Machine Learning Project 2020/21

Detecting inconsistent Amazon reviews

Fabrizio Rossi 1815023
Emanuele Vincitorio 1811290
Matteo Orsini 1795119

Sapienza
Università di Roma

Contents

Chapter 1

Introduction

Nowadays, **online shopping** has become more and more popular, so **reviews** of other people are a powerful way to make sure the product that users are going to buy will reflect what they expect. Reviews are also a way to assess the **quality** of an item when you can't physically inspect it, which is the usual scenario in an online e-commerce like Amazon. For this reason, the rating of a product can directly influence the choice of purchase of a customer, and it's important to keep it as close as possible to the real sentiment expressed by other buyers, in order to keep high the trustability of your e-commerce.

In our work, our goal is to build a system that can detect when a review expresses a certain sentiment in the title or the body, but the rating is **inconsistent**. In the following picture we can see an example of what in this document will be referred to as "inconsistent review":

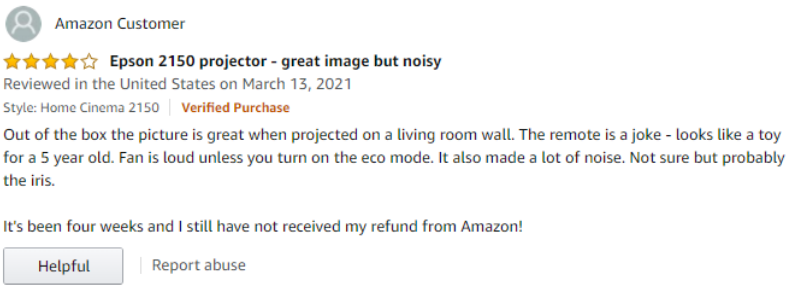


Figure 1.1: A discordant review of a projector

In this case, we can see that the Anonymous reviewer thinks that the projector is not that great, since it has lots of problems, but he still rated it 4 stars, even if he asked for a refund.

This system could be very useful for an e-commerce company, because as we said reviews are really important for customers, and having good quality reviews helps both the customer to make more informed purchase decisions, but also helps the company to improve their quality of services. For this reason, our idea is to implement a system that when a new review is being posted, it will check if the rating provided by the customer it's in line with the text and metadata provided by the review. To do so, we are going to build a **Machine Learning model** which is going to predict if the feedback that a user wants to express is negative, neutral or positive.

1.1 Dataset overview

As our dataset we used a set of reviews of **electronics** products from Amazon.com, available at the following link. The dataset contains about **6.7 million** records, but we used only a portion of the data (1 million) in order to process them in a faster way and keep all of them in main memory.

Even if reviews for other product categories of Amazon are available on the same website, we picked only data about electronics products, because having limited computational resources we wouldn't be able to process more data, and so we chose one of the most popular category, which is also our favorite one.

The features of the dataset are shown in Table 1.1.

Name	Type	Description
reviewerID	string	ID of the reviewer
asin	string	ID of the product
reviewerName	string	Name of the reviewer
vote	number	Number of users that found the review helpful
style	dictionary	A dictionary of the product metadata
reviewText	string	Text of the review
overall	number	Rating of the product from 1 to 5
summary	string	Title of the review
unixReviewTime	timestamp	Time of the review (unix time in seconds)
reviewTime	string	Time of the review (raw)
image	list	URLs of images posted by the user after receiving the product
verified	boolean {True,False}	If the purchase is verified by Amazon

Table 1.1: Features available in the dataset

1.2 Data Preprocessing

We started the data preprocessing phase inspecting the dataset and checking how many values are empty or missing for each feature. Doing so, we found out that many values of the features *vote*, *style* and *image* are **missing**.

For the feature *vote*, the reason is that when a review hasn't yet been marked as helpful from any other user, instead of having value 0, it doesn't have any value. So, we filled those values with the correct value, i.e. zero.

Instead, for the other features, we filled the missing values of the *style* property with an empty dictionary and the values of the *image* feature with an empty list.

Moreover, we dropped the records with one of the remaining features missing, since the data available is more than enough also without those records.

Then, since the *vote* feature follows the English format for numbers, using commas to separate groups of thousands, we removed them in order to treat them as integers.

Finally, in order to predict if a review indicates a positive, neutral or negative feedback we mapped the *overall* feature in the following way:

$$new\_overall = \begin{cases} 0 & \text{if } overall \in \{1, 2\} \\ 1 & \text{if } overall \in \{3\} \\ 2 & \text{if } overall \in \{4, 5\} \end{cases} \tag{1.1}$$

Chapter 2

Exploratory Data Analysis

In this section we proceed analyzing the data in order to produce some useful **insights** on the dataset and answer some related interesting **questions**.

2.1 What are the most used words in review titles?

In the first place, we wanted to see which are the **most used words** for positive, neutral and negative reviews. To show the occurrences we used bi-words in order to keep the association of a word and the negation, e.g. "not good" instead of only "good". Instead, for the graphical representation we used a Word Cloud which shows words using a bigger font size the more they occur in the review title.

Figure 2.1: From left to right, word clouds of positive, neutral and negative reviews

In Figure 2.1, we can see the results where bi-words like "works great" or "great product" are the most used in positive reviews, instead, bi-words like "poor quality" or "waste money" are the most used for negative reviews. Finally, we can also see that neutral reviews present coherent bi-words like "it works" or "not bad".

2.2 How long should be a review to be helpful?

When you read a review on Amazon website, you have the opportunity to **vote** it as **helpful** in order to increase its visibility to other users. For this reason, we wanted to check if there is a correlation between the length of a review and the number of people that marked it as helpful. So, we plotted the average review length for all the reviews with a number of votes less than 200.

From Figure 2.2 we can see that the average review length increases in a logarithmic way, and for reviews with a high number of votes, i.e. more than 100, the average review

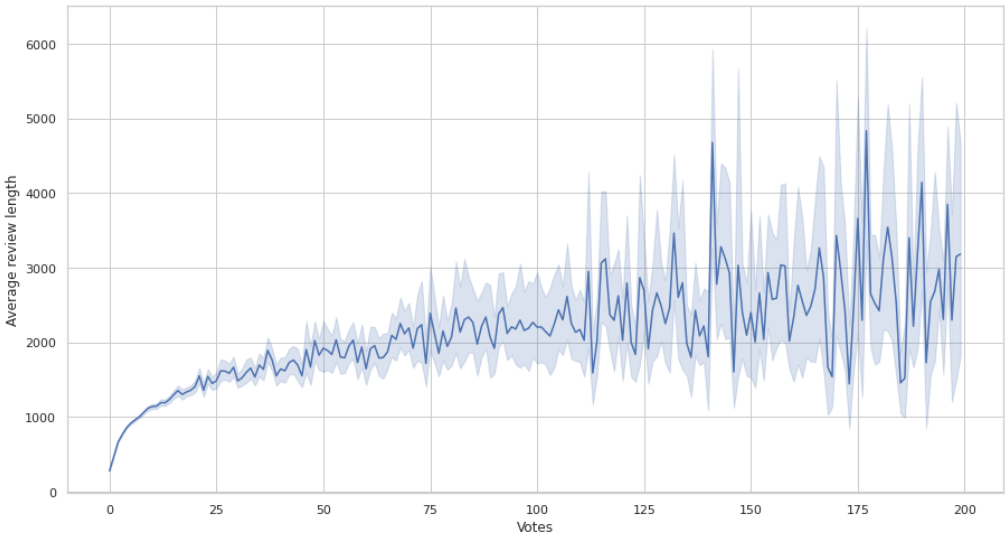


Figure 2.2: Plot of the average length of reviews with less than 200 votes

length has a very high variance. This tells us, that helpful reviews tends to be longer than other reviews, but after a certain point, we can get mixed results.

2.3 Are images helpful for reviews?

Another feature offered by Amazon is the possibility to add **images** to reviews in order to show the purchased product. So, we asked ourselves if posting pictures helps to increase the helpfulness of a review. To check this, we plotted the average number of votes for reviews with and without images, as shown in Figure 2.3. From the plot we can see that the average number of votes for reviews containing images is much higher than the one for image-less reviews.

Figure 2.3: Plot of the average number of votes for review containing and not containing images

This means that reviews which contain at least an image are in general more helpful

than those without any images.

2.4 How many reviews are posted per month?

Checking the number of reviews in each **month** of the year can give us an insight on which are the most profitable months for an e-commerce company, since usually products reviews are posted in the same month as the purchase. For this reason, we were interested in analyzing the average number of reviews for each month, and given the dataset, we averaged all the data from 1999 to 2008. From the resulting plot, shown in Figure 2.4, we can see that we have a non linear trend: we have a peak of reviews in December/January probably because of the Christmas Holidays where a lot of consumer buy gifts, while the number of reviews decreases in the summer, maybe because people go on vacation.

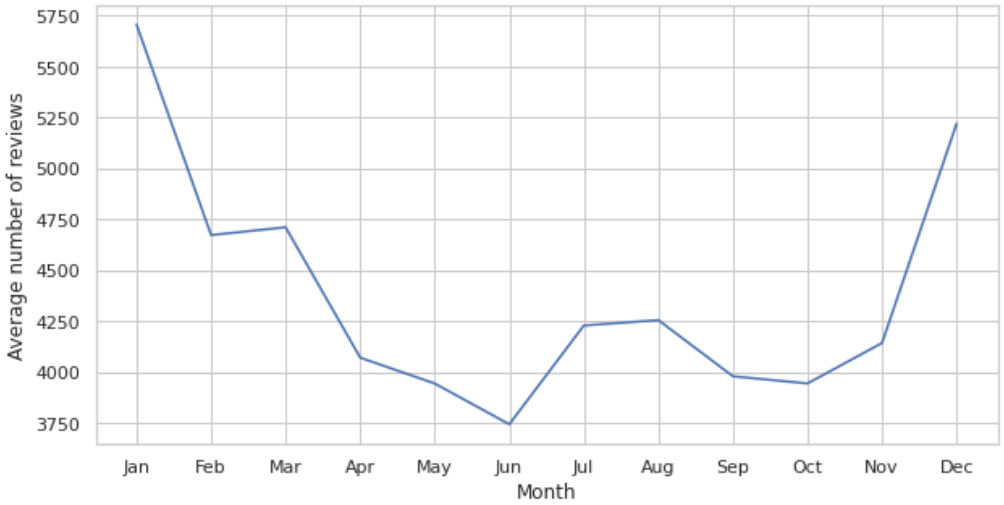


Figure 2.4: Average number of reviews per month from 1999 to 2008

2.5 Are verified purchase-reviews more helpful?

On Amazon there is also the possibility to post a review without having purchased the product directly on the site, but this is clearly marked, showing a **"verified purchase"** label on reviews posted by users that purchased the product through Amazon, like shown in Figure 2.5.

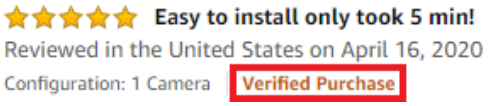


Figure 2.5: Verified purchase label marked by a red rectangle

This is a double-edged sword since when we look at the reviews of a product, we can both see an increased number of them since people that bought the product from another shop can still leave a review, but this means that also people that didn't really purchase the product can do so, in order to both increase or decrease the product rating.

For this reason, we checked if verified purchase-reviews are more helpful than the others, using the plot shown in Figure 2.6.

Figure 2.6: Plot of the average number of votes for verified or not verified purchases

From the plot we can see, in contrast to our expectations, that verified reviews are in general less helpful than non-verified reviews. In fact, reviews for verified purchases have an average number of votes of about 1.6, which is lower than non-verified purchases, which is about 8.4.

Chapter 3

Feature Engineering

In this chapter, we describe the process of feature engineering carried out in order to produce the plots shown in the EDA, and to prepare the data to be processes by our Machine Learning models.

3.1 Feature manipulation

In particular, these are the steps carried out to produce the plots:

- We created a new feature called *reviewLength* computed based on the length of *reviewText*;
- We created a new binary feature *hasImages* that indicates if the review contains one or more images;
- We transformed the feature *unixReviewTime* to the more tractable *day, month* and *year* features.

Moreover, we performed some additional operations to the remaining features:

- *reviewerName* has been mapped to a new feature called *isUserAnonymous* since some reviews have a generic name like *Amazon Consumer* or *Kindle Consumer* for users that posted anonymously;
- *style* has been transformed to *hasStyle* which is a binary feature indicating if the product comes in different versions;
- *verified* has been mapped from booleans to a new feature called *isVerified* composed by binary values, i.e. 0 or 1.

3.2 Correlation Matrix

In a typical Machine Learning pipeline is also important to check if there are some **highly correlated features**, in order to avoid redundancy and improve training times. For this reason, we computed the Pearson's Correlation Matrix, which is shown in Figure 3.1.

As we can see from the matrix, without doing further adjustments, we don't have some particularly correlated features: we have a slightly positive correlation between the number of *helpful votes* and the *review length*, which has been already showed in the EDA and some other correlation between *year* and the *review length* or *verified purchases*, but we don't think it's something relevant.

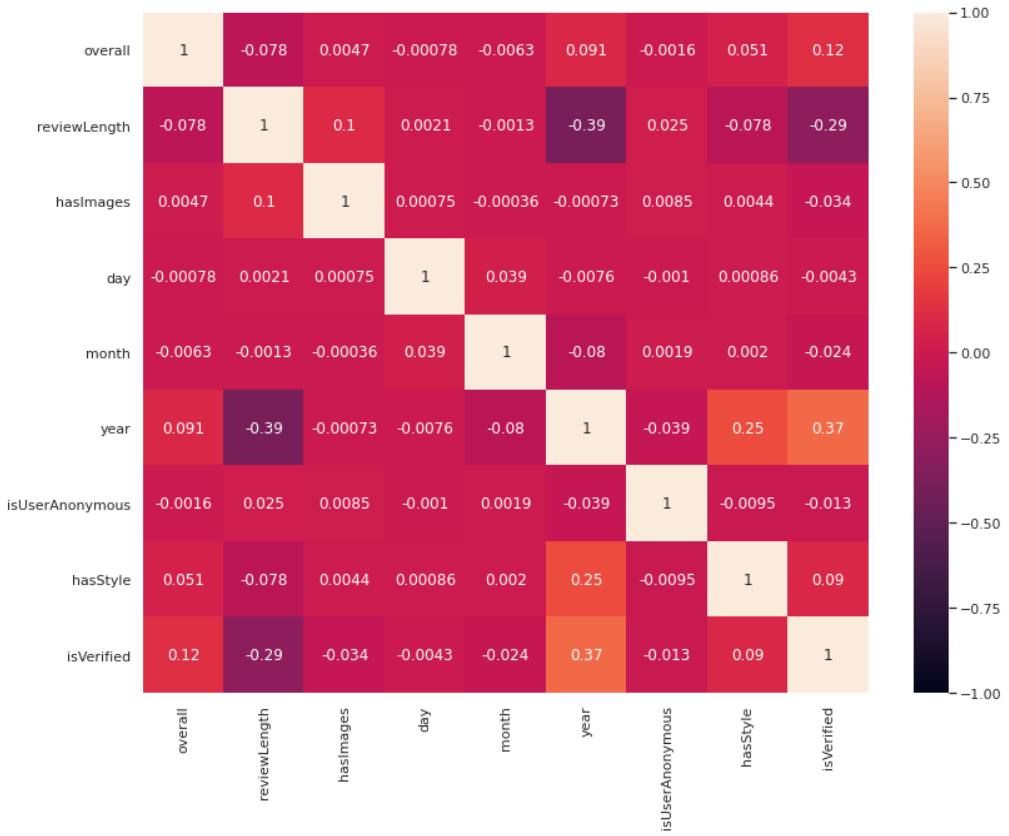


Figure 3.1: Correlation matrix of the features

3.3 Drop unused features

In order to start using Machine Learning models on our dataset, we also **removed** some features which we are not going to use. Among these ones, we have some features that we deemed not useful:

- *asin* which is the id of the product;
- *style* which are some tags describing the product version;
- *reviewerID* which is the id of the reviewer;
- *reviewTime* and *unixReviewTime* because they have been replaced by the *day, month* and *year* features;
- *image* which is a list of URLs pointing to the review images, and was replaced by the *hasImages* feature.

Finally, we also removed the *vote* feature, which indicates how many helpful votes received a review, not because it is not useful, but because since our system needs to predict the sentiment of a review as soon as it is posted, at this stage we cannot obtain this information, and so we cannot use it to predict if the review is negative, neutral or positive.

Chapter 4

Methods

In this project we tested different **approaches** starting from the simpler ones, relying only on the metadata of the reviews without using plain words as features (Section 4.1), passing through approaches based on the occurrences of terms and TfIdf (Section 4.2), and finally arriving to approaches based on more complex techniques like embeddings (Section 4.3). In the following sections we are going to present them, showing which models have been used and their architectures.

4.1 Text-free approach

This is the baseline approach, where we **didn't consider** at all the words contained either in the **title** or in the **content** of the reviews, but we only leveraged the **length** of the content and all the other features offered by the dataset mentioned before.

Since in this approach we used only 8 features, we tried numerous Machine Learning models offered by the *scikit-learn* library in order to check which one would perform the best. In particular we experimented with the following models:

- **Dummy Classifier**, which is just a random guesser;
- **Logistic Regression**, using a multinomial loss;
- **AdaBoost**, which is one example of a boosting ensemble;
- **Linear SVM**, because a normal SVM would have taken too much time to train;
- **Random Forest**, which is one example of a bagging ensemble;
- **MLP**, which is a neural network composed by multiple neurons.

4.2 Bag-of-Words approach

In this approach, we started to consider **text features**, namely the content and the title of the review. In order to do so, we first tried to only consider those features, and then we also added all the remaining ones in order to increase the accuracy of the models.

The first step to process text is **tokenization**, where a collection of sentences is split into single words, called *tokens*. In this phase, usually punctuation and some very common words (called *stop words*) are removed, but we chose to consider them, to preserve negations in negative results.

The second step is to represent the tokens contained in a text in a **vector space**. Encoding text in a suitable way to run Machine Learning models has always been a challenge in the field, and numerous representation have been proposed.

The first method we used is the **Bag-of-Words** model, which doesn't take in consideration the order of words, their multiplicity or their semantic relationships. For this reason, a text is represented as a multiset of its words, and can be turned into a vector x by first estimating the number of distinct words contained in the whole collection, building the so called **vocabulary** V, and then creating the vector in the following way:

$$x_i = \begin{cases} 1 & \text{if the } i\text{-th word of the vocabulary } V \text{ is contained in the document} \\ 0 & \text{otherwise} \end{cases}$$

We then considered another vector representation, the **TfIdf weighting scheme**, which is similar to the Bag-of-Words model, but it takes in consideration also how often a word appears in a text or in the whole collection. For this reason, we used the notions of term frequency $\text{tf}(t, d)$ and inverse document frequency $\text{idf}(t)$.

The first one, takes in consideration the number of times that a term t appears in a document d, or in our case in a review, and it is computed as follows:

$$\text{tf}(t, d) = \frac{\text{raw count of } t \text{ occurrences in } d}{\text{total number of tokens in } d}$$

The second one, instead, takes in consideration the rarity of a word in the collection, and it is computed as follows:

$$\text{idf}(t) = \log\left(\frac{1 + \text{total number of documents in the collection}}{1 + \text{number of documents in which } t \text{ appears}}\right) + 1$$

Finally, to represent a document d as a vector x we used the following formula:

$$x_i = \text{tf}(i, d) \times \text{idf}(i)$$

And then we normalized the vector using its L2-norm:

$$x_{norm} = \frac{x}{||x||_2} = \frac{x}{\sqrt{x_1^2 + x_2^2 + ... + x_n^2}}$$

In order to use both representations we used the *scikit-learn* library, which provides us the *CounterVectorizer* class, and using the option *binary=True* computes the Bag-of-Words representation of a text. To transform review into TfIdf vectors, instead we used the *TfidfVectorizer* class, which outputs vectors using the TfIdf weighting scheme.

Finally, we tested a Random Forest Classifier and a simple Multi-Layer Perceptron on the dataset both containing only the review text, and also adding back the previous metadata features.

4.3 Embeddings approach

The last approach we tested is based on the concept of **embeddings**. A major drawback of working with the Bag-of-Words representation of a text is the **sparse matrix** that it produces, since in a sentence we will not have all the terms contained in the vocabulary, but just a few of them, resulting in lots of zeros.

Word embeddings give us an efficient way to handle text data without working with a sparse matrix: we represent each word in a d-dimensional vector space, where d is an hyper-parameter of the model, obtaining dense vectors with floating-point values.

The model based on embeddings learns the best **latent space representation** for the entire corpus made available as training set. Moreover, this space will capture some properties of the language used: words that have similar meaning will be closer in the vector space.

Since training embeddings on our task would be highly expensive in terms of computation and time needed, we used the embeddings already produced by the GloVe model, proposed by the Stanford University.

In our model we used the **GloVe** version available at the following link, which offers embeddings of length 100 based on a vocabulary containing 400.000 terms. Using the GloVe embeddings we created an embedding matrix in order to feed it to the *Embedding layer* offered by Keras. The embedding matrix is a matrix of size $|V| \times d$ where V is the vocabulary and d is the chosen embedding dimension, in our case 100. The i-th row of the matrix will store the association between the i-th term in the vocabulary and its relative embedding.

In order to build a **vocabulary** from our corpus we used the *Tokenizer* class of Keras which splits the terms, transforms them in lowercase and assigns to each of them an identifier. At the end, it builds the entire vocabulary V and returns the integer-representation of each review using terms identifiers instead of words. Since each review vector has a different length, we needed to pad them, so we used the *pad_sequences* utility offered by Keras to construct sequences all of the same length.

Regarding the Machine Learning models, we tested two architectures: a **Convolutional Neural Network** described in Section 4.3.1 and a **Long Short-Term Memory** model described in Section 4.3.2.

Both models are composed by an initial *Embeddings* layer which transforms the produced padded sequences into embeddings vectors. This layer uses the GloVe weights as starting point, and then improves the vector representations for our task during the learning phase, performing a form of transfer learning.

Moreover, they both use the same loss function, the **sparse categorical cross entropy**, which is a version of the categorical cross entropy designed for integer encoded classes. Given a set of classes C, it is computed in the following way:

$$L(y, \hat{y}) = -\sum_{c=1}^{|C|} y_c \log(\hat{y}_c)$$

We used this version of the loss in order to reduce the memory utilization, since it enables us to store only the integer index of the class, instead of a one-hot vector of size $|C| = 3$ for each sample.

The hyper-parameters for both architectures have been selected after manually tuning them, checking the performances on the validation set.

4.3.1 Convolutional Neural Network (CNN)

CNNs are usually used for image processing but it is proven that they can be useful also for **text processing** since they are able to capture the context of words using the **convolution** operation. For this reason, we used a CNN model, whose architecture is shown in Figure 4.1, composed by an initial embedding layer, which converts the words from integers to 100-dimensional vectors of a latent space.

The vectors are then modified by a *SpatialDropout1D* layer, which actually drops an entire dimension using zeros given a certain probability, and is typically used to avoid overfitting, since it modifies the input. Then we used some *Conv1D* layers which are used to learn some patterns in the sequence, followed by *MaxPooling1D* layers used to reduce the data dimensionality.

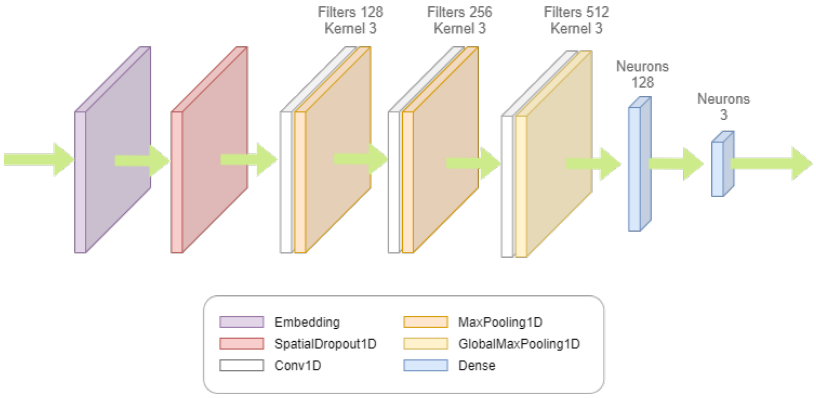


Figure 4.1: CNN-based model

Moreover, there is a fully connected layer with the *ReLu* activation function, which alleviates the vanishing gradient problem and is defined as follows:

$$ReLu(x) = \max(0, x) \tag{4.1}$$

At the end of the architecture, we have another fully connected layer with the *softmax* activation function, which produces a vector $\hat{y}$ that represents a class probability distribution according to the following formula:

$$softmax(\hat{y}_i) = \frac{e^{\hat{y}_i}}{\sum_{j=1}^{n} e^{\hat{y}_j}} \tag{4.2}$$

4.3.2 Long Short-Term Memory (LSTM)

Recurrent Neural Networks (RNNs) are one of the most used models to process text since they are really good for sequences thanks to their **recurrent** nature. One type of RNNs is the LSTM which overcomes the shortcomings of the basic RNN model, solving the problem of the vanishing gradient using a **cell state**. Moreover, they are composed by an **input gate** which is used to add new information to the cell state, a **forget gate**, which can make the cell forget some information, and finally the **output gate** which produces and filters the output of the cell.

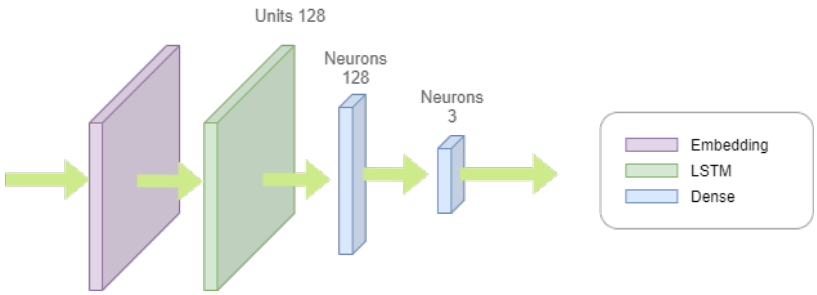


Figure 4.2: LSTM-based model

Our LSTM architecture is shown in Figure 4.2: we still used **dropout**, which is offered by Keras as a hyper-parameter of the LSTM layer, to avoid overfitting, and then we used the same two fully connected layers as the CNN as output, in order to learn some useful data transformations to predict the correct class.

Chapter 5

Testing and Evaluation

In this section are described the preparation steps performed before the testing and the results obtained by the three approaches illustrated in the previous chapter.

5.1 Data preparation

In order to feed the data to the Machine Learning models we built two different structures containing the **train features** (X) and the **target feature** (y) whose components are respectively shown in Table 5.1 and Table 5.2.

Feature name	**Type**
isUserAnonymous	boolean {0,1}
hasStyle	boolean {0,1}
reviewText	string
reviewLength	number
summary	string
day	number [1, 31]
month	number [1, 12]
year	number [1999, 2008]
hasImages	boolean {0,1}
isVerified	boolean {0,1}

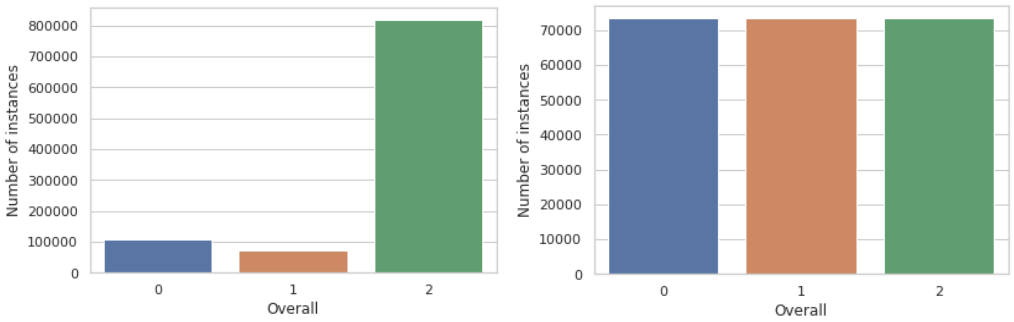
Table 5.1: X structure

Feature name	**Type**
overall	number {0,1,2}

Table 5.2: y structure

Then, we proceeded to inspect the target variable **distribution**, which is shown in Figure 5.1a. From the plot we can see that the data is **highly unbalanced**: the number of instances with overall 2, i.e. positive reviews with 4 or 5 stars, is much higher than the number of instances with overall 0 and 1.

Many Machine Learning models need a balanced dataset to perform unbiased predictions, so we chose to undersample our dataset using the popular library *imblearn*, and not to oversample the other classes since we have a lot of data for our task.



(a) Before

(b) After

Figure 5.1: Distribution of the target variable before and after the undersampling

After the process of balancing, as we can see from Figure 5.1b, the number of instances is reduced to about 75.000 for each class, summing up to about 220.000 records in total.

Finally, we proceeded to split the data into three different sets as shown in Figure 5.2:

- 70% **training set**, used to train the model;
- 15% **validation set**, used to validate models with different hyper-parameters;
- 15% **test set**, used to evaluate the performances of the models in a real case scenario.

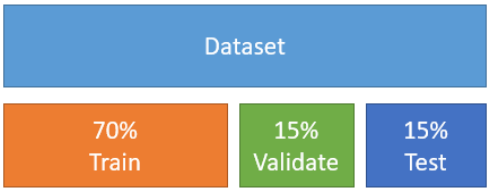


Figure 5.2: Different portions of the dataset

5.2 Text-free approach

In this approach, in order to determine the best hyper-parameters for the different models tested, we created a small wrapper of the standard grid search available in the *scikit-learn* library. Since the **grid-search** uses the cross-validation approach we used the *StratifiedShuffleSplit* class, which shuffles the samples and creates a number of defined splits, that we set to 1 in order to obtain just 2 sets, the train set and the validation set.

Then, using this method we proceeded to test different *scikit-learn* models only on the text-free features. The obtained results are reported in Table 5.3.

As we can see from the results shown before, the **Random Forest Classifier** is the best model for our task with a very high accuracy on the train set, but a not so high accuracy on the validation set. From the theory we know that the Random Forest is very

Model name	Accuracy Train Set	Accuracy Validate Set
Dummy Classifier	0.332	0.333
Logistic Regression	0.402	0.406
AdaBoost	0.423	0.428
Linear SVM	0.401	0.402
Random Forest	**0.915**	**0.450**
MLP	0.427	0.425

Table 5.3: Accuracy computed for each model on the train and validate set

good in fitting the training data, but since in our case we don't have many discriminative features for our task, we can't achieve very high results also on the validation set.

Then, we proceeded to inspect the performance of the best model on the test set. As we can see from Figure 5.3, the Random Forest Classifier classifies correctly the majority of the samples on the train set, instead, as expected, on the test set the performances are a bit worse, committing some errors especially on classes representing actual neutral and positive reviews.

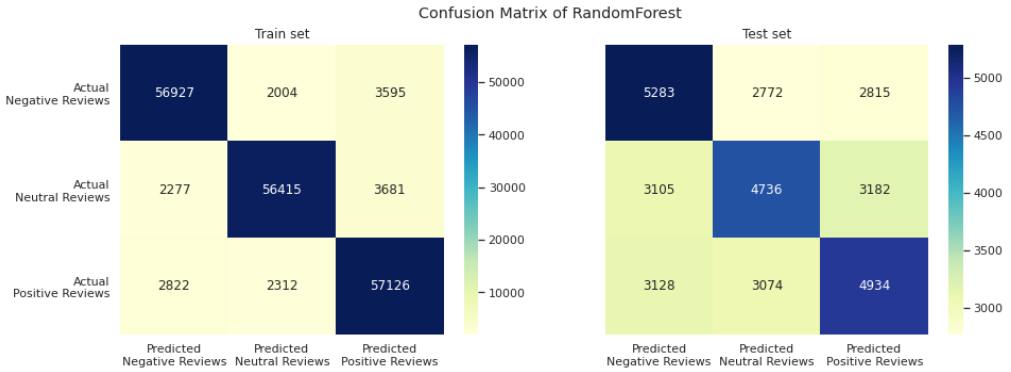


Figure 5.3: Confusion Matrix for the Random Forest Classifier on the complete train set and test set

As we can see from the plot, the most important feature is the review length which can be an important factor to determine the positiveness of a review, and is followed by day, month and year, leaving all the other features to the bottom of the ranking.

Moreover, we produced the **classification report** for the Random Forest Classifier, to have a more in-depth view of its performance on train and test set. In Table 5.4 and Table 5.5 is reported the classification report for the train and test set containing the precision, recall, F1-score and accuracy metrics.

Class	Precision	Recall	F1-score	Support
Class 0	0.92	0.91	0.91	62526
Class 1	0.93	0.90	0.92	62373
Class 2	0.89	0.92	0.90	62260
Accuracy			0.91	187159
Macro avg	0.91	0.91	0.91	187159
Weighted avg	0.91	0.91	0.91	187159

Table 5.4: Classification report of the Random Forest Classifier on the train set

Class	Precision	Recall	F1-score	Support
Class 0	0.46	0.49	0.47	10870
Class 1	0.45	0.43	0.44	11023
Class 2	0.45	0.44	0.45	11136
Accuracy			0.45	33029
Macro avg	0.45	0.45	0.45	33029
Weighted avg	0.45	0.45	0.45	33029

Table 5.5: Classification report of the Random Forest Classifier on the test set

From the results we can see that the predictions are balanced among the different classes. On the train set the Random Forest Classifier reaches an average F1-score and average accuracy of 0.91. Instead, on the test set the performance are worse, having an average F1-score and accuracy of about 0.45.

Since the best performing model is the Random Forest Classifier, we can also gather insights on the **feature importance** computed during the training phase of the model, as reported in Figure 5.4.

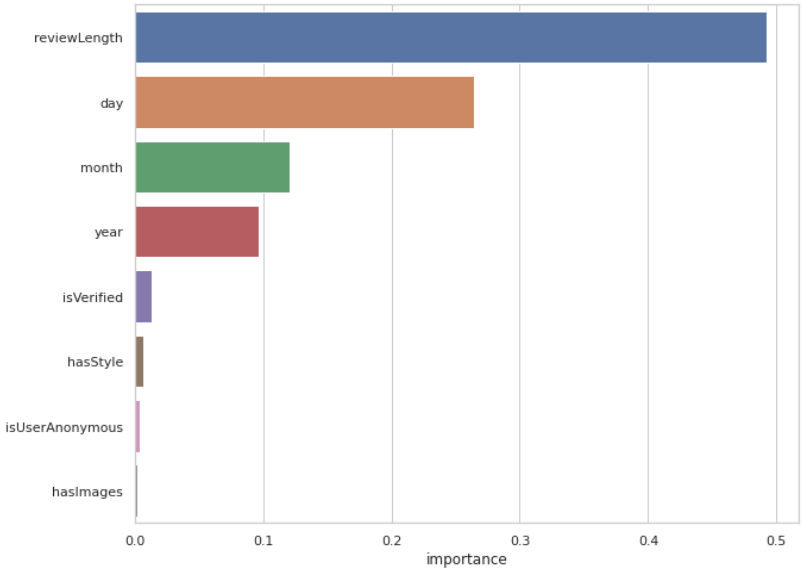


Figure 5.4: Plot of the feature importance gave by the Random Forest Classifier

So, the performance of this approach is not so good, because as expected, it has a major drawback, which is the lack of significant features for the model to express a reasonable prediction on the review.

5.3 Bag-of-Words approach

Then, we proceeded with the second approach based on the vector representation of the reviews using the Bag-of-Words model.

Regarding the **vocabulary** size we took in consideration only the top 500 words, which is a good trade-off between efficiency and performance.

One important factor is that we chose to **not ignore stop words**, since some of them can be really important for the classification: for example, the adverb "not" can

be a really useful information to have in order to discriminate the judgment expressed by a review.

For the models of this approach we **didn't use the grid search** since it would be unfeasible to run multiple times the models with the amount of features we have. So, we defined manually the set of hyper-parameters for the models, as described in Table 5.6.

Model name	Number estimators	Criterion	Max depth	Min samples leaf
Random Forest	50	gini	50	5

Model name	Hidden layer sizes	Activation	Solver	Alpha
MLP	(32)	relu	adam	0.001

Table 5.6: Hyper-parameters used for the Random Forest Classifier and the Multi-Layer Perceptron

For this approach we trained a **Random Forest Classifier** and a **Multi-Layer Perceptron** on different types of representations of the dataset as described in the previous chapter:

- **Bag-of-Words (BoW)**: vector representation of the *reviewText* and *summary* features of size $|V|$ which has 1 as the i-th component if the i-th word of the vocabulary V is present in the review, 0 otherwise;
- **Bag-of-Words (BoW) + text-free features**: vector representation as before concatenated with the text-free features used in the previous approach;
- **TfIdf**: vector representation similar to the BoW representation but instead, of 1s we have the TfIdf weighting scheme as components;
- **TfIdf + text-free features**: vector representation as before concatenated with the text-free features used in the previous approach;

Initially, we only used the *reviewText* feature to produce the Bag-of-Words representation. In Table 5.7 we can see the results obtained with the Random Forest Classifier and the Multi-Layer Perceptron on the different types of features specified before, ignoring the summary.

Model name	Features	Accuracy Train Set	Accuracy Validate Set
Random Forest	BoW	0.821	0.647
MLP	BoW	0.739	0.642
Random Forest	BoW + text-free features	0.837	0.649
MLP	BoW + text-free features	0.661	0.645
Random Forest	TfIdf	0.876	0.655
MLP	TfIdf	0.735	0.647
Random Forest	TfIdf + text-free features	**0.884**	**0.657**
MLP	TfIdf + text-free features	0.628	0.623

Table 5.7: Accuracy computed for each model on the train and validate set

Then, we proceeded to include also the *summary* feature in the Bag-of-Words representation, and in Table 5.8 we can see an increase in the accuracy compared to the initial approach and the model built only with the *reviewText* feature.

Model name	**Features**	**Accuracy Train Set**	**Accuracy Validate Set**
Random Forest	BoW	0.863	0.715
MLP	BoW	0.799	0.708
Random Forest	BoW + text-free features	0.878	0.714
MLP	BoW + text-free features	0.732	0.719
Random Forest	TfIdf	0.913	**0.727**
MLP	TfIdf	0.800	0.716
Random Forest	TfIdf + text-free features	**0.919**	0.724
MLP	TfIdf + text-free features	0.726	0.720

Table 5.8: Accuracy computed for each model on the train and validate set

From the above results we can see that the best model for our task is the **Random Forest** with the **TfIdf** vector representation of the review text. Regarding the addition of the text-free features, from the results it seems like that including them didn't increase the performance for both of the models.

Moreover, we plotted the feature importance of the Random Forest Classifier which performed the best, trained on the TfIdf representation, obtaining the results shown in Figure 5.5.

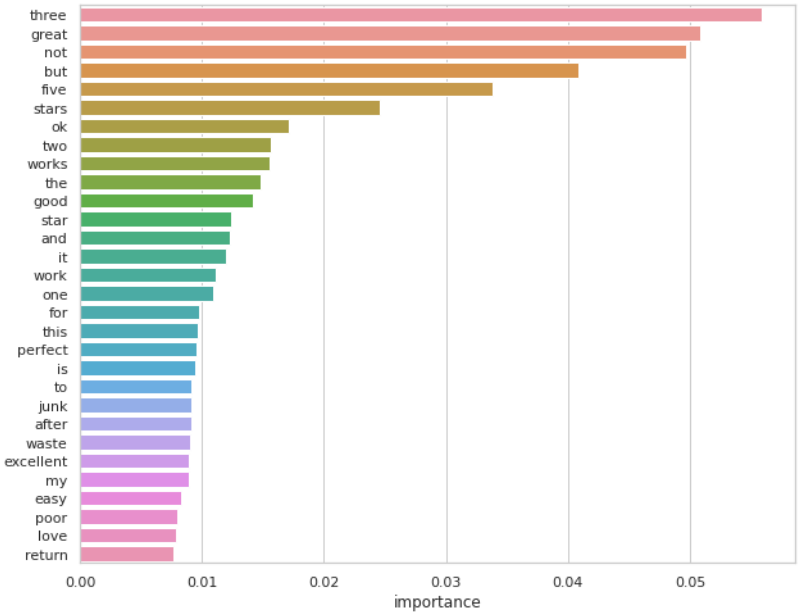


Figure 5.5: Plot of the feature importance computed by the Random Forest Classifier trained on the TfIdf representation

From the feature importance plot we can see that the words that are more important are "three", "great" and "not", indicating that some words are more likely to occur in a certain type of reviews. At the fourth place instead we have "but" which should not be

included in any particular category of reviews, but during the training the model found out that is very discriminative.

In addition, in Figure 5.6, is reported the **confusion matrix** computed both from the train and test set using the best model, showing that there isn't a particular class which is preferred by the classifier, but there are some cases where neutral reviews are confused for negative ones.

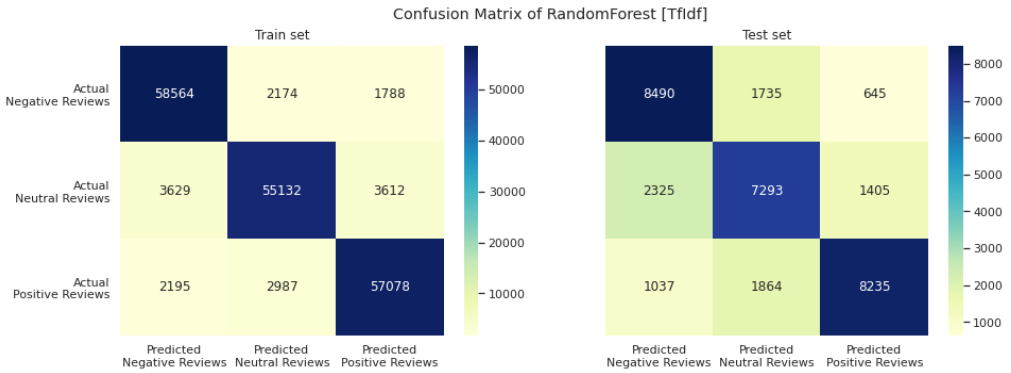


Figure 5.6: Confusion Matrix for the Random Forest Classifier on the complete train set and test set using TfIdf representation

Finally in Table 5.9 and Table 5.10 are presented the **classification reports** achieved by the best model on the train and test set.

Class	Precision	Recall	F1-score	Support
Class 0	0.91	0.94	0.92	62526
Class 1	0.91	0.88	0.90	62373
Class 2	0.91	0.92	0.92	62260
Accuracy			0.91	187159
Macro avg	0.91	0.91	0.91	187159
Weighted avg	0.91	0.91	0.91	187159

Table 5.9: Classification report of the Random Forest Classifier on the train set

Class	Precision	Recall	F1-score	Support
Class 0	0.72	0.78	0.75	10870
Class 1	0.67	0.66	0.67	11023
Class 2	0.80	0.74	0.77	11136
Accuracy			0.73	33029
Macro avg	0.73	0.73	0.73	33029
Weighted avg	0.73	0.73	0.73	33029

Table 5.10: Classification report of the Random Forest Classifier on the test set

5.4 Embeddings approach

The last approach we tested is based on the use of the **GloVe** model, which gives us embeddings of dimension 100. Moreover, we defined the maximum sequence length as

1000, which determines the maximum amount of words of the review that are processed, and the maximum number of words as 2048, which instead determines the size of the vocabulary V.

Since we built two different models, one based on **CNNs** and one based on **LSTMs**, we are going to present their results separately in the following two subsections.

Both models have been trained for **10 epochs**, since we didn't have a lot of computational resources, and it seemed a pretty reasonable number to make the model express their full potential, since training for more epochs didn't bring us significant increase on the performance. The batch size has been fixed to 128 and the validation split to 0.2. Moreover, while training, samples have been shuffled to avoid feeding them to the models always in the same order. During the training phase we observed that the first model takes 265s per epoch running on a GPU-enabled **Google Colab** instance, instead the second one takes 216s per epoch running on the same machine. This time is mainly due to the number of parameters which compose the *Embedding* layer, which is more than **10 million**.

5.4.1 Convolutional Neural Network (CNN)

In this section we show our results for the CNN-based model. First of all, in Figure 5.7 we can see the trend of the **loss** on the train and validate set for each epoch. Instead, in Figure 5.8 we can see the changes in the training and validation accuracy during the training phase.

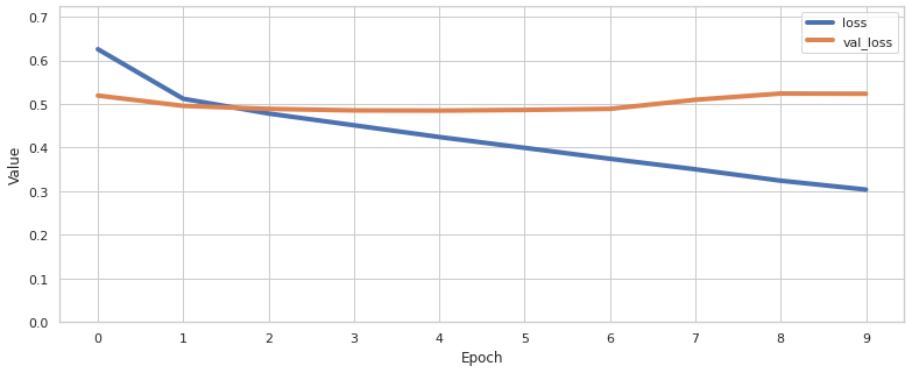


Figure 5.7: Plot of the train and validation loss of the CNN model

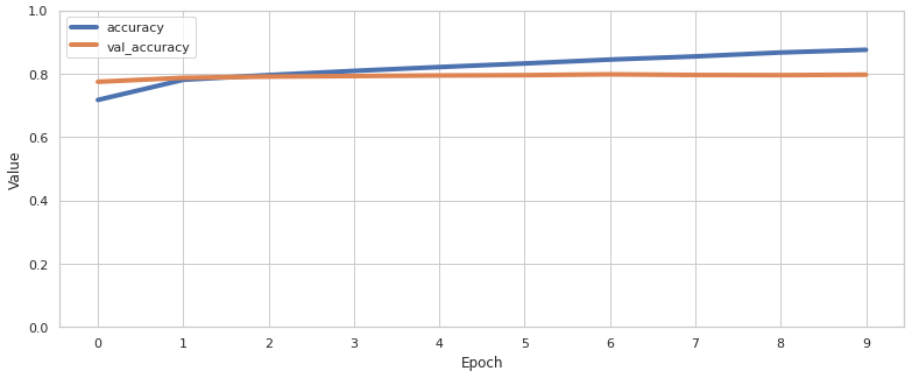


Figure 5.8: Plot of the train and validation accuracy of the CNN model

From the first plot, we can see that the training loss is **decreasing** as it should, so the model is correctly learning, while the validation loss, decreases until epoch 6, while after that it starts to slightly **increase**, indicating that we might start to see a little bit of overfitting.

Instead, in the second plot we can see that the **accuracy** on the validation set already starts from a high value (0.76), and then slightly increases reaching about 0.80.

Moreover, in Figure 5.9 is shown the **confusion matrix** computed over the train and test set, while in Table 5.11 and Table 5.12 are reported respectively the **classification report** on the train and on the test set.

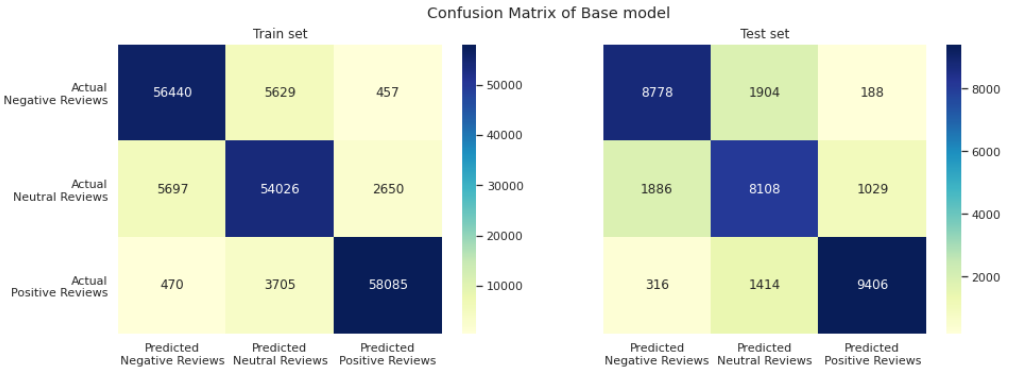


Figure 5.9: Confusion Matrix for the CNN on the complete train set and test set

Class	Precision	Recall	F1-score	Support
Class 0	0.90	0.90	0.90	62526
Class 1	0.85	0.87	0.86	62373
Class 2	0.95	0.93	0.94	62260
Accuracy			0.90	187159
Macro avg	0.90	0.90	0.90	187159
Weighted avg	0.90	0.90	0.90	187159

Table 5.11: Classification report of the CNN on the train set

Class	Precision	Recall	F1-score	Support
Class 0	0.80	0.81	0.80	10870
Class 1	0.71	0.74	0.72	11023
Class 2	0.89	0.84	0.86	62260
Accuracy			0.80	33029
Macro avg	0.80	0.80	0.80	33029
Weighted avg	0.80	0.80	0.80	33029

Table 5.12: Classification report of the CNN on the test set

5.4.2 Long Short-Term Memory (LSTM)

Instead, in this section we show our results for the LSTM-based model. The **loss** and **accuracy** for the train and validation sets are shown in Figure 5.10 and Figure 5.11.

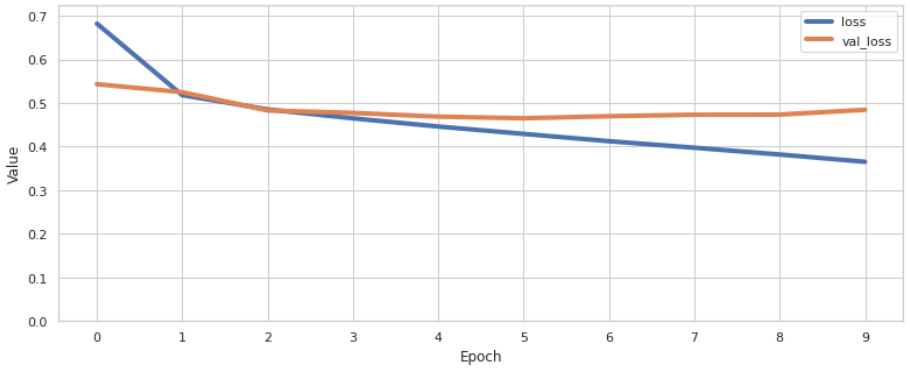


Figure 5.10: Plot of the train and validation loss of the LSTM model

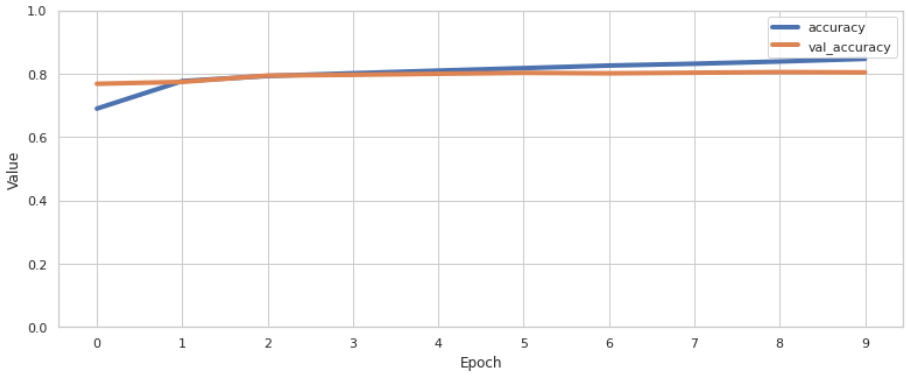


Figure 5.11: Plot of the train and validation accuracy of the LSTM model

Also in these results we can see that the training loss is still **decreasing**, but the validation loss, in the last epochs, starts to **increase**. Regarding the accuracy instead, it is always increasing both on the training and validation set.

In addition, in Figure 5.12 is shown the **confusion matrix** produced by the LSTM both on the train and on the test set, and in Table 5.13 and Table 5.14 are shown the **classification reports** on both sets.

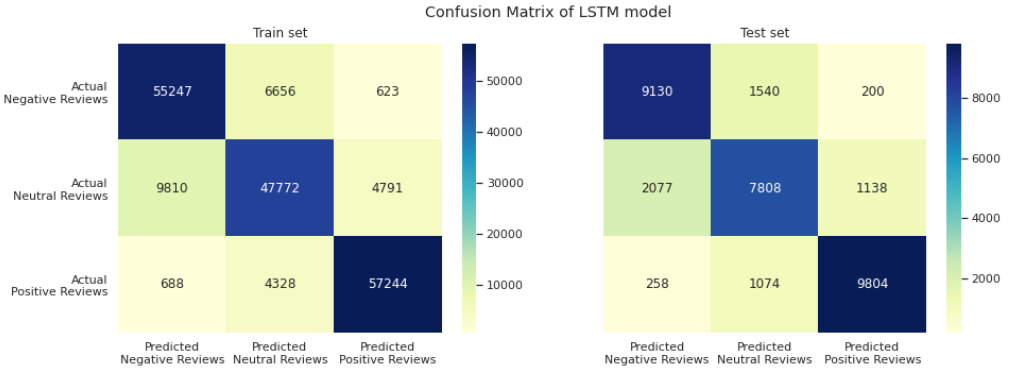


Figure 5.12: Confusion Matrix for the LSTM on the complete train set and test set

From the classification report, we can see an interesting property of the model: as

predictable the most difficult class to predict are the neutral reviews, since it is easier to classify them either as positive or negative.

Class	Precision	Recall	F1-score	Support
Class 0	0.84	0.88	0.86	62526
Class 1	0.81	0.77	0.79	62373
Class 2	0.91	0.92	0.92	62260
Accuracy			0.86	187159
Macro avg	0.86	0.86	0.86	187159
Weighted avg	0.86	0.86	0.86	187159

Table 5.13: Classification report of the LSTM on the train set

Class	Precision	Recall	F1-score	Support
Class 0	0.80	0.84	0.82	10870
Class 1	0.75	0.71	0.73	11023
Class 2	0.88	0.88	0.88	11136
Accuracy			0.81	33029
Macro avg	0.81	0.81	0.81	33029
Weighted avg	0.81	0.81	0.81	33029

Table 5.14: Classification report of the LSTM on the test set

5.5 Comparison and Final results

In order to provide a final comparison among all the approaches, we reported in Table 5.15 the results obtained by the **best model** for each approach. As we can see, it is obvious that the best approach is the one based on **embeddings** since it is also the most complex.

Approach	Best model	Test Accuracy
Text-free	Random Forest	0.45
BoW	Random Forest (TfIdf)	0.73
Embeddings	LSTM	**0.81**

Table 5.15: Comparison of the best model for each approach

Moreover, we wanted to show how the model we designed could be used in **practice**, so we picked the first review we showed in Figure 1.1 at the beginning of the report, and another review from the electronics category of Amazon.com, shown in Figure 5.13. We then fed both reviews into the CNN and LSTM models to predict if they are positive, neutral or negative.

The first review, has been predicted as neutral by the CNN model, while negative from the LSTM, which seems pretty **satisfactory** since the consumer has found some flaws in the product, even if he submitted a rating of four stars, indicating instead that the product was quite good. The second one instead has been predicted as neutral by the CNN and as positive by the LSTM. From this one we can see that the LSTM model performs a little bit better, since it is surely a positive review.

Figure 5.13: A positive review of an headset on Amazon.com

We also found a [paper] which tried to address our **same task** using paragraph vectors to produce a vector representation of the reviews, and concatenated them to a latent representation of the product. At the end, they used an SVM to predict the sentiment expressed by the review.

In Table 5.16 is reported a **comparison** among the already shown results obtained by this project, and the results published by the cited paper. Inspecting the outcomes we can see that the performance we were able to achieve are pretty comparable with the state-of-the-art.

While comparing the results with the paper, we noticed that they showed the distribution of the target variable but they never mentioned in the paper about **re-balancing** it. We suspect that their results can be affected by this choice to not address this problem, since they obtain a low precision and recall compared to the achieved accuracy. In any case, looking at the precision and recall values reported in the paper, we can conclude that our model can achieve better results.

Metrics	**Precision**	**Recall**	**Accuracy**
Our - CNN	0.80	0.80	0.80
Our - LSTM	**0.81**	**0.81**	0.81
Paper - SVM	0.59	0.43	**0.82**

Table 5.16: Comparison among our best models and the state-of-the-art

Chapter 6

Conclusions

In conclusion we can say that in this project we investigated possible approaches to detect the **sentiment** of a review in order to detect inconsistent reviews, using different approaches and different **text representations**. From the results, we saw that reviews **metadata** are not so relevant to detect the sentiment, but, as predictable, the most important part of a review is its title and its content. Moreover, we obtained good results using a CNN and an LSTM-based models, achieving **81% of accuracy** in predicting the sentiment of a review.

As possible future studies, we can experiment with more complex models to represent text into a vector space like **BERT**, which seems a promising alternative to the GloVe model, and also to use **Transformers** to process text instead of the LSTM, using the attention mechanism to focus more on some parts of the review to detect the sentiment expressed.