Data Science Lab: Process and methods

Politecnico di Torino

## Project report

## Student ID: s276922

### Exam session: Winter 2020

# **Data exploration (max. 400 words)**

The first step of my pipeline was importing the two datasets as Pandas dataframes, in order to easily explore the data.

The development and evaluation datasets are composed of two textual columns: “text” and “class”.

In each dataset each review (“text” column) is different so there are no duplicates to remove.

Moreover, the train set consists of 28.000+ rows while the test set size is about 12.000+ rows.

The “class” column, present in the training set, represents the sentiment associated to the review, it may either be “pos” or “neg”.

Checks on both datasets:

* Missing values: none of the datasets presents them,
* Dataframe info: the dataset is made up by two columns of type string: “text” and “class”,

Using the df.head() method i can explore the first few rows of the datasets and understand the data.

A useful insight, for the next step, is that all the reviews are written in Italian and there is punctuation in the sentences as well as emojis, misspelled words, numbers and symbols.

*Exploration of the training set:*

print(df\_train.isnull().sum())

print(df\_train.info())

print(df\_train.head())

print(df\_train.describe())

Checks on the training dataset:

* Balanced dataset: via counting the number of reviews for each sentiment we can see (**Fig.1**) that the training set is not balanced. The number of positive reviews is double of the number of negative reviews, this will result in a bias of our classifier behaviour,
* Statistics on the reviews: i’ve grouped by class the reviews and computed the main statistics for the “text” column of the training set. The result (**Fig.2**) is that the negative reviews are generally longer than the positive ones, however I will not make use of this information since it may be greatly affected by the not balanced classes

*Statistics for the positive reviews:*

aus = df\_train.loc[df\_train['class'] == 'pos']

# characterizing distribution

**print(aus.text.str.len().mean())**

print(aus.text.str.len().min())

print(aus.text.str.len().max())

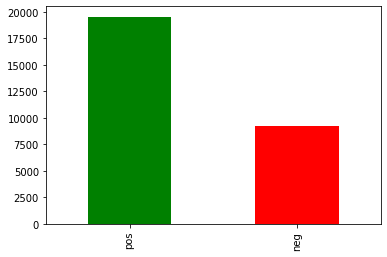
print(aus.text.str.len().std())

**aus.text.str.len().hist()**

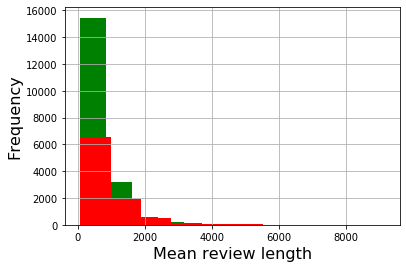
Finally, I’ve displayed wordclouds for the positive (**Fig. 3**) and negative (**Fig. 4)** reviews in the training set in order to have, at a glance, the most common and important words in the review corpus after the preprocessing step.

It’s interesting to point out that the wordcloud is quite well defined for the positive sentiment while it’s noisier in the case of negative reviews. This could be owing to the fact that the dataset is unbalanced, and it will be partially overcome by taking into consideration both unigrams and bigrams, so that adjectives and nouns may be exploited together. (Example: “ottimo”, “hotel” carries more meaning if i consider the bigram “ottimo hotel” in a classifier task).

By either looking at the wordclouds or printing the most common words i can conclude that a quite large number of words appear for both positive and negative sentiment. This will be addressed using stopwords, TFIDF representation and its parameters and via removing some noise from the data.



**Fig. 1**: Number of positive and negative reviews in the training set.



**Fig. 2**: Mean length distribution for positive (green) and negative (red) reviews in the training set.

Mean review length:

* Positive reviews: 624.567 chars
* Negative reviews: 864.228 chars

****

**Fig. 3**: Wordcloud for positive reviews

****

**Fig. 4**: Wordcloud for negative reviews

# **Preprocessing (max. 400 words)**

In this step i’ve exploited multiple techniques in order to achieve a representation of the data that will be adequate as the classifier input.

Firstly, i’ve created a dictionary of unique words present in the training set reviews.

The dictionary is created having the words retrieved from the reviews as keys and their lemma as values. The reason i’ve adopted such structure is the fast reading of dictionaries that will be useful in the cleaning step.

The lemmatization has been addressed by using [spaCy](https://spacy.io/) library that has allowed me to use interesting text processing with the **Italian dictionary**.

*Incomplete lines of code used in the vocabulary creation:*

set\_of\_words = set(all\_reviews.split(" "))

list\_myvocabulary = [(x.text,x.pos\_,**x.lemma\_)** for x in **nlp**(str\_set\_of\_words)]

my\_vocabulary = pd.DataFrame(list\_myvocabulary, columns = ['base', 'pos', 'lemma'])

**my\_dict\_lemmas** = {row[0]: row[2] for row in my\_vocabulary.values}

The second step of the preprocessing pipeline was the actual cleaning of the data.

In order to achieve a representation of the reviews understandable by the classifier i’ve exploited the TFIDF representation.

The TFIDF allows to represent the documents (reviews) as weighted feature vectors.

The weight assigns more importance to terms that are frequent in a certain document (TF) yet infrequent in the entire set of documents (IDF).

vectorizer = TfidfVectorizer(tokenizer = tokenizer, stop\_words=stopwords, encoding='utf-8',lowercase=True, min\_df = 1, max\_df = 0.7, ngram\_range = (1,2), strip\_accents = 'unicode', sublinear\_tf = True, use\_idf = True, analyzer='word')

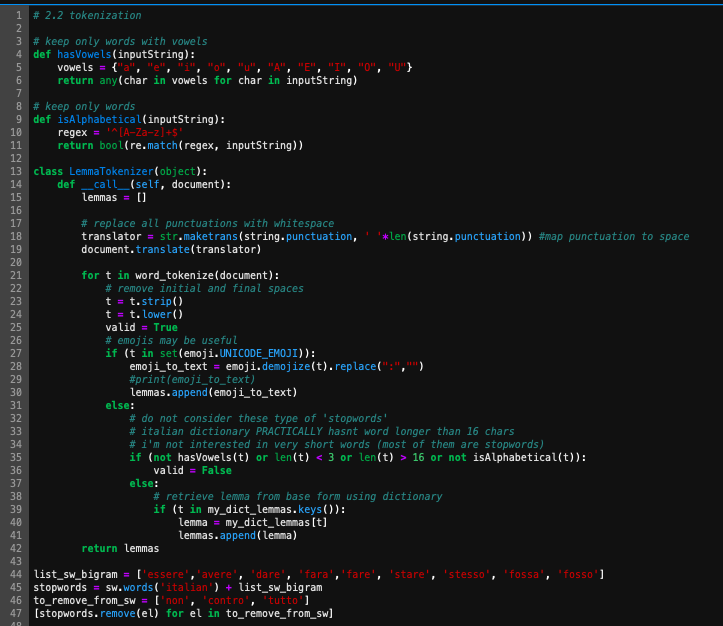
The most important parameters of the vectorizer in our case study are:

* Min\_df, set to 1 in order to not exclude any word
* Max\_df, set to 0.7 in order to exclude really frequent words not identified as stopwords
* Sublinear\_tf, set to True in order to normalize the weight of frequent words in a document
* Tokenizer

In addition, into the custom tokenizer class i’ve adopted different document processing techniques in order to maximize the information carried by the TFIDF matrix and minimize its size:

* Tokenisation: split the document into words
* Case normalization: each word has been converted to lowercase
* Checks on the word, each word must: be alphabetical, contain at least a vowel and have length [between 3 and 16 chars.](http://www.ravi.io/language-word-lengths)
* Stopwords removal: useful in order to remove the frequent and not so meaningful words (prepositions, articles…). I deleted “non”, “contro” and “tutto” from the stopwords since they may be useful in classifying polarized reviews
* Lemmatization (preferred to the Stemmatization owing its higher accuracy), via [spaCy](https://spacy.io/)
* Emoji normalization: since emojis may be important in an hotel review (especially the most common ones such as thumbs or happy/sad faces) i’ve used the [Python emoji library](https://pypi.org/project/emoji/) in order to detect and convert them into text.

Preprocessing output: TFIDF sparse numerical matrices of train and test sets (dimensionality reduction has not been applied intentionally in the final algorithm and will be discussed in the next sections).



**Fig. 5**: complete tokenisation function

# **Algorithm choice (max. 400 words)**

In order to perform the classification task, I’ve have tried and tested multiple approaches in order to maximize the F1-score and the accuracies of the predictions, such techniques are the following.

* **MLPClassifier**, the MLPClassifier is based on neural nerworks in order to perform the classification task.

This particular classifier did not work well in my case mainly owing to the very large dataset I used to train it and the fact that the problem was not balanced.

Without using dimensionality reduction, the classification was simply not feasible.

The classifier would have needed re-sampling of the dataset in order to overcome the unbalanced limit.

Moreover,the parameters tuning and the usage of the MLPClassifier were somehow more complex with respect to the other classifiers and finally led me towards other choices.

* **Random forest classifier**, an ensemble of decision tree classifiers that are quite robust to noise and outlier.

In this particular case study though the random forest classifier showed its limitations such as: difficulty with managing an unbalanced dataset and slowness when dealing with a very large sparse data.

The result of the classifier was in fact not so accurate.

Moreover, in order to make the classification feasible a dimensionality reduction was needed (**Fig. 6**), that resulted in a loss of accuracy of the overall model in comparison with the LinearSVC classifier (**Fig. 7**).

* **SVM classifier**, especially its LinearSVC implementation. The SVM algorithms try to represent the points of the train dataset in a space so that points that belongs to different classes are divided by a gap as wide as possible. The point in the test dataset are then added to the same space and predicted to be in the category in which they fall with respect to the class division.

In this case study i’ve used only linear SVM in particular the LinearSVC classifier.

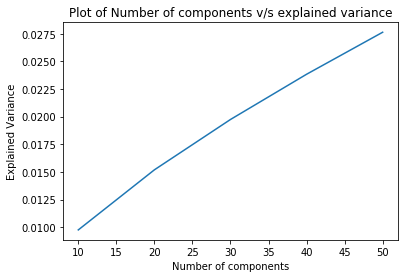
SVMs work well with sparse matrices so i have not performed any kind of dimensionality reduction on the TFIDF matrices, in order to give the classifier the maximum information possible of the documents and the maximal explained variance.

Another pro of this approach was the very fast training of the model and its flexibility toward unbalanced datasets (via the parameter class\_weight).

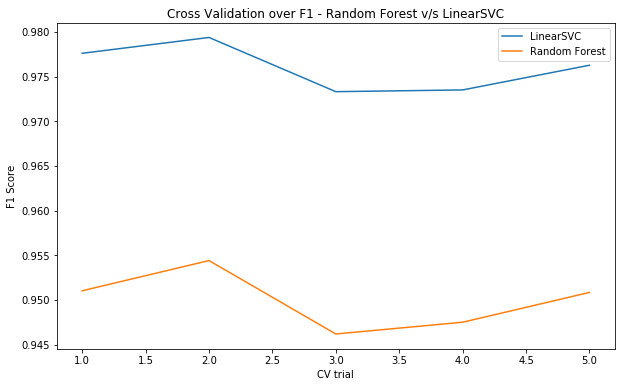
The linear SVC gave me the best result in terms of accuracy and f1-score in comparison to the other classifiers (**Fig. 7**).

To sum up I have deeply tested both the LinearSVC and the Random Forest Classifier configurations and I’ve chosen the first one in terms of scoring, explained variance and velocity.

Further analysis may comprehend some other classifiers such as the SGDClassifier, neural networks or combinations of classifiers, however in my opinion improvements should be focused primarily in the preprocessing step (stopwords, parameter tuning in the vectorizer and so on), that is the reason why I considered just a handful of classifiers.

****

**Fig. 6**: Explained variace of the data when applying SVD before Random Forest

****

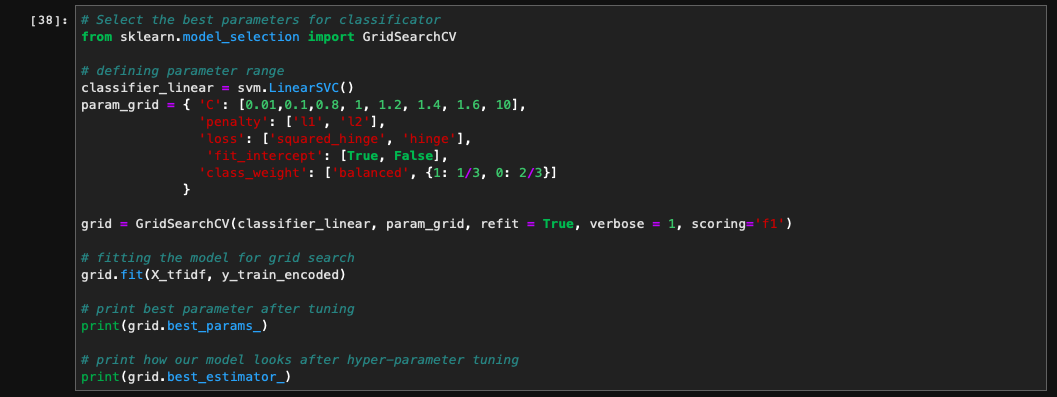
**Fig. 7**: Cross validation of data over f1-score: LinearSVC v/s Random Forest (SVD: 50 components)

# **Tuning and validation (max. 400 words)**

**Tuning**

In order to choose the best parameters for the LinearSVC classifier (the best one in terms of results) I’ve used the following snippet of code (**Fig. 8**) that allows me to maximize the selected scoring function (f1) and make the program more robust and stable when a different test set is considered (the parameters are in fact dynamically chosen).

The sklearn class that allows me to do that is the GridSearchSVC that automatically performs all possible parameters combinations and returns via the best\_params\_ attribute the optimal configuration to maximize the stated score.



**Fig. 8:** Hyperparameters dynamic choice for the classifier

The most important parameters, for the LinearSVC classifier, are:

* C, the penalty parameter of the error term
* Class\_weight, I assigned class weights that are inversely proportional to the frequencies of the classes in the dataset, the balanced clause computes automatically class weights based on the y values
* The kernel is by default linear in the LinearSVC classifier

**Validation**

In order to validate the chosen model and have a real estimate for the f1-score reached by the above-mentioned classifier i’ve exploited the following two techniques: holdout and cross validation.

In both cases the goal was to avoid overfitting and have a real estimate for the f1 score by dividing the dataset into train and test set.

In the first case (holdout) I simply subdivide the dataset in two sub-datasets following the 80/20 proportion, this technique allows a first validation of the classification model.

The major part of the information is used as training set to train the model, the remaining rows are used to test the predictions made by the model via comparing its predictions to the ground truth.

The result when using holdout was however very close to one, that means that the given estimate is not that reliable and I’m probably overfitting the train set.

The second approach exploits the cross validation and it’s more reliable as an estimate score for the model.

The dataset is divided in n=CV partitions. At each iteration a different partition is used as test set while the others combined are used to train the model, at each loop a scoring metric is computed.

This approach returns a Numpy array of length n of scoring metrics, one for each iteration considered.

The result was, in this, case quite close to the one assigned from the evaluation platform.

I report here the two snippets of code used to validate the model and their result:

Holdout:

X\_train,X\_test,Y\_train,Y\_test = train\_test\_split(X\_tfidf,y\_train,test\_size=0.2,random\_state=1000)

solution = classifier\_linear.predict(X\_test)

f1\_score(Y\_test, solution, average='weighted')

# result: 0.9993045714005748

Cross-validation:

f1\_linearsvc = cross\_val\_score(classifier\_linear, X\_tfidf, y\_train\_encoded, cv=5, scoring='f1')

print(f1\_linearsvc)

# result: [0.97760717 0.97937882 0.9733112 0.9735057 0.97626946]

**References: used libraries**

* Matplotlib
* Sklearn
* Spacy, <https://spacy.io>

I have used the spacy library in the data preprocessing step of my pipeline in order to exploit some text-processing techniques.

These were in particular: lemmatization and POS that are available for a number of languages, Italian included. Prior to Spacy I used the Porter and Snowball stemmer for the Italian language, however I’ve found the lemmatization as more accurate.

* Pandas
* Numpy
* Emoji, <https://pypi.org/project/emoji/>

As stated in the report I thought that the emojis, if properly handled, could be useful in determine the sentiment expressed by a review.

In order to exploit this kind of information I’ve used the emoji library that has allowed me to detect and transform into text the emojis. In particular the latter has been addressed using the “demojize” function.

As a rule of thumb I think the most common emojis, such as smiles are the most useful in this kink of text classification.

* NLTK
* Wordcloud