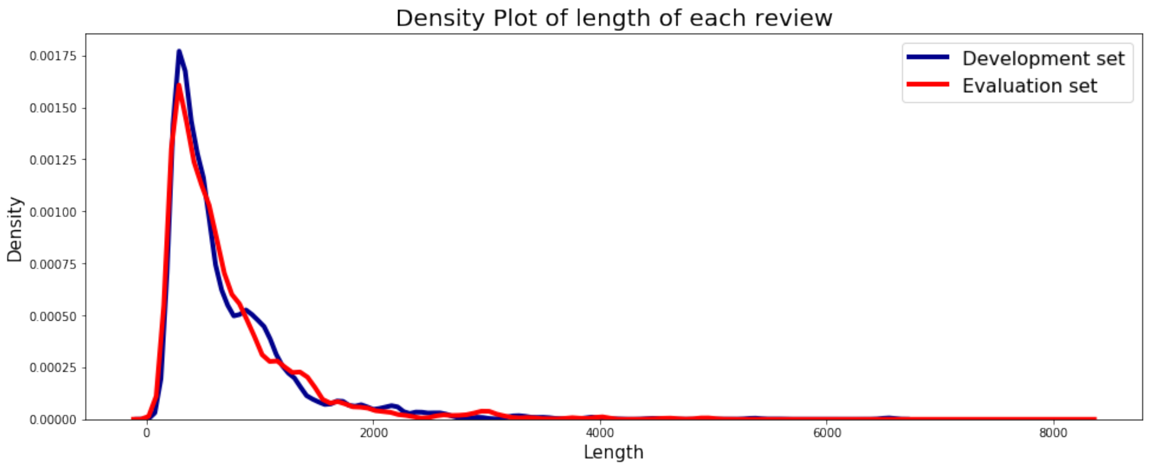

Figure 2: Density Plot of length of each review

Therefore, given the average and the density plot (Figure 2), I can confirm that the distribution of the sets is similar.

Most reviews have fewer than 1500 characters, in fact the 90-percentile is 1345 characters. The graph below (Figure 3) shows the density of each class by analyzing only the reviews with less than 1345 characters.

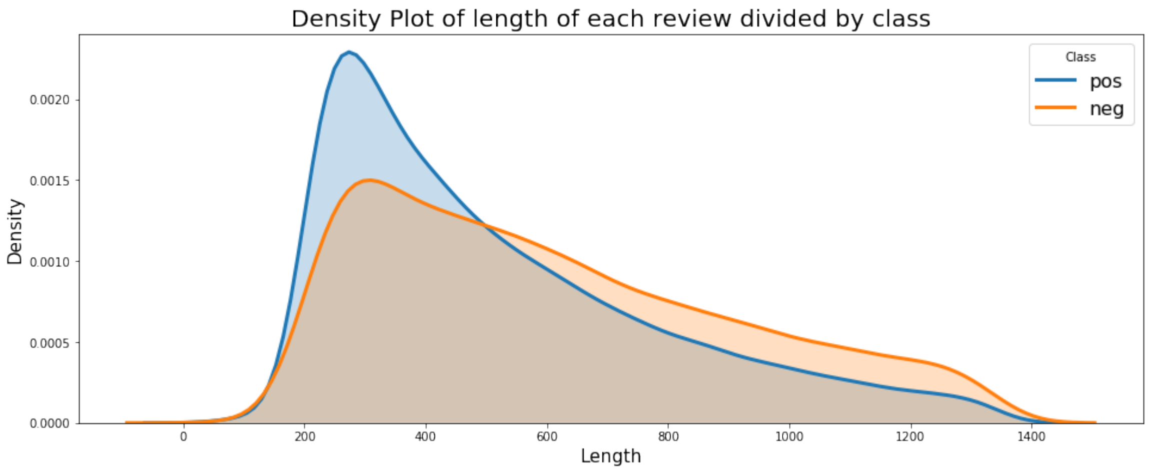

Figure 3: Density Plot of length of each review divided by class

2 Preprocessing (max. 400 words)

During the preprocessing phase I have to transform the review dataset into a matrix usable by mathematical tools.
I used *TfidfVectorizer* [10] to convert the collection of reviews to a matrix of TF-IDF features.
I created a tokenizer that tokenizes the words, but being that it is not a formal text there could be words written not in the correct language and therefore they are to be corrected:

- the symbols at the beginning and end of the word are eliminated,
- if there are symbols in the middle of the word it is divided into two words,
- in order to have words composed only by letters and numbers, the tokenizer eliminates emojis [13], dashes (as in wi-fi to make it uniform to those who write wifi), special symbols and non-Latin characters [11],
- if the words contain both numbers and letters, the numbers are deleted because the people may have written "casa15" due to a typo,
- the word is lemmatized so as to obtain the word in a generic format,
- the lemmatized word is stemmed,
- words composed by less than 2 or more than 16 characters, punctuation marks, numbers and words written in non-Latin characters are eliminated.

In the *TfidfVectorizer* parameters there is *strip_accents='ascii'* so that it removes accents and performs character normalization.
In *stop_words* parameter I used a list that contain stop words. I got this list by joining two sets of stop words from two different packages. In order to identify correctly the stop words in the tokenized text, also the list of stop words have to be processed by tokenizer and *strip_accents* in advance.
I used the *ngram_range=(1,3)* parameter so that the *TfidfVectorizer* analyzes unigrams, bigrams and trigrams.

The resulting matrix has millions of columns. The use of algorithms to decrease the number of components did not lead to an improvement in the forecast of the class. For example, the TruncatedSVD algorithm (version of the PCA for sparse data) was tested. Even Nystroem used to approximate a kernel map did not lead to improvements.

Using *y = df['class'].astype('category').cat.codes* I have converted the classes into numbers so that all algorithms work, even those that require class labels to be numeric.

3 Algorithm choice (max. 400 words)

I divided the development set, using the *train_test_split*, into a set for the model train and one for the model test. Since the classes are unbalanced, I splitted the data in a stratified fashion.

I used the *GridSearchCV* [2] on the train set to find the best parameters for each classification algorithm and then I tested them on the test set analyzing the performances with the *f1_score*:

- **DecisionTreeClassifier** [1]: I tested it with various choices of the following parameters: *criterion*, *splitter*, *max_features* and *class_weight*, but the f1 score on the best estimator is less than 0.85.
- **RandomForestClassifier** [7]: I tested it with various choices of the following parameters: *n_estimators*, *criterion*, *max_features* and *class_weight*, but the f1 score on the best estimator is between 0.88 and 0.93. It makes sense that there is a higher score than the previous algorithm because many trees were used here instead of just one.
- **MLPClassifier** [6, 12]: This algorithm is not the best because it has a very long training time, it is also weakly scalable and it is not an interpretable model. Since our dataset has an matrix of TF-IDF features with millions of columns, this algorithm generates memory problems.
- **KNeighborsClassifier** [3]: The KNeighborsClassifier is not effective because it measures the distance between vectors. To be close, two vectors must have many identical words, but two reviews cannot be so similar even if both are positive or both negative.
- **LogisticRegression** [5]: I tested it with various choices of the following parameters: *penalty*, *solver* and *class_weight* or *l1_ratio*. The f1 score is approx 0.955.
- **SVC** [9]: This implementation of SVM for classification is impractical for large dataset (even if the f1 score is 0.965 with linear kernel), so I tested LinearSVC and SGDClassifier.
- **SGDClassifier** [8]: This estimator implements regularized linear models with stochastic gradient descent (SGD) learning. I tested it with various choices of the following parameters: *penalty*, *loss*, *alpha* and *learning_rate*. The f1 score is approx 0.971.
- **LinearSVC** [4]: I tested it with various choices of the following parameters: *penalty*, *loss*, *class_weight* and *C*. The f1 score is approx 0.971.

4 Tuning and validation (max. 400 words)

The algorithms that satisfy the characteristics of this dataset and that have obtained the highest f1 score are *SVC* with linear kernel, *LinearSVC* and *SGDClassifier*.

- I choose **LinearSVC** as one of the two classification algorithm of this dataset since it is similar to SVC with parameter kernel='linear', but the first one has more flexibility in the choice of penalties and loss functions and should scale better to large numbers of samples.
  In the previous steps I noticed that the L2 norm used in the penalization is better than L1.
  Since $n\_samples < n\_features$ then I put the parameter *dual=True*.
  Since the classes are not balanced, I inserted the *class_weight='balanced'* parameter so that the values of y are automatically used to adjust the weights inversely proportional to the frequencies of the classes in the input data.
  The best loss function is *squared_hinge* which is the square of the standard SVM loss.

- I also choose **SGDClassifier** as the second classification algorithm.
  The L1 penalty leads to sparse solutions, driving most coefficients to zero. The Elastic Net solves some deficiencies of the L1 penalty in the presence of highly correlated attributes. The parameter *l1_ratio* controls the convex combination of L1 and L2 penalty. Doing many simulations I found that the optimal value of *l1_ratio* is between 0.1 and 0.2.
  The best loss function is *hinge* which is the standard SVM loss.
  Doing several simulations I saw that the best value of the constant that multiplies the regularization term *alpha* is between 10^{-6} and 10^{-4}, in fact the best value is approximately 10^{-5}.
  The best learning rate schedule is *optimal* and therefore it is not necessary to establish an *eta0* value.

Using one of these two algorithms allows me to obtain an f1 score equal to 0.975 and therefore they are the best obtainable among the group of algorithms studied.

References

[1] *DecisionTreeClassifier* by scikit learn. URL https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html.

[2] *GridSearchCV* by scikit learn. URL https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html.

[3] *KNeighborsClassifier* by scikit learn. URL https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html.

[4] *LinearSVC* by scikit learn. URL https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html.

[5] *LogisticRegression* by scikit learn. URL https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html.

[6] *MLPClassifier* by scikit learn. URL https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html.

[7] *RandomForestClassifier* by scikit learn. URL https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html.

[8] *SGDClassifier* by scikit learn. URL https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html.

[9] *SVC* by scikit learn. URL https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html.

[10] *TfidfVectorizer* by scikit learn. URL https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html.

[11] EliFinkelshteyn. *alphabet detector*. URL https://github.com/EliFinkelshteyn/alphabet-detector.

[12] Amal Nair. *A Beginner's Guide to Scikit-Learn's MLPClassifier*. URL https://analyticsindiamag.com/a-beginners-guide-to-scikit-learns-mlpclassifier/.

[13] PyPI. *emoji*. URL https://pypi.org/project/emoji/.

[14] Michael Saruggia. *Data Visualization: Guida Completa con Python*. URL https://michaelsaruggia.com/data-visualization-plotly/.