Hogeschool van Amsterdam

Author
Jaime Raynaud – 500884682
BD3

School
Zarina Efendieva & Evert-Jan
Hogeschool van Amsterdam

Date
1/11/2021

Report Data Engineer and Data Scientist individual assignment

Sentiment Analysis on Hotel Reviews

Table of contents

1. Summary

Imagine that you are thinking of going on vacation the next summer, one of the multiple tasks that you have to do before departure is to look for a hotel where to stay. If you enter a site like Booking or TripAdvisor chances are that one of the main reasons to choose a hotel is their reviews. So, nowadays hotels can benefit from all this information that their guests publish on the multiple hotel recommendations sites. Reviews can tell you how the hotel is satisfying the customer's needs, which is crucial for developing marketing strategies.

With the grown of Big Data, tools like Sentiment Analysis can help businesses understand their customers. Sentiment analysis is contextual mining of text which identifies and extracts subjective information in source material, and helps a business to understand the social sentiment of their brand, product or service while monitoring online conversations.

In this report, we will see how using the knowledge acquired during the course Big Data Scientist and Engineer and following a process of data discovery, data preparation, and model building we can predict whether or not a review is positive or negative.

At the end of it, we will see how this forecasting task is satisfied with the Machine Learning techniques that I will use, achieving accuracy in the predictions superior to 90%.

2. Introduction

This report will follow all the necessary steps of a Big Data Project to build a model able to predict if a hotel review is positive or negative.

To carry out this task I will start describing the Data discovery process, in which I will talk about the different datasets that have been used for this project and how I obtained them.

Secondly, the Data preparation, where we will see where the datasets are stored and which processes are needed to handle the data, so the models can work with it, this part is crucial for a project like this.

In third place, one of the main parts of this project, the Model building. Four different classifiers have been selected, the theory behind them will be explained and we will see different techniques to improve their efficacy, like data splitting and parameter tunning techniques.

Finally, I will discuss the results obtained, talking about different metrics like accuracy, confusion matrix, and other scores, explaining their theoretical background, and discussing if the results are satisfactory or not. Also, will talk about future possible upgrades in the models and other stages of the project so I could even improve their scores in future versions.

To be able to carry out this task we will use the Visual Studio environment, programming in Python using useful libraries for Data Science like Pandas or Scikit-Learn. Also, to store the data in a database I will use MySQL database.

3. Methods

In this section, we will see the different methods that have been applied to create a model capable of predicting if a review is positive or negative.

3.1. Data discovery

Data discovery involves the collection and evaluation of data from various sources and is often used to understand trends and patterns in the data.

It requires a progression of steps that organizations can use as a framework to understand their data.

Data discovery, usually associated with business intelligence (BI), helps inform business decisions by bringing together disparate, siloed data sources to be analyzed. Having mounds of data is useless unless you find a way to extract insights from it.

The data discovery process includes connecting multiple data sources, cleansing and preparing the data, sharing the data throughout the organization, and performing analysis to gain insights into business processes.

In this case, the first step into this phase is obtaining the data from three different sources:

3.1.1. Kaggle Dataset

As the main source of our data, this dataset contains 515.000 customer reviews and scoring of 1493 luxury hotels across Europe. Meanwhile, the geographical location of hotels is also provided for further analysis, but will not be used for our forecasting task. Originally, it has been scraped from Booking.com.

The CSV file contains 17 fields. The description of each field is as below:

-Review_Date: Date when reviewer posted the corresponding review.
-Hotel_Address: Address of hotel.
-Average_Score: Average Score of the hotel, calculated based on the latest comment in the last year.
-Hotel_Name: Name of Hotel
-Reviewer_Nationality: Nationality of Reviewer
-Negative_Review: Negative Review the reviewer gave to the hotel. If the reviewer does not give a negative review, then it should be: 'No Negative'
-ReviewTotalNegativeWordCounts: Total number of words in the negative review.
-Positive_Review: Positive Review the reviewer gave to the hotel. If the reviewer does not give a negative review, then it should be: 'No Positive'
-ReviewTotalPositiveWordCounts: Total number of words in the positive review.
-Reviewer_Score: Score the reviewer has given to the hotel, based on his/her experience

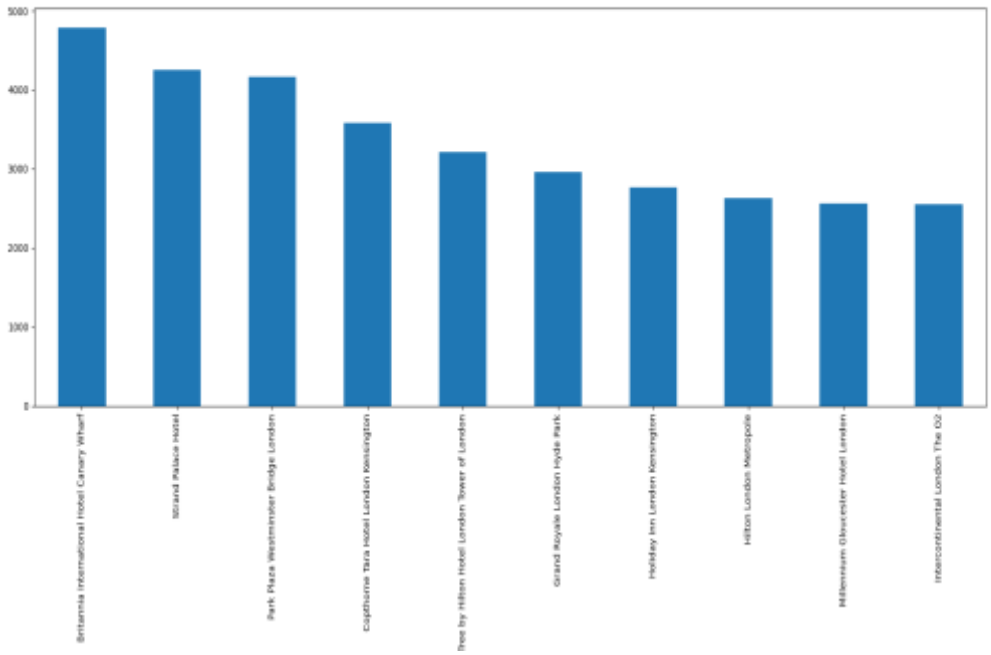
-TotalNumberofReviewsReviewerHasGiven: Number of Reviews the reviewers have given in the past.
-TotalNumberof_Reviews: Total number of valid reviews the hotel has.
-Tags: Tags reviewer gave the hotel.
dayssincereview: Duration between the review date and scrape date.
-AdditionalNumberof_Scoring: There are also some guests who just made a scoring on the service rather than a review. This number indicates how many valid scores without review in there.
-lat: Latitude of the hotel
-lng: longitude of the hotel

From this dataset, I just need the Negative_Review and Positive_Review, as our final dataset will have two features (*review* and *is_positive*, this last one being 1 when the review is positive and 0 when is negative). For this reason, I will drop the unnecessary columns and use the melt method from pandas to generate just one column *review.*

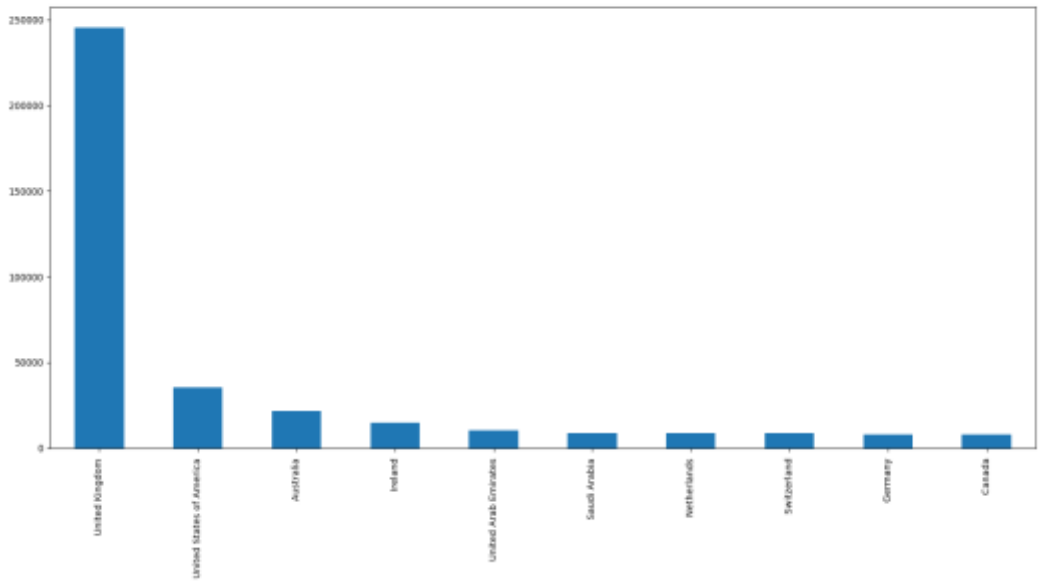
```
def return_df():
    df = pd.read_csv('Hotel_Reviews.csv', header=0)
    df = df.drop(columns =['Hotel_Address', 'Additional_Number_of_Scoring', 'Review_Date',  'Average_Score',
                  'Hotel_Name', 'Reviewer_Nationality', 'Review_Total_Negative_Word_Counts',
                  'Total_Number_of_Reviews',    'Review_Total_Positive_Word_Counts',
                  'Total_Number_of_Reviews_Reviewer_Has_Given', 'Reviewer_Score',   'Tags', 'days_since_review',
                  'lat',    'lng'])
    df = df.melt(value_vars=['Positive_Review', 'Negative_Review'], var_name = 'rating', value_name = 'review')
    df["rating"].replace({"Positive_Review": 1, "Negative_Review": 0}, inplace=True)

    df = df[["review", "rating"]]
    return df
```

As the result of melting the two columns Positive and Negative review, I will obtain the double of rows that I had at the beginning.

As a first insight into this major part of the data, I created some plots. First, the top ten hotels with more reviews:

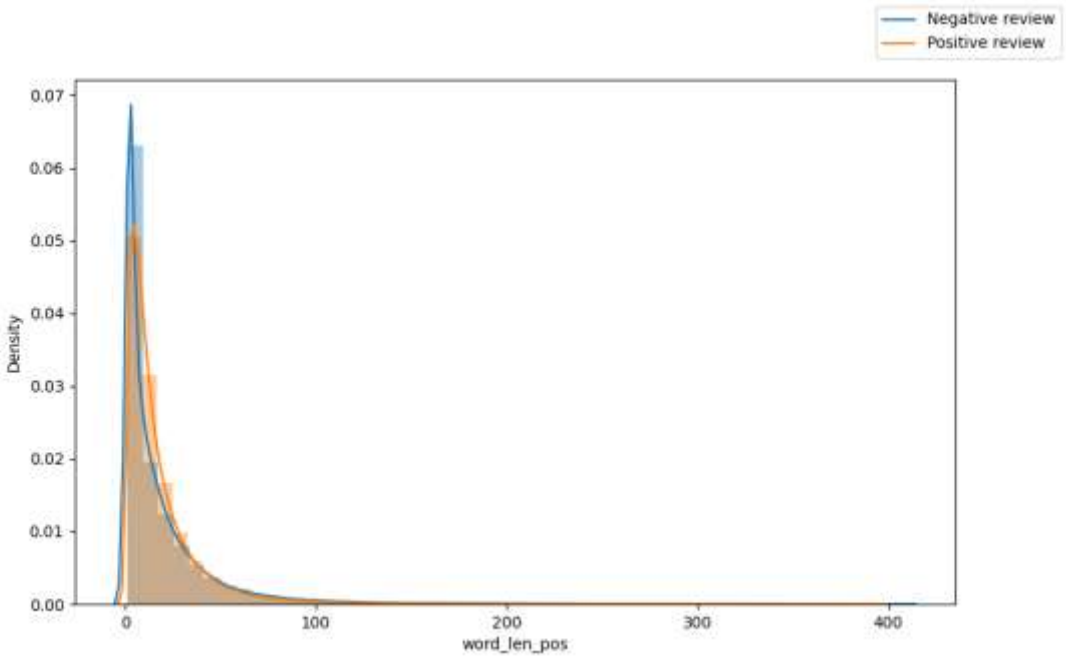


Secondly, the top ten nationalities with more reviews:



Being the United Kingdom the first one by far, with almost 250000 reviews.

And finally, the distribution of the number of words per positive and negative review:



We can see that the distribution is pretty similar, negative and positive reviews have the same length in general, so we should not have this into account in our research.

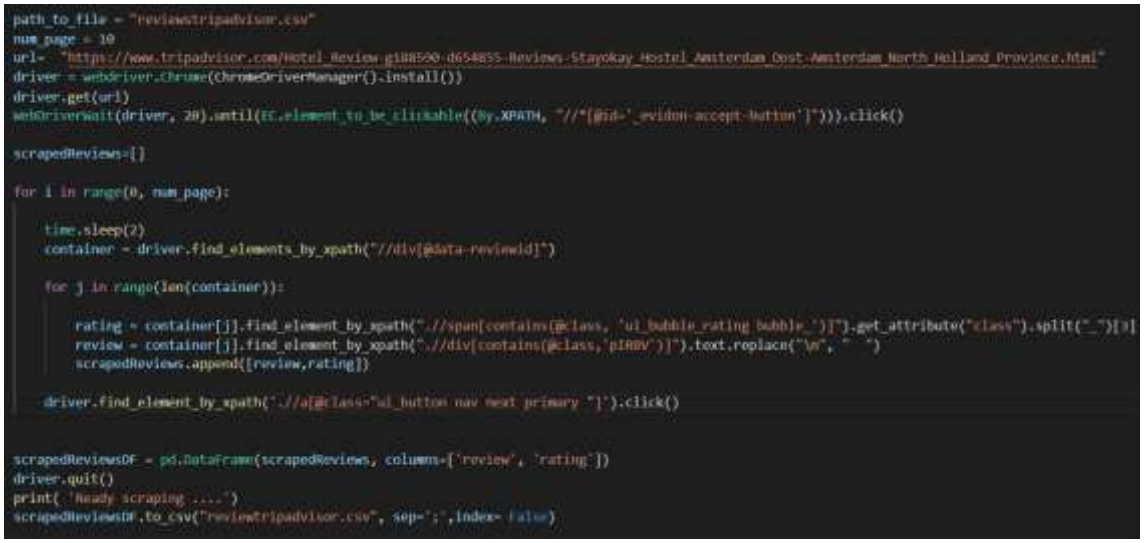
With this, we have our Kaggle dataset prepared to be cleaned.

3.1.2. Webscraped dataset

Using the webscrapping knowledge acquired during the classes, I managed to obtain 50 reviews from Tripadvisor and 100 from Booking, in particular, I selected the hotel Stayokay Hostel Amsterdam Oost because is the hostel where I stayed on my first night here in Amsterdam.

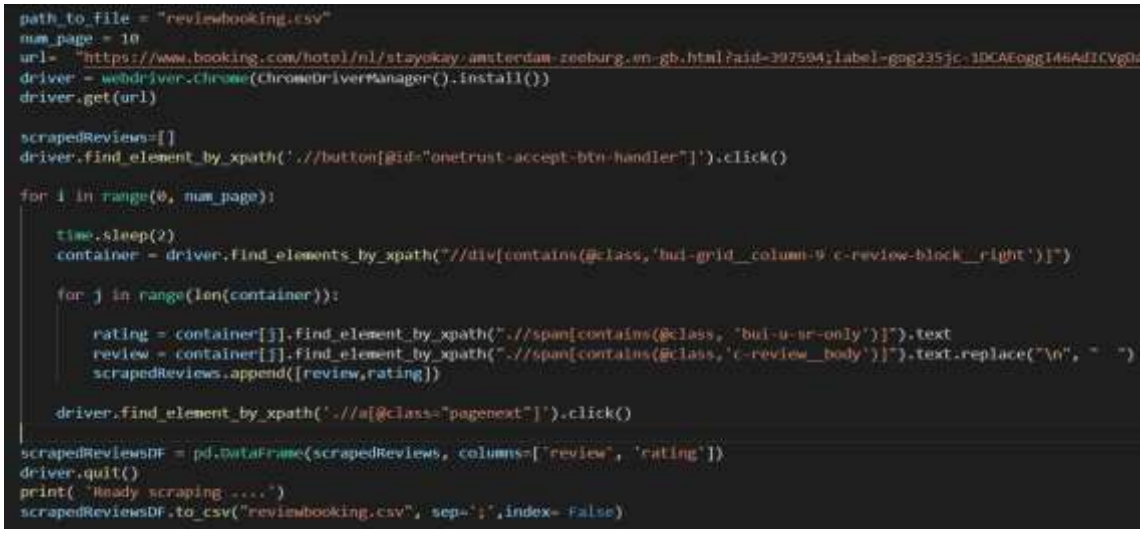
For the Web scrapping, I created two scripts, one for TripAdvisor and one for Booking, both based on the Web scraping script uploaded to the DLO, but of course with modifications to make it work in Booking.

Both of them follow the same structure, the web scrapping program will go to the URL, click in the needed buttons to see the reviews, and after that start collecting the reviews through two for loops, one for the pages, and one for the reviews. To indicate where is the “place” where each element of the web page is located I just used the *find_elements_by_xpath* method of the Chrome web driver.



```
path_to_file = "reviewstripadvisor.csv"
num_page = 10
url=  "https://www.tripadvisor.com/Hotel_Review-g188590-d654855-Reviews-Stayokay_Hostel_Amsterdam_Oost-Amsterdam_North_Holland_Province.html"
driver = webdriver.Chrome(ChromeDriverManager().install())
driver.get(url)
WebDriverWait(driver, 20).until(EC.element_to_be_clickable((By.XPATH, "//*[@id='_evidon-accept-button']"))).click()

scrapedReviews=[]

for i in range(0, num_page):

    time.sleep(2)
    container = driver.find_elements_by_xpath("//div[@data-reviewid]")

    for j in range(len(container)):

        rating = container[j].find_element_by_xpath(".//span[contains(@class, 'ui_bubble_rating bubble_')]").get_attribute("class").split("_")[3]
        review = container[j].find_element_by_xpath(".//div[contains(@class,'pIRBV')]").text.replace("\n", " ")
        scrapedReviews.append([review,rating])

    driver.find_element_by_xpath('.//a[@class="ui_button nav next primary "]').click()


scrapedReviewsDF = pd.DataFrame(scrapedReviews, columns=['review', 'rating'])
driver.quit()
print( 'Ready scraping ....')
scrapedReviewsDF.to_csv("reviewtripadvisor.csv", sep=';',index= False)
```

TripAdvisor web scraper

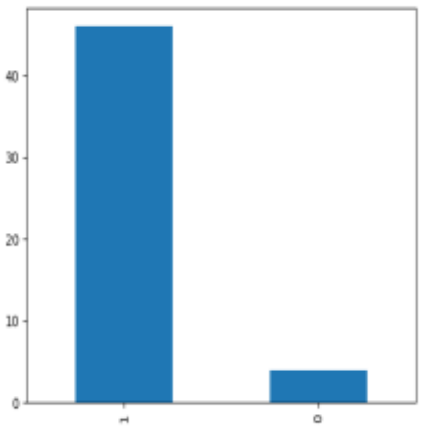


```
path_to_file = "reviewbooking.csv"
num_page = 10
url=  "https://www.booking.com/hotel/nl/stayokay-amsterdam-zeeburg.en-gb.html?aid=397594;label=gog235jc-1DCAEoggI46AdICVgD
driver = webdriver.Chrome(ChromeDriverManager().install())
driver.get(url)

scrapedReviews=[]
driver.find_element_by_xpath('.//button[@id="onetrust-accept-btn-handler"]').click()

for i in range(0, num_page):

    time.sleep(2)
    container = driver.find_elements_by_xpath("//div[contains(@class,'bui-grid__column-9 c-review-block__right')]")

    for j in range(len(container)):

        rating = container[j].find_element_by_xpath(".//span[contains(@class, 'bui-u-sr-only')]").text
        review = container[j].find_element_by_xpath(".//span[contains(@class,'c-review__body')]").text.replace("\n", " ")
        scrapedReviews.append([review,rating])

    driver.find_element_by_xpath('.//a[@class="pagenext"]').click()

scrapedReviewsDF = pd.DataFrame(scrapedReviews, columns=['review', 'rating'])
driver.quit()
print( 'Ready scraping ....')
scrapedReviewsDF.to_csv("reviewbooking.csv", sep=';',index= False)
```

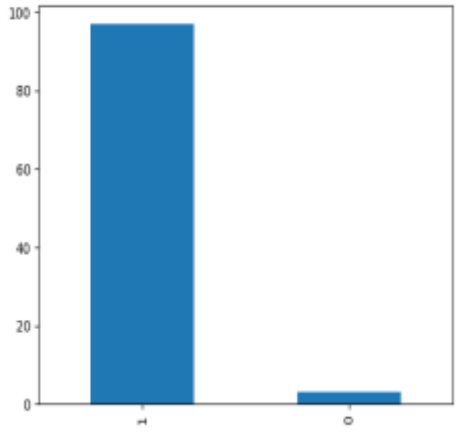
Booking web scraper

From TripAdvisor, I just got the text of the review and the rating from 0 to 50 that we can find in the HTML. I transformed this rating to 1 (positive) if the rating is greater or equal to 30 and 0 (negative) in other cases.

Booking works differently, here I have two reviews per user, a positive review and a negative one, this can be noticed because of the Liked or Disliked element in the HTML. Having this into account, I did the same as with TripAdvisor, generated a dataset with the text of the review, and 1 for positive reviews and 0 for negatives.



TripAdvisor reviews



Booking reviews

Now we have the other two datasets ready for the next stage.

3.1.3. Handwritten reviews

I created another dataset containing three handwritten reviews, one positive and one negative. These are the reviews:

Positive: I like this hotel, I slept very well and everything was clean

Negatives:

I don't like this hotel, the sheets were dirty and the staff was rude.

The smell in the bathroom was really bad, also the food in the restaurant was disgusting.

The final part of this stage of the project was to create one final dataset from these 4 datasets mentioned before. I used the *concat* method from Pandas to join the 4 datasets, right after that, I had to mix all the data with the *sample* method (using *frac=1* to get all the data) and reset the indexes, because the *concat* method keeps the indexes from the previous datasets and this would be a problem when manipulating the data.

```python
def return_df_final():
    df = return_df()
    df_tripadvisor = return_df_tripadvisor()
    df_booking = return_df_booking()
    df_add = return_df_add()
    dataframes = [df, df_tripadvisor, df_booking, df_add]

    df_final = pd.concat(dataframes)
    df_final = df_final.sample(frac=1, random_state=42).reset_index(drop=True)
    df_final = df_final.rename(columns={"rating": "is_positive"})
    return df_final
```

With this, I have my dataset ready to be cleaned and uploaded to the database.

3.2. Data preparation

Data preparation is the process of cleaning and transforming raw data prior to processing and analysis. It is an important step prior to processing and often involves reformatting data, making corrections to data, and combining data sets to enrich data.

Data preparation is often a lengthy undertaking for data professionals or business users, but it is essential as a prerequisite to put data in context in order to turn it into insights and eliminate bias resulting from poor data quality.

For example, the data preparation process usually includes standardizing data formats, enriching source data, and/or removing outliers.

In the particular case of this project, I will dedicate this section to explaining how the data has been stored and processed so the models can work with them.

3.2.1. Database

First of all, I will talk about the database that I have created. Its name is *zipcode*, containing 3 different tables, one for the dataset that I talked about before and the other two tables for the stemmed and lemmatized datasets, I will explain this later.

For now, I will just talk about the *hotelreviews* table, which contains the data without cleaning. This table contains 1031628 reviews that have been uploaded to the database using the function:

```python
def upload_data(df):
    engine = create_engine("mysql+mysqlconnector://root:root@localhost:3307/zipcode",
        connect_args={'connect_timeout': 300})
    try:
        df.to_sql(name='hotelreviews',con=engine, if_exists='replace',index=False,chunksize=1000)
        print('Succesfully uploaded data')
    except Exception as e:
        print('Something went wrong:', e)
```

Also, I have created some stored procedures to interact with the data, like:

- ***select_all()***: That returns all the data from the *hotelreviews* table.
- ***select_all_stem()***: That returns all the data from the *hotelreviews_stem* table.
- ***select_all_lemmatize()***: That returns all the data from the *hotelreviews_lemmatize* table.
- ***delete_table(tabla)*:** That drops the table table.
- ***delete_reviews():*** This stored procedure cleans the given table of reviews that just contain certain words like *Nothing*, *No Negative*, *No Positive, Everything,* etc. I decided to get rid of these reviews because they will make the models perform worst, this is because 188429 reviews are just the shown words.

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `delete_reviews`()
BEGIN
    SET SQL_SAFE_UPDATES = 0;
    DELETE FROM hotelreviews
    WHERE review = 'No Negative' OR review = 'No Positive' OR review like ' No Negative' OR review like ' No Positive'
    OR review like ' Nothing' OR review like ' Not a thing'
    OR review like ' nothing' OR review like ' NA' OR review like ' None' OR review like ' Everything'
    OR review like ' NOTHING' OR review like ' EVERYTHING' OR review like ' everything' OR review like ' N a';
    SET SQL_SAFE_UPDATES = 1;
END
```

With all of these stored procedures, I can do a partially clean of the data, delete the reviews that are equal to the words shown in the screenshot, and get the data cleaned from the database. The procedures *delete_reviews()* and *select_all()* are called in Python with the following function:

```
def return_db_from_mysql(procedure):
    try:
        connection = mysql.connector.connect(
            host="localhost",
            port=3307,
            user="root",
            password="root",
            database="zipcode")
        cursor = connection.cursor()
        cursor.callproc('delete_reviews')
        cursor.callproc(procedure)

        for result in cursor.stored_results():
            results = result.fetchall()
        df = pd.DataFrame(results)
        df = df.rename(columns = {0:'review', 1:'is_positive'})

    except mysql.connector.Error as error:
        print("Failed to execute stored procedure: {}".format(error))
    finally:
        if (connection.is_connected()):
            cursor.close()
            connection.close()
            print("MySQL connection is closed")

    return df
```

That creates a MySQL connector to call the procedures one after the other so I can obtain the database cleaned, the function has a parameter procedure so I can call the *select_all_stem()* or the *select_all_lemmatize()* as well. Also, I had to use the *rename* Pandas method to rename the columns, because I was getting them back with the names 0 for *review* and 1 for *is_positive*.

With this, all the MySQL part of the project is explained, and our data would be uploaded to the database and partially cleaned on it.

As part of the cleaning of the data, I have a function in Python that transforms all the text into lowercase, as a common practice in Sentiment analysis, for a better performance of the models so they don't difference words if they have capital letters. Also, this function deletes all the numbers from the reviews to not interfere with the models. For this, I used the *replace* Pandas method with the regular expression *'\d+'* that matches one or more decimal digits. Finally, we drop the rows that contain empty reviews with the *dropna* Pandas method, right after this, we have to use *reset_index* so our indexes are well placed again.

3.2.2. Wordclouds

Once we have our data completely cleaned, it is time to show the word clouds. But, what is a word cloud?

A word cloud is a collection, or cluster, of words depicted in different sizes. The bigger and bolder the word appears, the more often it's mentioned within a given text and the more important it is.

Word clouds (also known as text clouds or tag clouds) work in a simple way: the more a specific word appears in a source of textual data (such as a speech, blog post, or database), the bigger and bolder it appears in the word cloud.

Also known as tag clouds or text clouds, these are ideal ways to pull out the most pertinent parts of textual data, from blog posts to databases. They can also help business users compare and contrast two different pieces of text to find the wording similarities between the two.

Particularly, I have generated 3 word clouds, one general word cloud, one for the positive reviews and one for the negative ones, positive and negative ones getting rid of stopwords and words like: hotel, room, staff, rooms, breakfast, and bathroom, since they would occupy the majority of the word cloud and would not let us see other interesting words. With the following code:

```
def display_wordcloud(df):
    my_stopwords = ENGLISH_STOP_WORDS.union(['hotel', 'room', 'staff', 'rooms', 'breakfast', 'bathroom'])

    my_cloud = WordCloud(background_color='white',stopwords=ENGLISH_STOP_WORDS).generate(' '.join(df['review']))
    plt.imshow(my_cloud, interpolation='bilinear')
    plt.axis("off")
    plt.show()


    df_positive = df[df['is_positive']==1]
    my_cloud_positive = WordCloud(background_color='white',stopwords=my_stopwords).generate(' '.join(df_positive['review']))
    plt.imshow(my_cloud_positive, interpolation='bilinear')
    plt.axis("off")
    plt.show()

    df_negative = df[df['is_positive']==0]
    my_cloud_negative = WordCloud(background_color='white',stopwords=my_stopwords).generate(' '.join(df_negative['review']))
    plt.imshow(my_cloud_negative, interpolation='bilinear')
    plt.axis("off")
    plt.show()
```

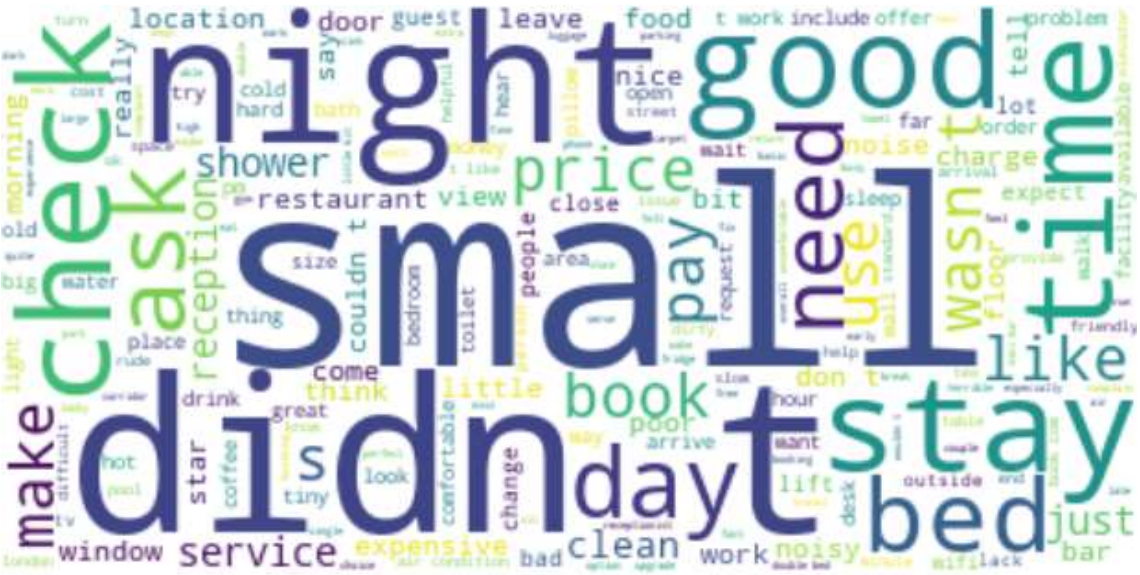
The word clouds generated are:

General Word Cloud

We can see the importance of words like staff, stay, room or hotel. This information will be useful in future stages of the project.

Positive Word Cloud

It is easy to visualize that this is the positive word cloud, we can find interesting words for our research like friendly, helpful or clean, between others, that indicate that is a good review.



Negative Word Cloud

With the negative word cloud, I find it more difficult, although there are some words like small, poor, or little that can be indicators of a bad review.

3.2.3. Stemming and Lemmatization

After seeing the word clouds it is time to do another modification of our data. Having completed the Sentiment analysis DataCamp course, one of the most important steps is Stemming and Lemmatizing, but why they are useful for our research?

The purpose of both stemming and lemmatization is to reduce morphological variation. In the context of machine learning-based NLP, stemming makes your training data denser. It reduces the size of the dictionary (number of words used in the corpus) two or three-fold.

Having the same corpus, but fewer input dimensions, ML will work better.

The downside is, if in some cases the actual word (as opposed to its stem) makes a difference, then your system won't be able to leverage it. So you might lose some precision.

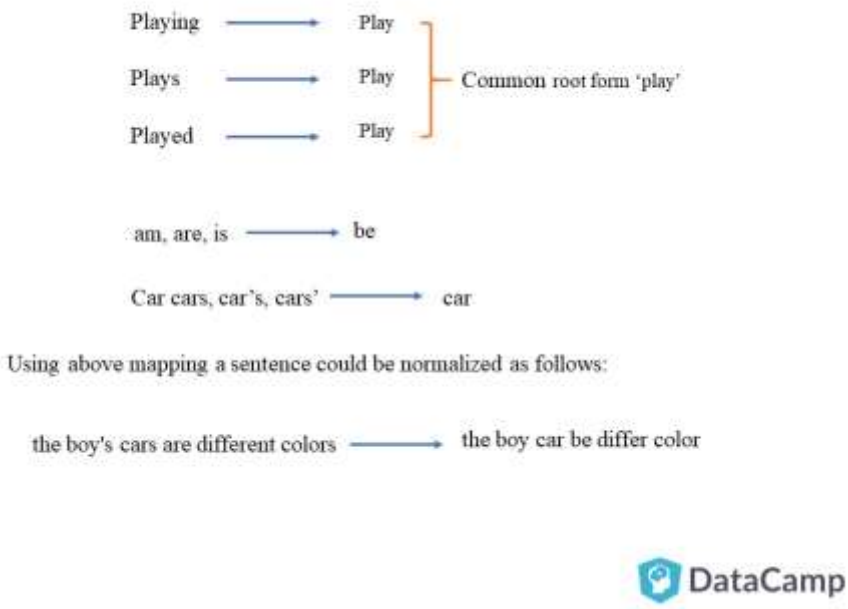
First of all, I will explain the theory behind these two:

Stemming and Lemmatization are Text Normalization (or sometimes called Word Normalization) techniques in the field of Natural Language Processing that are used to prepare text, words, and documents for further processing.

When a language contains words that are derived from another word, as their use in the speech changes, is called Inflected Language.

In grammar, inflection is the modification of a word to express different grammatical categories such as tense, case, voice, aspect, person, number, gender, and mood. An inflection expresses one or more grammatical categories with a prefix, suffix or infix, or another internal modification such as a vowel change.

The degree of inflection may be higher or lower in a language. As you have read the definition of inflection with respect to grammar, you can understand that an inflected word(s) will have a common root form. Let's look at a few examples,



Stemming and Lemmatization helps us to achieve the root forms (sometimes called synonyms in search context) of inflected (derived) words.

In particular for Stemming, Stem (root) is the part of the word to which you add inflectional (changing/deriving) affixes such as (-ed,-ize, -s,-de,mis). So stemming a word or sentence may result in words that are not actual words. Stems are created by removing the suffixes or prefixes used with a word.

To apply **Stemming**, I have followed the DataCamp course, creating the next function:

```
def stemming(df):
    # Import the function to perform stemming
    # Call the stemmer
    new_column = []
    porter = PorterStemmer()    # Transform the array of tweets to tokens

    tokens = [word_tokenize(review) for review in df['review']]
    # Stem the list of tokens
    stemmed_tokens = [[porter.stem(word) for word in review] for review in tokens]
    for l in stemmed_tokens:
        stem_sentence = " ".join(l)
        new_column.append(stem_sentence)
        #print(stem_sentence)
    #print(stemmed_tokens)
    df['stem_review'] = new_column
    return df[['stem_review', 'is_positive']]
```

For all this process I have used the Natural Language Tool Kit (NLTK), which is a Python library to make programs that work with natural language.

To do the Stemming I have used the Porter Stemming Algorithm, PorterStemmer uses Suffix Stripping (remove the suffix from a word) to produce stems.

Why use it? Because PorterStemmer is known for its simplicity and speed, opposed to LancasterStemmer, that is simple, but heavy stemming due to iterations and over-stemming may occur.

The Stemmer sees every review as a word, so I have to separate each sentence into words and stem each word. For this, we use the NLTK tokenizer.

Now we have just to stem each word and join them again to create a stemmed review, after this, we create a new column for our dataframe and the stemming will be completed.

Let's see the theory behind **Lemmatization**:

Lemmatization, unlike Stemming, reduces the inflected words properly ensuring that the root word belongs to the language. In Lemmatization root word is called Lemma. A lemma (plural lemmas or lemmata) is the canonical form, dictionary form, or citation form of a set of words.

I have used Python NLTK provides WordNet Lemmatizer that uses the WordNet Database to lookup lemmas of words.

My code looks as it follows:

```
def pos_tagger(nltk_tag):
        if nltk_tag.startswith('J'):
            return wordnet.ADJ
        elif nltk_tag.startswith('V'):
            return wordnet.VERB
        elif nltk_tag.startswith('N'):
            return wordnet.NOUN
        elif nltk_tag.startswith('R'):
            return wordnet.ADV
        else:
            return None

def lemmatize(df):
    new_column = []
    lemmatizer = WordNetLemmatizer()

    for sentence in df['review']:
        #print(sentence)
        pos_tagged = pos_tag(word_tokenize(sentence))
        wordnet_tagged = list(map(lambda x: (x[0], pos_tagger(x[1])), pos_tagged))

        lemmatized_sentence = []
        for word, tag in wordnet_tagged:
            if tag is None:
                lemmatized_sentence.append(word)
            else:
                lemmatized_sentence.append(lemmatizer.lemmatize(word, tag))
        lemmatized_sentence = " ".join(lemmatized_sentence)
        #print(lemmatized_sentence)
        new_column.append(lemmatized_sentence)
    df['lemmatized_review'] = new_column
    return df[['lemmatized_review', 'is_positive']]
```

As you can see, there is a function called *pos_tagger(nltk_tag).* The process to apply lemmatization is the same as with the stemming function but, this time, we add a tag with a particular word defining its type (verb, noun, adjective, etc).

For Example,

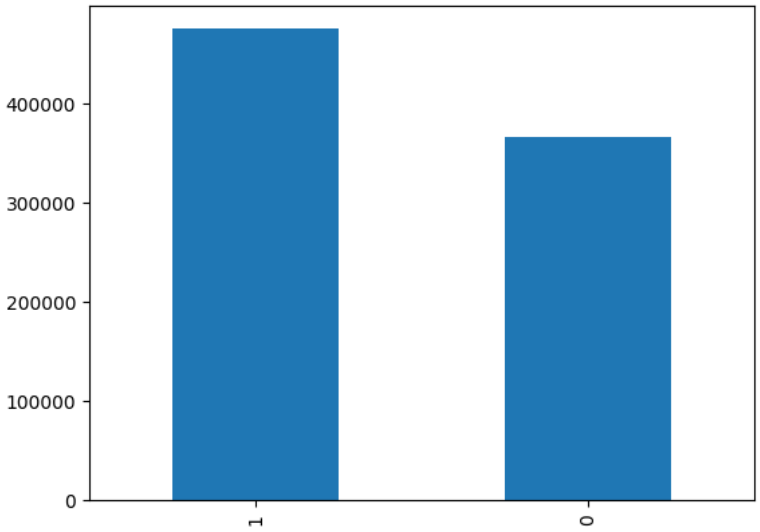
Word	+	Type (POS tag)	—>	Lemmatized Word
driving	+	verb ‘v’	—>	drive
dogs	+	noun ‘n’	—>	dog

So the lemmatizer knows if it should transform the word into an adjective, verb, noun, or adverb.

Following the same process, I do a for loop between the reviews, I tokenize each review into words and apply the *pos_tag* method from *nltk.tag* so I can pass this tag to the *pos_tagger(nltk_tag)* so it gives me the type of each particular word, after that, I apply the lemmatization to each word with its tag. Finally, I join again the review and store it in a new column that I will add to the dataframe.

With this, the lemmatization process of the reviews is done.

After cleaning our data, with 842141 reviews looks like:



Being the percentage of positive reviews: 56.4661% and negative reviews: 43.5339%.

Now that I have two new dataframes, one lemmatized and one stemmed I stored them into the MySQL database with the *upload_data(df)* function so I can have access to both of them at any moment.

3.2.4. Vectorizers

The next step into the Data preparation is to transform the *review* column into multiple columns or features, so our classification models can work with the data. To do this task I used two methods: Count vectorizing and Tfidf transformation, which I will compare to see which one is more effective.

First count vectorizing, applying the method ***CountVectorizer*** from the *sklearn* library that converts a collection of text documents to a matrix of token counts. This is the code to apply it:

```
def count_vectorizer(df, my_stopwords):
    # Build the vectorizer
    vect = CountVectorizer(stop_words=my_stopwords, ngram_range=(1, 2),
                           max_features=1000, token_pattern=r'\b[^\d\W][^\d\W]+\b').fit(df.review)
    # Create the bow representation
    X = vect.transform(df.review)
    # Create the data frame
    X_df = pd.DataFrame(X.toarray(), columns=vect.get_feature_names_out())
    return X_df
```

The function receives a dataframe and a set of stopwords, in my case, it will be the *sklearn.feature_extraction.text ENGLISH_STOP_WORDS*. As parameters for the CountVectorizer we have:

- ***stop_words***: If a list, that list is assumed to contain stop words, all of which will be removed from the resulting tokens.
- ***ngram_range***: The lower and upper boundary of the range of n-values for different word n-grams or char n-grams to be extracted. All values of n such that min_n <= n <= max_n will be used. In my case it will be unigrams and bigrams, so we have a bigger range of options to analyze.
- ***max_features***: As we will discuss later, this parameter is very important. If not None, build a vocabulary that only considers the top max_features ordered by term frequency across the corpus. It will affect considerably the time of training the model and its effectiveness.
- ***token_pattern***: Regular expression denoting what constitutes a “token”. In my case the regular expression *'\b[^\d\W][^\d\W]+\b'* will ignore digits and other characters and only consider words of two or more letters, as it is used in the DataCamp course.

Secondly, the Tfidf transformer, using the *sklearn.feature_extraction.text.* ***TfidfTransformer****.*

It transforms a count matrix to a normalized tf or tf-idf representation.

Tf means term-frequency while tf-idf means term-frequency times inverse document-frequency. This is a common term weighting scheme in information retrieval, that has also found good use in document classification.

The goal of using tf-idf instead of the raw frequencies of occurrence of a token in a given document is to scale down the impact of tokens that occur very frequently in a given corpus and that are hence empirically less informative than features that occur in a small fraction of the training corpus.

The code to apply it looks as follows:

```
def tfidf(df, my_stopwords):
    # Build the vectorizer
    vect = TfidfVectorizer(stop_words=my_stopwords, ngram_range=(1, 2),
                           max_features=1000, token_pattern=r'\b[^\d\W][^\d\W]+\b').fit(df['review'].values.astype('U'))
    # Create sparse matrix from the vectorizer
    X = vect.transform(df['review'].values.astype('U'))
    # Create a DataFrame
    X_df = pd.DataFrame(X.toarray(), columns=vect.get_feature_names_out())
    return X_df
```

I used the same parameters as with the *CountVectorizer*. After that, in both functions, I transform the reviews and create a new dataframe *X_df* that will contain all the new generated features.

Now that I have our data cleaned, uploaded to the database, and created functions to stem, lemmatize, count vectorize and Tfidf transform the reviews is time to make some decisions, as we can not use all these methods at the same time.

The next section will compare results in all the models that I selected, to see which pair of operations I will use in the definitive models.

The code for this model looks like this:

```
def model_comparator(df, df_vect):
    my_stopwords = ENGLISH_STOP_WORDS

    y = df.is_positive
    X = df_vect

    y_pred_logres = cross_val_predict(LogisticRegression(), X, y, cv=3)
    print('Accuracy for LR: ', accuracy_score(y, y_pred_logres))

    y_pred_randomfor = cross_val_predict(RandomForestClassifier(), X, y, cv=3)
    print('Accuracy for RF: ', accuracy_score(y, y_pred_randomfor))

    y_pred_mnb = cross_val_predict(MultinomialNB(), X, y, cv=3)
    print('Accuracy for MNB: ', accuracy_score(y, y_pred_mnb))

    y_pred_gb = cross_val_predict(GradientBoostingClassifier(), X, y, cv=3)
    print('Accuracy GB: ', accuracy_score(y, y_pred_gb))
```

And the results obtained are:

Vectorizer	*Count vectorizer*	*Tfidf*
Word transformation		
Stemming	Accuracy for LR: 0.89526 Accuracy for RF: 0.891 Accuracy for MNB: 0.87386 Accuracy for GB: 0.86466	Accuracy for LR: 0.89587 Accuracy for RF: 0.89243 Accuracy for MNB: 0.87307 Accuracy GB: 0.86534
Lemmatization	Accuracy for LR: 0.88746 Accuracy for RF: 0.88265 Accuracy for MNB: 0.86166 Accuracy GB: 0.85704	Accuracy for LR: 0.88806 Accuracy for RF: 0.88459 Accuracy for MNB: 0.8604 Accuracy GB: 0.85821

Where LR=Logistic Regression, RF=Random Forest, MNB=Multinomial Naïve Bayes, and GB=Gradient Boosting.

I will explain the details of the models and scores in the next sections, in particular, in this case I have used the default parameters of the models, 100000 rows of data, and *max_features*=250 in the *CountVectorizer* and *Tfidf* methods to make it work faster.

With these results, I can know that the best combination to apply for Logistic Regression, Random Forest and Gradient Boosting is Tfidf to the stemmed data because is the one that has the highest accuracy value. Multinomial Naïve Bayes seems to perform better with Count vectorized stemmed data. So these will be the combinations that I use.

Another thing to say is that I also tried adding to the set of stopwords the next words: hotel, room, staff, rooms, breakfast, and bathroom, because they were the most common words in the first sample word cloud that I generated, to see if the model performs better, this is the result:

Vectorizer	*Count vectorizer*	*Tfidf*
Word transformation		
Stemming	Accuracy for LR: 0.89297 Accuracy for RF: 0.88685 Accuracy for MNB: 0.86917 Accuracy GB: 0.86342	Accuracy for LR: 0.89351 Accuracy for RF: 0.88873 Accuracy for MNB: 0.86872 Accuracy GB: 0.8646
Lemmatization	Accuracy for LR: 0.8852 Accuracy for RF: 0.87932 Accuracy for MNB: 0.85794 Accuracy GB: 0.8554	Accuracy for LR: 0.88596 Accuracy for RF: 0.88233 Accuracy for MNB: 0.85696 Accuracy GB: 0.85776

But, as you can see, the result was the opposite, it decreased the accuracy of all models. So this idea was discarded.

3.3. Model building

Now the data is completely processed and prepared, so the models will be able to work with it.

I will apply Machine Learning models for this forecasting task. A machine learning model is a file that has been trained to recognize certain types of patterns. I will train a model over a set of data, providing it an algorithm that it can use to reason over and learn from those data.

Once I have trained the model, I can use it to reason over data that it has not seen before, and make predictions about those data.

Particularly, in this project, I will use supervised learning, this is a machine learning task that establishes the mathematical relationship between input X and output Y variables. Such X, Y pair constitutes the labeled data that are used for model building to learn how to predict the output from the input. Our X variables will be the vectorized text, and Y will be if the review is positive or negative.

Because of this, we can consider this problem as a classification one. In machine learning, classification refers to a predictive modeling problem where a class label is predicted for a given example of input data, in our case it is a binary classification since the algorithm has to decide between two options, positive or negative.

First of all, I will explain the theoretical background behind the four chosen models and why I decided to choose them.

3.3.1. Logistic Regression

The first chosen model is Logistic Regression. All the models here shown have been taken from the *sklearn* library.

What is Logistic regression?

Logistic regression, despite its name, is a linear model for classification rather than regression. Logistic regression is also known in the literature as logit regression, maximum-entropy classification (MaxEnt), or the log-linear classifier. In this model, the probabilities describing the possible outcomes of a single trial are modeled using a logistic function.

Logistic regression is implemented in *LogisticRegression*. This implementation can fit binary, One-vs-Rest, or multinomial logistic regression with optional l_1, l_2, or Elastic-Net regularization. In this case, we will implement binary logistic regression, since we have to predict if the review is positive (1) or negative (2).

What parameters it has?

During this project, I have tried tweaking the next parameters:

- ***max_iter***: int, default=100
  Maximum number of iterations taken for the solvers to converge.

- ***penalty***: Penalized logistic regression imposes a penalty to the logistic model for having too many variables. This results in shrinking the coefficients of the less contributive variables toward zero. This is also known as regularization.
  Possible penalties: {'l1', 'l2', 'elasticnet', 'none'}, default='l2'
  Specify the norm of the penalty:
    - 'none': no penalty is added;
    - 'l2': add a L2 penalty term and it is the default choice;
    - 'l1': add a L1 penalty term;
    - 'elasticnet': both L1 and L2 penalty terms are added.

  I just tried with *l2* and *none* penalty, since the other norms produced errors.

Other parameters and their default values are: dual=False, tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None, random_state=None, solver='lbfgs', multi_class='auto', verbose=0, warm_start=False, n_jobs=None, l1_ratio=None

Why did I choose it?

Because it is incredibly fast, compared with the rest of the models, so it has served me as the tryout model, and even with that it has very high accuracy. Also is easy to implement and understand.

3.3.2. Random Forest classifier

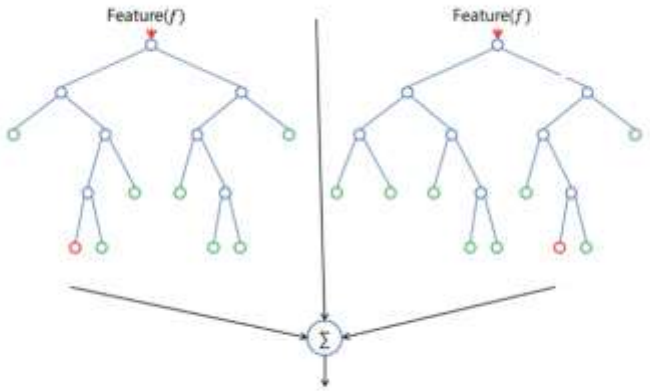
The second model is the Random forest classifier.

What is Random forest?

Random forest is a supervised learning algorithm. The "forest" it builds, is an ensemble of decision trees, usually trained with the “bagging” method. The general idea of the bagging method is that a combination of learning models increases the overall result.

Put simply: random forest builds multiple decision trees and merges them to get a more accurate and stable prediction.

Below you can see how a random forest would look like with two trees:



What parameters it has?

I have tweaked the following parameters:

- ***n_estimators***: int, default=100. The number of trees in the forest.
- ***max_depth***: int, default=None
  The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than *min_samples_split* samples.

Other parameters and their default values are: criterion='gini', min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0, bootstrap=True, oob_score=False, n_jobs=None, random_state=None, verbose=0, warm_start=False, class_weight=None, ccp_alpha=0.0 and max_samples=None.

Why did I choose it?

Since I have a huge amount of features (1000 for the final models), I thought that a Random forest classifier would be a great option, because Random forest adds additional randomness to the model, while growing the trees. Instead of searching for the most important feature while splitting a node, it searches for the best feature among a random subset of features. This results in a wide diversity that generally results in a better model. So following its theory instead of using all the features, the model will select those who perform better.

3.3.3. Gradient Boosting classifier

The third model to try is the Gradient boosting classifier.

What is Gradient boosting?

The Gradient Boosting Classifier depends on a loss function. A custom loss function can be used, and many standardized loss functions are supported by gradient boosting classifiers, but the loss function has to be differentiable.

Classification algorithms frequently use logarithmic loss, while regression algorithms can use squared errors. Gradient boosting systems don't have to derive a new loss function every time the boosting algorithm is added, rather any differentiable loss function can be applied to the system.

Gradient boosting systems have two other necessary parts: a weak learner and an additive component. Gradient boosting systems use decision trees as their weak learners. Regression trees are used for the weak learners, and these regression trees output real values. Because the outputs are real values, as new learners are added to the model the output of the regression trees can be added together to correct for errors in the predictions.

The additive component of a gradient boosting model comes from the fact that trees are added to the model over time, and when this occurs the existing trees aren't manipulated, their values remain fixed.

A procedure similar to gradient descent is used to minimize the error between given parameters. This is done by taking the calculated loss and performing gradient descent

to reduce that loss. Afterwards, the parameters of the tree are modified to reduce the residual loss.

The new tree's output is then appended to the output of the previous trees used in the model. This process is repeated until a previously specified number of trees is reached, or the loss is reduced below a certain threshold.

What parameters it has?

- ***learning_rate***: float, default=0.1
  Learning rate shrinks the contribution of each tree by learning_rate. There is a trade-off between learning_rate and n_estimators.
- ***n_estimators***: int, default=100
  The number of boosting stages to perform. Gradient boosting is fairly robust to over-fitting so a large number usually results in better performance.

Other parameters and their default values are: loss='deviance', subsample=1.0, criterion='friedman_mse', min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_depth=3, min_impurity_decrease=0.0, init=None, random_state=None, max_features=None, verbose=0, max_leaf_nodes=None, warm_start=False, validation_fraction=0.1, n_iter_no_change=None, tol=0.0001, ccp_alpha=0.0.

Why did I choose it?

Comparing it with Random forest I come with the next differences:

Like random forests, gradient boosting is a set of decision trees. The two main differences are:

- How trees are built: random forests builds each tree independently while gradient boosting builds one tree at a time.
- Combining results: random forests combine results at the end of the process (by averaging or "majority rules") while gradient boosting combines results along the way.

If you carefully tune parameters, gradient boosting can result in better performance than random forests. However, gradient boosting may not be a good choice if you have a lot of noise, as it can result in overfitting. They also tend to be harder to tune than random forests.

With this in mind, I wanted to see which one performs better.

3.3.4. Multinomial Naïve Bayes classifier

The last model that I have tried is the Multinomial Naïve Bayes Classifier.

What is Multinomial Naïve Bayes classifier?

Multinomial Naive Bayes algorithm is a probabilistic learning method that is mostly used in Natural Language Processing (NLP). The algorithm is based on the Bayes theorem and predicts the tag of a text such as a piece of email or newspaper article. It calculates the probability of each tag for a given sample and then gives the tag with the highest probability as output.

Naive Bayes classifier is a collection of many algorithms where all the algorithms share one common principle, and that is each feature being classified is not related to any other feature. The presence or absence of a feature does not affect the presence or absence of the other feature.

What parameters it has?

- ***alpha***:float, default=1.0
  Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).
- ***fit_prior***:bool, default=True
  Whether to learn class prior probabilities or not. If false, a uniform prior will be used.
- ***class_prior***:array-like of shape (n_classes,), default=None
  Prior probabilities of the classes. If specified the priors are not adjusted according to the data.

Why did I choose it?

The way the different types of Naive Bayesian classifiers have been designed makes them work very well on all kinds of text-related problems. Document classification is one such example of a text classification problem that can be solved by using both Multinomial and Bernoulli Naive Bayes. The calculation of probabilities is the major reason for this algorithm to be a text classification-friendly algorithm and a top favorite among the masses. This classifier is highly used for predictions in real-time and also used in recommendation systems along with collaborative filtering.

It has three main advantages:

- It is easy and fast to predict the class of the test data set. It also performs well in multi-class prediction.
- When the assumption of independence holds, a Naive Bayes classifier performs better compared to other models like logistic regression and you need less training data.
- It performs well in the case of categorical input variables compared to numerical variable(s).

3.3.5 Data splitting: K-Folds Cross-Validation

Once we can understand the theory behind the models is time to train them so we can test them after. As a common practice in machine learning, I will have to split the data into Train and Test sets.

In statistics and machine learning it is usual to split the data into two subsets: training data and testing data, and fit the model on the train data, to make predictions on the test data. When we do that, one of two things might happen: we overfit our model or we underfit our model. I don't want any of these things to happen, because they affect the predictability of our model — I might be using a model that has lower accuracy and/or is ungeneralized (meaning you can't generalize your predictions on other data). Let's see what under and overfitting actually mean:

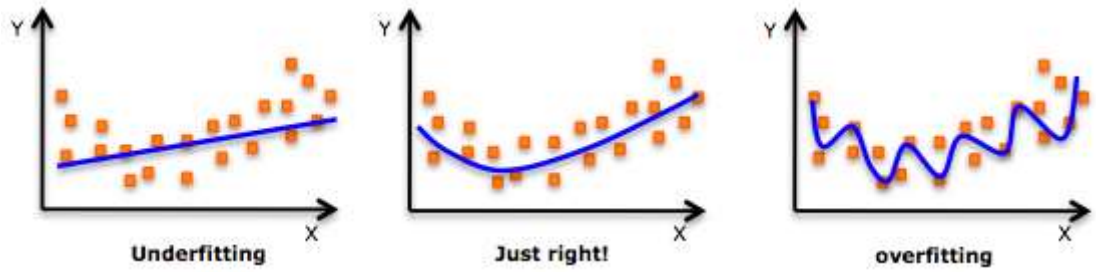
Overfitting

Overfitting means that the model I trained has trained “too well” and is now, well, fit too closely to the training dataset. This usually happens when the model is too complex (i.e. too many features/variables compared to the number of observations, in this case, I have a big number of features, so I must be careful). This model will be very accurate on the training data but will probably be very not accurate on untrained or new data. It is because this model is not generalized, meaning you can generalize the results and can’t make any inferences on other data, which is, ultimately, what you are trying to do. Basically, when this happens, the model learns or describes the “noise” in the training data instead of the actual relationships between variables in the data. This noise, obviously, is not part of any new dataset, and cannot be applied to it.

Underfitting

In contrast to overfitting, when a model is underfitted, it means that the model does not fit the training data and therefore misses the trends in the data. It also means the model cannot be generalized to new data. This is usually the result of a very simple model (not enough predictors/independent variables). It could also happen when, for example, we fit a linear model (like linear regression) to data that is not linear. It almost goes without saying that this model will have poor predictive ability (on training data and can’t be generalized to other data).

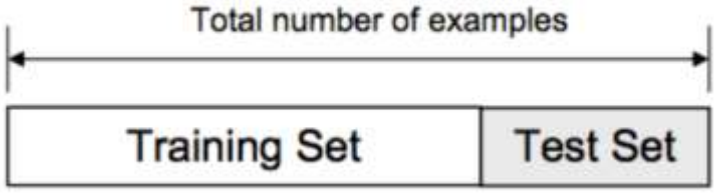
A visual example:



I want to avoid both these problems so train/test split and cross-validation will help avoiding overfitting more than underfitting.

Train/Test Split

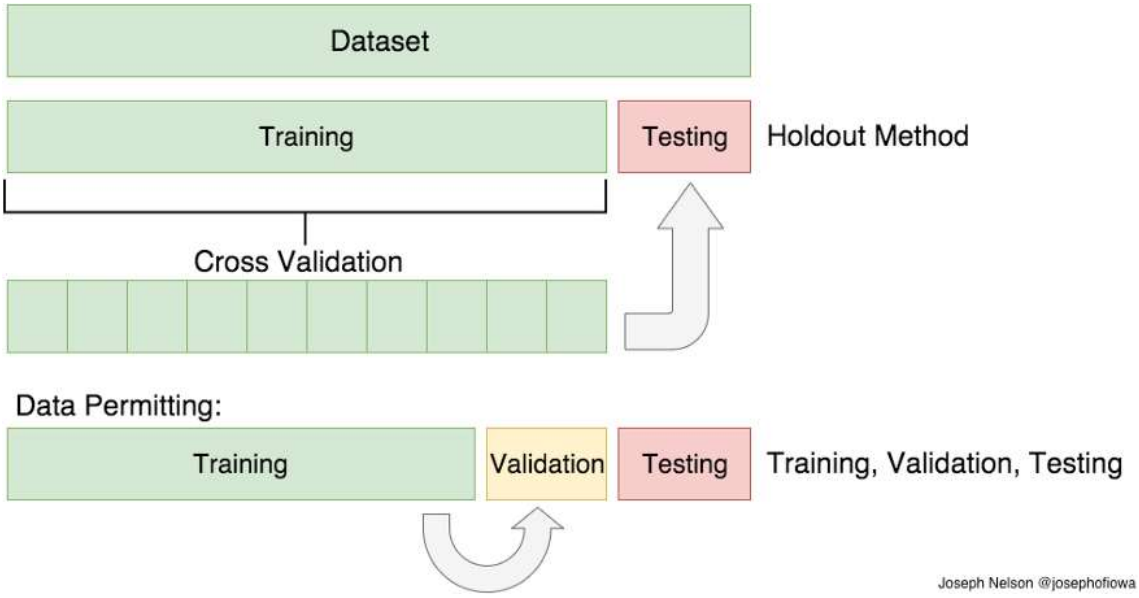
As I said before, the data we use is usually split into training data and test data. The training set contains a known output and the model learns on this data in order to be generalized to other data later on. We have the test dataset (or subset) to test our model’s prediction on this subset.



But train/test split does have its dangers — what if the split we make isn't random? What if one subset of our data has an overwhelming majority of positive or negative reviews (it is very unlikely but this type of bias can happen)? This will result in overfitting. So now is where cross-validation comes in.

Cross-Validation

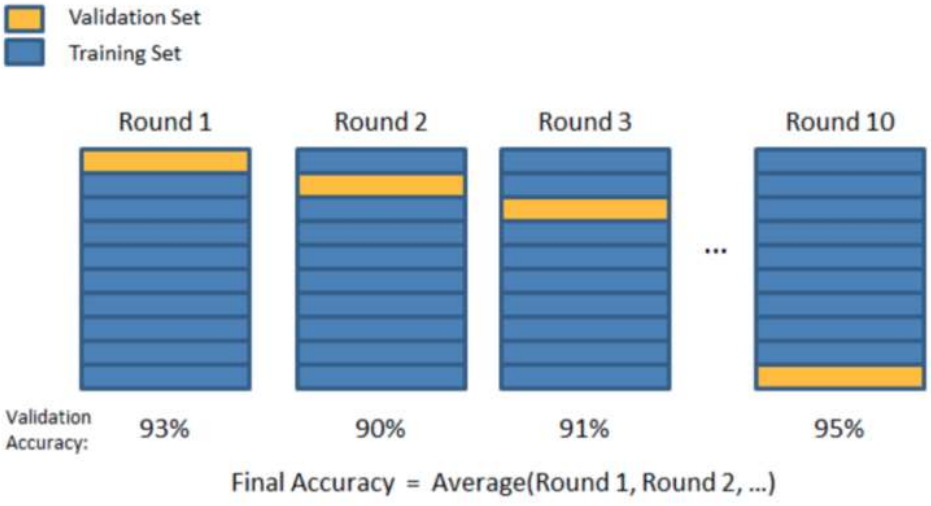
It's very similar to train/test split, but it's applied to more subsets. Meaning, we split our data into k subsets, and train on k-1 one of those subsets. What we do is to hold the last subset for test. We're able to do it for each of the subsets.



There are a bunch of cross-validation methods, I will use K-Folds Cross-Validation, as it is the most common among Data Scientists.

K-Folds Cross-Validation

In K-Folds Cross-Validation we split our data into k different subsets (or folds). We use k-1 subsets to train our data and leave the last subset (or the last fold) as test data. We then average the model against each of the folds and then finalize our model. After that, we test it against the test set.



To implement it I will import from *sklearn.model_selection* the *cross_val_predict* that will generate cross-validated estimates for each input data point.

The data is split according to the *cv* parameter. Each sample belongs to exactly one test set, and its prediction is computed with an estimator fitted on the corresponding training set.

The code to implement it on Logistic Regression looks as follows:

```
y_pred_logres = cross_val_predict(LogisticRegression(max_iter=100, penalty='none', random_state=0), X, y, cv=5)
```

Where I specify the model, the input data (X), the dependent variable (y), and the number of kFolds to use, I will use 5 kFolds as it is common in the use of this technique, I do not use more because it increases significantly the time to train the models.

The next advanced technique that I used to train the models is **GridSearchCV**.

3.3.5. Hyperparameter tunning: GridSearchCV

In Machine Learning models we use parameters to specify how the models should train with the data. The most simple way to train a model will be to use the default hyperparameters, but of course, with this, we are letting behind a lot of margin to improve our model, since we do not know which hyperparameters will make the model perform better.

First, what is grid search? It is the process of performing hyperparameter tuning to determine the optimal values for a given model. As mentioned above, the performance of a model significantly depends on the value of hyperparameters. It is impossible to try all the possible values of the hyperparameters, but it will help me obtain a better performance since I am trying with different combinations of them.

GridSearchCV is a function that comes in *sklearn model_selection* package.

First, I pass predefined values for hyperparameters to the *GridSearchCV* function. I do this by defining a dictionary in which we mention a particular hyperparameter along with the values it can take.

GridSearchCV tries all the combinations of the values passed in the dictionary and evaluates the model for each combination using the Cross-Validation method. Hence after using this function I get accuracy/loss for every combination of hyperparameters and I can choose the one with the best performance.

Now, let's see the different combinations of the hyperparameters that I chose for the models and how I apply the method. In most of them, I use two hyperparameters with around three different values per each, this is because of time and computational reasons since this technique takes a lot of time to work.

The chosen hyperparameters for each model are the ones that I considered to be the most important by their definition, the values selected are generally chosen around the default value of each hyperparameter.

Logistic Regression:

```
parameters_lr = {'max_iter': [100, 150, 200], 'penalty': ['l2', 'none']}
lr_model = LogisticRegression(random_state = 0)
grid_lr = GridSearchCV(estimator=lr_model, param_grid=parameters_lr, cv=3)
grid_lr.fit(X_train, y_train)
print("Best parameters for Logistic Regression: ",grid_lr.best_params_)
```

For Logistic Regression I tried different combinations of *max_iter* and *penalty*.

In the beginning, I tried with all the possible values of penalty, but I got errors using the *l1* and *elasticnet* penalties. The default value of *max_iter* is 100, I tried with lower values but they resulted in convergence errors, so I decided to try with 150 and 200 as they are higher values.

The best values were: *max_iter*=100, *penalty*='none', so these will be the ones that I will use in the definitive model:

```
start = timeit.timeit()
y_pred_logres = cross_val_predict(LogisticRegression(max_iter=100, penalty='none', random_state=0), X, y, cv=5)
end = timeit.timeit()
print(end - start)
#Print accuracy score and confusion matrix on test set
print('Accuracy for Logistic Regression: ', accuracy_score(y, y_pred_logres))
print(confusion_matrix(y, y_pred_logres)/len(y))
print('Recall for Logistic Regression: ',recall_score(y, y_pred_logres))
print('Precision for Logistic Regression: ',precision_score(y, y_pred_logres))
print('ROC-AUC on the test set: ',roc_auc_score(y, y_pred_logres))
```

Random Forest:

```
parameters_rf = {'n_estimators': [50, 100, 150],'max_depth': [None,1,2]}
rf_model = RandomForestClassifier(random_state = 0)
grid_rf = GridSearchCV(estimator=rf_model, param_grid=parameters_rf, cv=3)
grid_rf.fit(X_train, y_train)
print("Best parameters for Random Forest: ",grid_rf.best_params_)
```

For Random Forest I tried with *n_estimators* and *max_depth*. These two are the values that define the general structure of the forest (the number of trees, and the depth of each one respectively), that is why I chose them.

Being 100 the default number of trees and None the default depth of the forest (then nodes are expanded until all leaves are pure or until all leaves contain less than 2 samples).

The best values were: *max_depth*=None, *n_estimators*=150

```
start = timeit.timeit()
y_pred_randomfor = cross_val_predict(RandomForestClassifier(max_depth=None, n_estimators=150, random_state = 0), X, y, cv=5)
end = timeit.timeit()
print(end - start)
#Print accuracy score and confusion matrix on test set
print('Accuracy for Random Forest Classifier: ', accuracy_score(y, y_pred_randomfor))
print(confusion_matrix(y, y_pred_randomfor)/len(y))
print('Recall for Random Forest: ',recall_score(y, y_pred_randomfor))
print('Precision for Random Forest: ',precision_score(y, y_pred_randomfor))
print('ROC-AUC on the test set: ',roc_auc_score(y, y_pred_randomfor))
```

Multinomial Naïve Bayes:

```
parameters_mnb = {'fit_prior': [True, False],'alpha': [0, 0.1, 1]}
mnb_model = MultinomialNB()
grid_mnb = GridSearchCV(estimator=mnb_model, param_grid=parameters_mnb, cv=3)
grid_mnb.fit(X_train, y_train)
print("Best parameters for Multinomial Naive Bayes: ",grid_mnb.best_params_)
```

For this model, we just have 3 different hyperparameters *alpha*, *fit_prior* and *class_prior*.

I tried combinations of *fit_prior* and the smoothing parameter *alpha*. I wanted to see whether to learn class prior probabilities or not performs better. Also, if it is better with a bigger smoothing parameter or without it.

The best values were: *alpha*=0, *fit_prior*=False

```
start = timeit.timeit()
y_pred_mnb = cross_val_predict(MultinomialNB(alpha=0, fit_prior=False), X_mnb, y, cv=5)
end = timeit.timeit()
#Print accuracy score and confusion matrix on test set
print('Accuracy on the test set for Multinomial Naive Bayes: ', accuracy_score(y, y_pred_mnb))
print(confusion_matrix(y, y_pred_mnb)/len(y))
print('Recall for Multinomial Naive Bayes Model: ',recall_score(y, y_pred_mnb))
print('Precision for Multinomial Naive Bayes Model: ',precision_score(y, y_pred_mnb))
print('ROC-AUC on the test set: ',roc_auc_score(y, y_pred_mnb))
```

Gradient Boosting:

```
parameters_gb = {'n_estimators': [50, 100, 200],'learning_rate': [0.001, 0.01, 0.1],}
gb_model = GradientBoostingClassifier(random_state = 0)
grid_gb = GridSearchCV(estimator=gb_model, param_grid=parameters_gb, cv=3)
grid_gb.fit(X_train, y_train)
print("Best parameters for GB: ",grid_gb.best_params_)
```

Talking about Gradient Boosting, we have a lot of hyperparameters to choose from.Between all of them, I selected the number of boosting stages to perform and the learning rate of the model.

I tried with smaller learning rates than the default value 0.1, and with a number of estimators around 100.

The best values were: *learning_rate=0.1, n_estimators=200*

```
start = timeit.timeit()
y_pred_gb = cross_val_predict(GradientBoostingClassifier(learning_rate=0.1, n_estimators=200, random_state=0), X_gb, y, cv=5)
end = timeit.timeit()
print(end - start)
#Print accuracy score and confusion matrix on test set
print('Accuracy on the test set for Gradient Boosting Classifier: ', accuracy_score(y, y_pred_gb))
print(confusion_matrix(y, y_pred_gb)/len(y))
print('Recall for Gradient Boosting Classifier: ',recall_score(y, y_pred_gb))
print('Precision for Gradient Boosting Classifier: ',precision_score(y, y_pred_gb))
print('ROC-AUC on the test set: ',roc_auc_score(y, y_pred_gb))
```

Now that I have the best parameters to train my models, it is time to train them and see the scores that they can provide us in the results section.

4. Results

This section stands as one of the most important of the report. The results in a Big Data project like this show how the solution performs. In particular, I will be showing different scores for a classification problem like this one.

Before, let's see the theory behind the scores here presented:

4.1. Theory behind the scores

4.1.1. Accuracy

To obtain the accuracy score I have used *sklearn.metrics.accuracy_score* that computes the accuracy classification score. In our case, it will return the percentage of well-predicted reviews over the total of them.

We can not take just into account the accuracy to know if a model is performing well or not. Imagine that there's an 80% of positive reviews and our model always predicts that the review is positive, even if the accuracy will be 0.8 the model is overfitting and not performing well.

4.1.2. Confusion Matrix

Predicted class / Actual class	P	N
P	TP	FN
N	FP	TN

I have used *sklearn.metrics.confusion_matrix* to compute it.

By definition, a confusion matrix C is such that $C_{i,j}$ is equal to the number of observations known to be in group i and predicted to be in group j.

Thus in binary classification, the count of true negatives is $C_{0,0}$, false negatives is $C_{1,0}$, true positives is $C_{1,1}$ and false positives is $C_{0,1}$.

This score provides us a better observation of the results than the accuracy because it says us how the model is performing on both, positive and negative reviews.

4.1.3. Recall

$$Sensitivity = \frac{TP}{TP + FN}$$

With *sklearn.metrics.recall_score*.

The recall is the ratio tp / (tp + fn) where tp is the number of true positives and fn the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples. Also known as True Positive Rate (TPR) or Sensitivity.

The best value is 1 and the worst value is 0.

4.1.4. Precision

I used *sklearn.metrics.precision_score* to compute it.

The precision is the ratio tp / (tp + fp) where tp is the number of true positives and fp the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

The best value is 1 and the worst value is 0.

4.1.5. ROC AUC Score

From *sklearn.metrics.roc_auc_score.*

Computes the Area Under the Receiver Operating Characteristic Curve (ROC AUC) from prediction scores. The Receiver Operator Characteristic (ROC) curve is an evaluation metric for binary classification problems. It is a probability curve that plots the TPR against FPR at various threshold values and essentially separates the ‘signal’ from the ‘noise’. The Area Under the Curve (AUC) is the measure of the ability of a classifier to distinguish between classes and is used as a summary of the ROC curve.

The False Negative Rate (FNR) tells us what proportion of the positive class got incorrectly classified by the classifier, and the TPR will be the same as the Recall.

Once we understand the scores from the inside we can have a much better overview of the results obtained. So let's see them.

4.2. Comment on the results

In the table below you can see the different scores obtained for the different models applied:

	Accuracy	*Confusion Matrix [TN FN] [FP TP]*	*Recall*	*Precision*	*ROC-AUC*
Linear Regression	0.9181	[[0.3981 0.0371] [0.0447 0.5198]]	0.9207	0.9333	0.9177
Random Forest	0.9183	[[0.3961 0.0391] [0.0424 0.5221]]	0.9247	0.9302	0.9173
Multinomial Naïve Bayes	0.9035	[[0.3910 0.0443] [0.0521 0.5125]]	0.9076	0.9204	0.9029
Gradient Boosting	0.8850	[[0.3912 0.0440] [0.0708 0.4937]]	0.8744	0.9180	0.8840

We can see the best results in green color for each score.

Talking about them, I can see that the models are performing pretty well, even if I had to use a number of features equal to 1000 to train the models for computational reasons, note that Gradient boosting was trained with 500 features because it takes too long to train it with 1000, this is why it has worse scores.

The accuracy for the first three models is around 91%, with an 88.5% for Gradient Boosting, I can say that is a good percentage taking into account the problem that we are trying to predict.

In general, talking about the confusion matrix, recall, and precision I can say that the models are predicting better the negative reviews than the positives. I can see this in the confusion matrix, where the percentage of False Negatives is always lower than the percentage of False Positives. This difference is even bigger in the Gradient Boosting model, maybe because of the lack of features.

We could say that it should be the opposite since there are more positive reviews (56.4661%) than negative (43.5339%) so the model could overfit the training set and tend to predict more positives than negatives, but in fact, it is a good sign that this is not happening, so I can say that my models are not overfitting the data.

Talking about the ROC-AUC score, when 0.88<AUC<0.9177 as is my case, there is a really high chance that the classifier will be able to distinguish the positive class values from the negative class values. This is so because the classifier is able to detect more numbers of True positives and True negatives than False negatives and False positives.

Now, in particular, about the models, the best performances are Logistic Regression and Random Forest, being the first one better predicting negative reviews and the second one better with the positive ones, this is very interesting.

Also, I wanted to compare Random Forest and Gradient Boosting, but since Gradient Boosting could not be trained with 1000 features, it is not fair to compare them.

This being said I can say that I am completely satisfied with the results obtained, and I can conclude that the models are doing great forecasting labor.

5. Conclusion and Recommendations

We have come to the final of this report on sentiment analysis in hotel reviews. We have been seen all the steps involved in a Big Data project like this. From the data discovery, where we saw how I obtained the data, from Kaggle, web scrapping, and self-made reviews. Passing through the data preparation, where I created a database with its stored procedures, cleaned the data, applied stemming, lemmatization, count vectorizing, tfidf transformation, and saw which ones performed better in each model. To finally show the results obtained, were I came to the conclusion that the forecasting task of this project has been a success.

Now, and to finish this report, I will make some recommendations that will help future versions of this project improve the forecasting performance of the models, most of these recommendations have not been applied because of lack of time to try them or because of computational requirements, like the max number of features.

- It can be better for the models to have a more equal number of positive and negative reviews, so we prevent overfitting, although I have not seen any sign of it.

- In the data preparation part, it will be much better if I do not compare the models just with the accuracy score, other metrics such as the confusion matrix or ROC-AUC should be also compared to make a better decision about using lemmatizing, stemming, count vectorizing and tfidf. Also, it will be better if they are compared with the whole data.

- It will be interesting to see the performance of the models just vectorizing with unigrams or just bigrams, to see if it decreases, although using both (as I did) should be the best option.

- The ideal way of count vectorizing or applying tfidf would be without a restriction in the number of features, in this way the performance of the models is clearly higher.

- More hyperparameters and a wider range of their values could show us better hyperparameters options for the models when applying GridSearchCV.

- Other classifiers like Support Vector Machine, K-nearest Neighbour, and specially XGBoosting, due to its fame as a good classifier, could be interesting options to try in the future.

6. References

1.1. Linear Models. (n.d.). Scikit-Learn. https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression

515K Hotel Reviews Data in Europe. (n.d.). Kaggle. https://www.kaggle.com/jiashenliu/515k-hotel-reviews-data-in-europe

A. (2021a, September 15). *Substitutions in Regular Expressions*. Microsoft Docs. https://docs.microsoft.com/en-us/dotnet/standard/base-types/substitutions-in-regular-expressions

Bhandari, A. (2020, July 20). *AUC-ROC Curve in Machine Learning Clearly Explained*. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2020/06/auc-roc-curve-machine-learning/

Boost Labs. (2020, November 3). *What are Word Clouds? The Value of Simple Visualizations*. https://boostlabs.com/blog/what-are-word-clouds-value-simple-visualizations/

Brownlee, J. (2020, August 19). *4 Types of Classification Tasks in Machine Learning*. Machine Learning Mastery. https://machinelearningmastery.com/types-of-classification-in-machine-learning/

Bronshtein, A. (2020, March 24). *Train/Test Split and Cross Validation in Python - Towards Data Science*. Medium. https://towardsdatascience.com/train-test-split-and-cross-validation-in-python-80b61beca4b6

Decision Tree vs Random Forest vs Gradient Boosting Machines: Explained Simply. (n.d.). Data Science Central. https://www.datasciencecentral.com/profiles/blogs/decision-tree-vs-random-forest-vs-boosted-trees-explained

Donges, N. (2021, September 17). *A Complete Guide to the Random Forest Algorithm*. Built In. https://builtin.com/data-science/random-forest-algorithm

GeeksforGeeks. (2021, September 15). *Python - Lemmatization Approaches with Examples*. https://www.geeksforgeeks.org/python-lemmatization-approaches-with-examples/

Gupta, S. (2018, June 17). *Sentiment Analysis: Concept, Analysis and Applications*. Medium. https://towardsdatascience.com/sentiment-analysis-concept-analysis-and-applications-6c94d6f58c17

Mujtaba, H. (2021, June 23). *Hyperparameter Tuning with GridSearchCV*. GreatLearning Blog: Free Resources What Matters to Shape Your Career! https://www.mygreatlearning.com/blog/gridsearchcv/

Nantasenamat, C. (2021, June 4). *How to Build a Machine Learning Model - Towards Data Science*. Medium. https://towardsdatascience.com/how-to-build-a-machine-learning-model-439ab8fb3fb1

Real Python. (2021, May 8). *Combining Data in Pandas With merge(), .join(), and concat()*. Combine Data. https://realpython.com/pandas-merge-join-and-concat/#pandas-concat-combining-data-across-rows-or-columns

S. (2021b, June 15). *12 Twitter Sentiment Analysis Algorithms Compared*. AI Perspectives. https://www.aiperspectives.com/twitter-sentiment-analysis/

S. (2021c, October 19). *Multinomial Naive Bayes Explained: Function, Advantages & Disadvantages, Applications in 2021*. UpGrad Blog. https://www.upgrad.com/blog/multinomial-naive-bayes-explained/

Shah, E. (2021, January 11). *Naive Bayes – Why Is It Favoured For Text Related Tasks?* Analytics India Magazine. https://analyticsindiamag.com/naive-bayes-why-is-it-favoured-for-text-related-tasks/

sklearn.feature_extraction.text.CountVectorizer. (n.d.). Scikit-Learn. https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html

sklearn.feature_extraction.text.TfidfTransformer. (n.d.). Scikit-Learn. https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfTransformer.html#sklearn.feature_extraction.text.TfidfTransformer

sklearn.linear_model.LogisticRegression. (n.d.). Scikit-Learn. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

sklearn.metrics.accuracy_score. (n.d.). Scikit-Learn. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html

sklearn.metrics.confusion_matrix. (n.d.). Scikit-Learn. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html

sklearn.metrics.precision_score. (n.d.). Scikit-Learn. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html

sklearn.metrics.recall_score. (n.d.). Scikit-Learn. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.recall_score.html

sklearn.metrics.roc_auc_score. (n.d.). Scikit-Learn. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_auc_score.html

sklearn.model_selection.GridSearchCV. (n.d.). Scikit-Learn. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

Stemming and Lemmatization in Python. (n.d.). DataCamp Community. https://www.datacamp.com/community/tutorials/stemming-lemmatization-python

What is Data Discovery? (n.d.). TIBCO Software. https://www.tibco.com/reference-center/what-is-data-discovery

What is Data Preparation? (+ How to Make It Easier) - Talend. (n.d.). Talend - A Leader in Data Integration & Data Integrity. https://www.talend.com/resources/what-is-data-preparation/