Sentiment Analysis & Neural Networks

An insight into hotel reviews and predictions

Nawaz A.Rahman

Thesis Report 30p

Simon Celinder

Data Scientist

Göteborg | 2 May 2023

Table of Contents

List of Abbreviations

NLTK: Natural Language Toolkit

EDA: Exploratory Data Analysis

N-grams: A sequence of N words in a sentence

LSTM: Long Short-Term Memory

VADER: Valence Aware Dictionary and sEntiment Reasoner

NLP: Natural Language Processing

TP: True Positive

FN: False Negative

FP: False Positive

TN: True Negative

Abstract

Sentiment analysis is a technique that has found greater use in a variety of industries, including hotel management. This project's goal is to conduct a sentiment analysis of hotel reviews using NLP tools like LSTM and NLTK. The project will investigate how effectively these tools function in order to categorize hotel reviews as either positively or negatively based on the information provided.

One of the major benefits of using sentiment analysis is that it allows for a quick and efficient analysis of large amounts of data, which may be quite helpful for hoteliers. This project will contribute to a better understanding of how the hotel experience may be improved by examining what factors influence the classification of reviews as positive or negative.
The outcome of the sentiment analysis can also be used to enhance the guest experience at hotels. The most common word combinations and phrases used in negative reviews can be analyzed to help hoteliers pinpoint the most pressing and pressing problem areas. On this basis, hoteliers can enhance their services and, as a result, improve customer satisfaction and community.

In conjunction with this project, LSTM and NLTK will be improved to increase their effectiveness for sentiment analysis of hotel reviews and, as a result, make it possible for hoteliers to identify and fix issues that impair guests' experiences quickly and easily.

Research Question

How effective are LSTM and NLTK as NLP tools for sentiment analysis of hotel reviews, and how can the insights gained from sentiment analysis be used to identify and address issues that impair guest experiences? Can the most common word combinations and phrases used in negative reviews be analyzed to pinpoint the most pressing problem areas, and can improving the effectiveness of LSTM and NLTK for sentiment analysis make it easier for hoteliers to conduct sentiment analysis on large amounts of data and quickly identify areas for improvement in their services?

Hypothesis

LSTM and NLTK are effective NLP tools for sentiment analysis of hotel reviews, and the insights gained from sentiment analysis can be used to identify and address issues that impair guest experiences. By analyzing the most common word combinations and phrases used in negative reviews, hoteliers can pinpoint the most pressing problem areas and improve their services, resulting in increased customer satisfaction and loyalty. Additionally, by improving the effectiveness of LSTM and NLTK for sentiment analysis, it will be easier for hoteliers to conduct sentiment analysis on large amounts of data and quickly identify areas for improvement in their services.

Boundaries

The project's primary goal is to analyze the sentiments of hotel reviews that have been collected from online platforms. Only text data will be used to train and test models with the aid of NLP tools like LSTM and NLTK. The project won't include implementing the results; rather, it will just provide guidance on how to use them to improve the guest experience. The project's timeframe is restricted to the class term, and work will be done on regular workdays.

Methods

1. Dataset Pre-processing

- Perform exploratory data analysis (EDA) to gain insights about the dataset
- Clean the dataset by removing irrelevant columns, duplicates, and missing values
- Create a new column for the sentiment labels (positive or negative) based on the star ratings and/or text reviews
- Extract the positive column and word count column for further analysis

2. Text Processing

- Use the Natural Language Toolkit (NLTK) library for text processing and cleaning

3. Text Analysis

- Check the review weights by counting reviews by rating
- Use word count to see how the review rating is affected

- Use word frequency analysis to identify frequently occurring words and n-grams (unigrams, bigrams, and trigrams)

4. Model Preparation

- Split the pre-processed dataset into training and testing sets
- Determine the number of words in the vocabulary to add in padding
    - Perform visual inspection using boxplot and displot
    - Implement the Interquartile Range (IQR) method to detect and remove outliers
- Tokenize the text data and create a word-to-index dictionary
- Perform padding to ensure that each review has the same length
- Perform one-hot encoding on the sentiment labels
- Reshape the data from 2D to 3D to prepare for LSTM modelling

5. Model Building

- Take steps to balance the dataset weights
- Save the model weights using the ModelCheckpoint function to prevent overfitting
- Build an LSTM model using Keras libraries
- Train the LSTM model with train data
- Test the LSTM model on the test data
- Evaluate the LSTM model's performance using accuracy, precision, recall, and F1-score metrics
- Generate a confusion matrix and summarize the results

6. Model Comparison

- Predict a new dataset on the LSTM model
- Use the VADER model as a baseline for comparison
- Use the VADER model to make predictions on the new dataset

- Compare the true positive values against the LSTM and VADER predictions
- Evaluate the LSTM and VADER models based on their scores and prediction accuracy
- Compare the LSTM and VADER models and identify their strengths and weaknesses

Overall, this method involves preprocessing the dataset, cleaning the text data, preparing the data for modelling, building the LSTM model, evaluating the model's performance, and comparing it to the VADER model. This approach can provide a comprehensive analysis of sentiment analysis using LSTM and VADER models.

Implementation

Dataset Pre-processing

This is what the dataset looks like loading it in to jupyter notebook. As stated in the boundaries this project will only be using “Review Text” and “Review Rating”.

```
# Let's have a look at our dataset
df.head()
```

	Property Name	Review Rating	Review Title	Review Text	Location Of The Reviewer	Date Of Review
0	Apex London Wall Hotel	5	Ottima qualità prezzo	Siamo stati a Londra per un week end ed abbiam...	Casale Monferrato, Italy	10/20/2012
1	Corinthia Hotel London	5	By far, my best hotel in the world	I had a pleasure of staying in this hotel for ...	Savannah, Georgia	3/23/2016
2	The Savoy	5	First visit to the American Bar at the Savoy	A very lovely first visit to this iconic hotel...	London	7/30/2013
3	Rhodes Hotel	4	Nice stay	3 of us stayed at the Rhodes Hotel for 4 night...	Maui, Hawaii	6/2/2012
4	The Savoy	5	Perfection	Form the moment we arrived until we left we ex...	London, United Kingdom	11/24/2017

Next, the step will be to remove everything except the two features mentioned above.

```
# We only need the review text and raiting
df = df[['Review Text', 'Review Rating']]
df
```

	Review Text	Review Rating
0	Siamo stati a Londra per un week end ed abbiam...	5
1	I had a pleasure of staying in this hotel for ...	5
2	A very lovely first visit to this iconic hotel...	5
3	3 of us stayed at the Rhodes Hotel for 4 night...	4
4	Form the moment we arrived until we left we ex...	5
...	...	...
27325	I come to London often but since I stayed in t...	5
27326	En cuarto que nos tocó no había toallas y habí...	3
27327	This is a quality quiet hotel located in an ex...	4
27328	Väldigt vackra rum, tyvärr med en mycket höglj...	4
27329	I have been staying in London hotels for 10 ye...	5

27330 rows × 2 columns

As seen in the picture to the right, the dataset contains 27,330 rows of data with two features.

```
# Checking for null values.
df.isna().any()
```

```
Review Text      False
Review Rating    False
dtype: bool
```

Checking for null values in the dataset because training model with null values can affect the performance. If the dataset contains null values, the model may learn to associate these null values with specific outcomes, leading to biased predictions.

A new column is created here for the sentiment labels (positive or negative) based on the star ratings and/or text reviews.

Rating 3 or below is considered negative and 4-5 is positive.

```
# If rating [1, 2, 3] = Negative and if rating [4, 5] = Positive
def ratings(rating):
    if rating>0 and rating<=3:
        return 0
    if rating>3 and rating<=5:
        return 1

df['Positive'] = df['Review Rating'].apply(ratings)
df.head()
```

	Review Text	Review Rating	Positive
0	Siamo stati a Londra per un week end ed abbiam...	5	1
1	I had a pleasure of staying in this hotel for ...	5	1
2	A very lovely first visit to this iconic hotel...	5	1
3	3 of us stayed at the Rhodes Hotel for 4 night...	4	1
4	Form the moment we arrived until we left we ex...	5	1

```
def word_count(review):
    review_list = review.split()
    return len(review_list)

df['Word_count'] = df['Review Text'].apply(word_count)
df.head()
```

	Review Text	Review Rating	Positive	Word_count
0	Siamo stati a Londra per un week end ed abbiam...	5	1	171
1	I had a pleasure of staying in this hotel for ...	5	1	265
2	A very lovely first visit to this iconic hotel...	5	1	54
3	3 of us stayed at the Rhodes Hotel for 4 night...	4	1	75
4	Form the moment we arrived until we left we ex...	5	1	62

The following function splits the reviews by word, and then creates a new column called "Word_count".

Text Processing

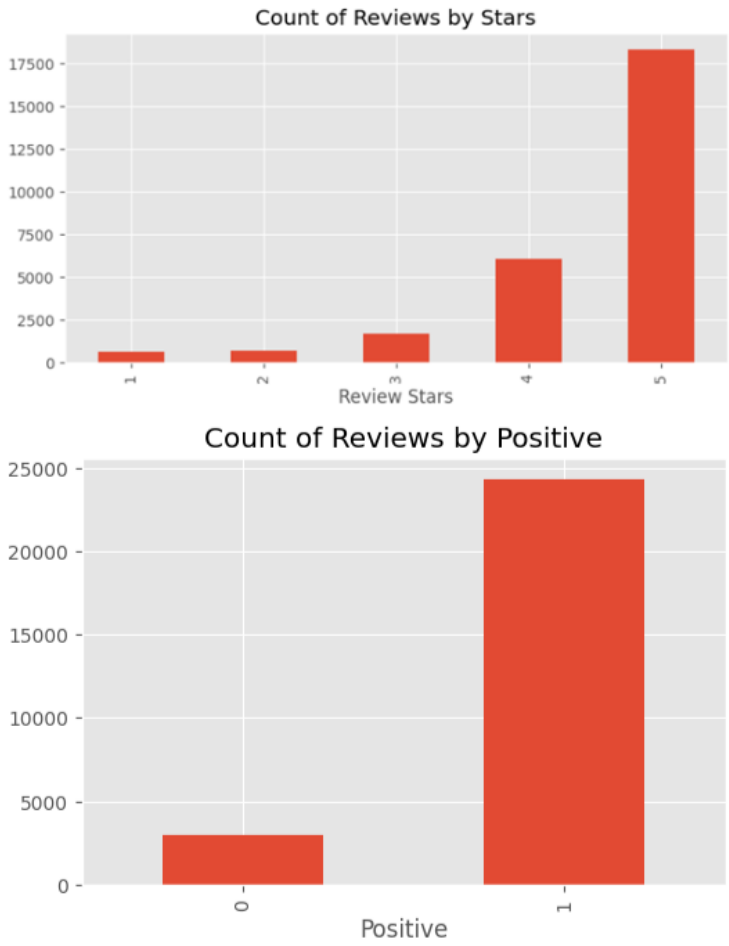
The text processing and cleaning step involves utilizing the Natural Language Toolkit (NLTK) library. The function is applied to the text column, which performs several actions, including removing punctuations and numbers, eliminating single characters that are not part of any words with more than one character, removing any instances of multiple spaces, and finally, removing all stop words (such as "a", "an", "and", "as", "at", "but", etc.) from the sentences.

```
def preprocess_text(sen):
    '''Cleans up text data, leaving 2 or more char long non-stopwords containing A-Z & a-z only in lowercase'''

    sentence = sen.lower()

    # Remove punctuations and numbers
    sentence = re.sub('[^a-zA-Z]', ' ', sentence)

    # Remove single characters that are not part of words with more than one character
    sentence = re.sub(r"\b[a-zA-Z]\b", '', sentence)

    # Remove multiple spaces
    sentence = re.sub(r'\s+', ' ', sentence)

    # Remove stopwords
    stop_words = set(stopwords.words('english'))
    words = sentence.split()
    filtered_words = [word for word in words if not word in stop_words]

    return ' '.join(filtered_words)
```

```
# Empty list where the cleaned text will be stored
clean_text = []

clean_sentences = list(df['Review Text'])
for sen in tqdm(clean_sentences):
    clean_text.append(preprocess_text(sen))
```

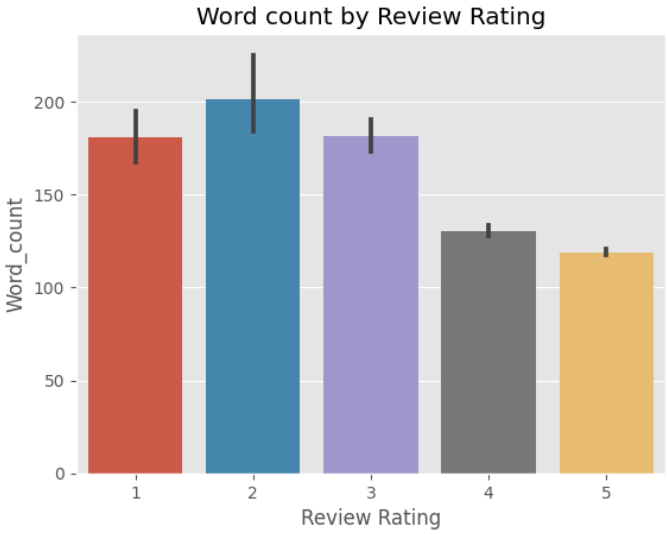
```
100%|██████████| 27330/27330 [00:07<00:00, 3567.32it/s]
```

Text Analysis



The two plots on the right show the review ratings based on the count of reviews made. As observed in the plots, the data is heavily skewed towards positive reviews. Therefore, when training the LSTM model, we should keep in mind that the weights are uneven and this may lead to biased accuracy towards the majority class, which in this case is positive reviews.

It's important to take steps to balance the dataset and adjust the training parameters to mitigate this issue which will be done in a later stage.



This plot shows average word count of the review rating. Based on the graph, it can be inferred that the number of words decreases as the star rating increases. This is not surprising, as people tend to have more to say when they are dissatisfied which often leads to more emotions being involved.

The next step in the analysis, which involves analyzing the most frequent n-grams in the reviews. A function called corpus() is created for this purpose, which first cleans the review text using the preprocess_text() function created earlier using the NLTK library. The cleaning process includes removing punctuations, numbers, single characters, multiple spaces, and stop words. The reason for this is to remove unnecessary elements that don't provide significant information about the sentiment and make it harder to find the words that matter. By removing these elements, it becomes easier to identify the most frequent n-grams in the reviews.

```
# A function that takes the review text and splits all the words
def corpus(text):
    test_list = text.split()
    return test_list
```

```
# Convert the clean_text list to a pandas Series object
clean_text_series = pd.Series(clean_text)

# Applies the corpus function on the review text and then saves into a new column
df['Corpus_Reviews'] = clean_text_series.apply(corpus)
df.head()
```

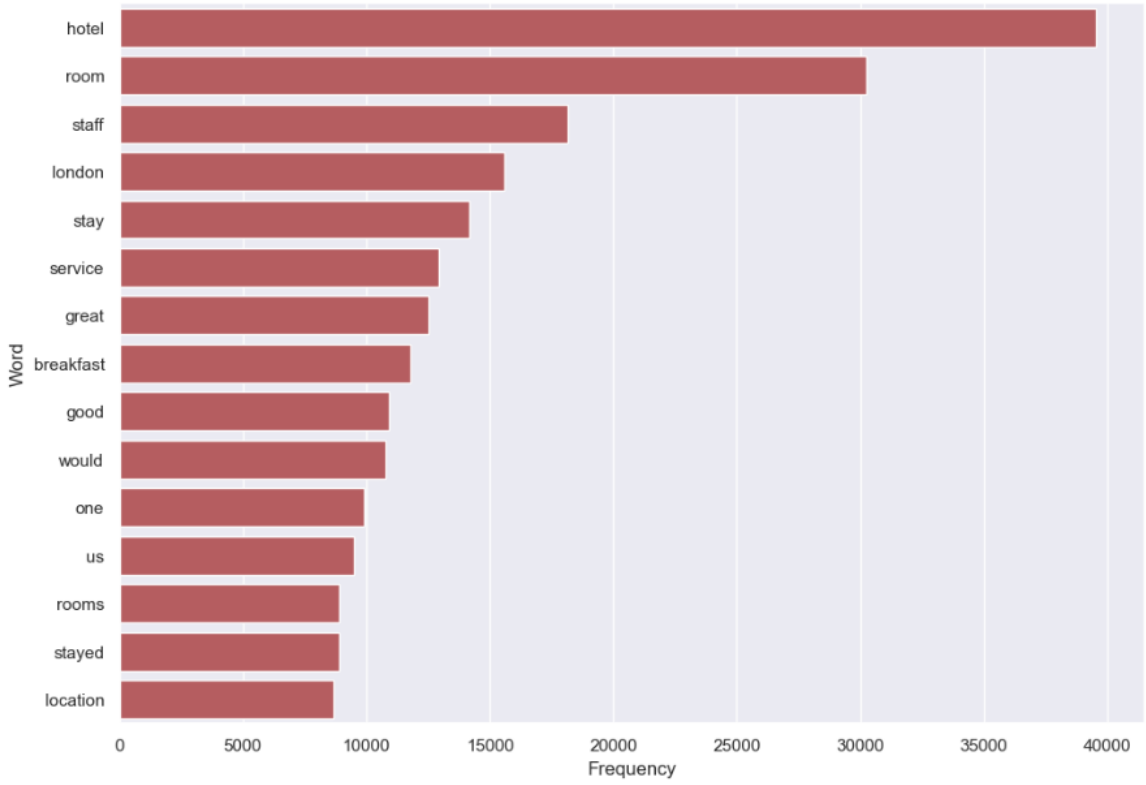
	Review Text	Review Rating	Positive	Word_count	Corpus_Reviews
0	Siamo stati a Londra per un week end ed abbiam...	5	1	171	[siamo, stati, londra, per, un, week, end, ed,...
1	I had a pleasure of staying in this hotel for ...	5	1	265	[pleasure, staying, hotel, nights, recently, h...
2	A very lovely first visit to this iconic hotel...	5	1	54	[lovely, first, visit, iconic, hotel, bar, won...
3	3 of us stayed at the Rhodes Hotel for 4 night...	4	1	75	[us, stayed, rhodes, hotel, nights, great, loc...
4	Form the moment we arrived until we left we ex...	5	1	62	[form, moment, arrived, left, experienced, abs...

After creating the corpus, an empty list called all_words is initialized. A for loop then iterates over each review in the corpus and adds all the words to the list. Next, the Counter() class is used to count the occurrence of each word in the list, resulting in a unigram frequency count.

```
# Create an empty list to store all words in the corpus
all_words = []

# Iterate over each review in the corpus and add all words to the all_words list
for review in df['Corpus_Reviews']:
    all_words.extend(review)

# Count the frequency of each word and get the 15 most common words
word_freq = Counter(all_words)
top_words = word_freq.most_common(15)

# Print the 15 most common words
print("The 15 most frequent words in the corpus are:")
for word, freq in top_words:
    print(f"{word}: {freq}")
```

```
The 15 most frequent words in the corpus are:
hotel: 39554
room: 30251
staff: 18144
london: 15577
stay: 14150
service: 12920
great: 12509
breakfast: 11799
good: 10914
would: 10792
one: 9896
us: 9518
rooms: 8923
stayed: 8919
location: 8671
```

Because it is a unigram it is hard to get valuable sentiment from the review. In this case it is hard to know what sentiment hotel, room, staff has. In cases like this, it may be necessary to use n-grams with a higher value of n (e.g. bigrams or trigrams) to get a more accurate representation of the sentiment.

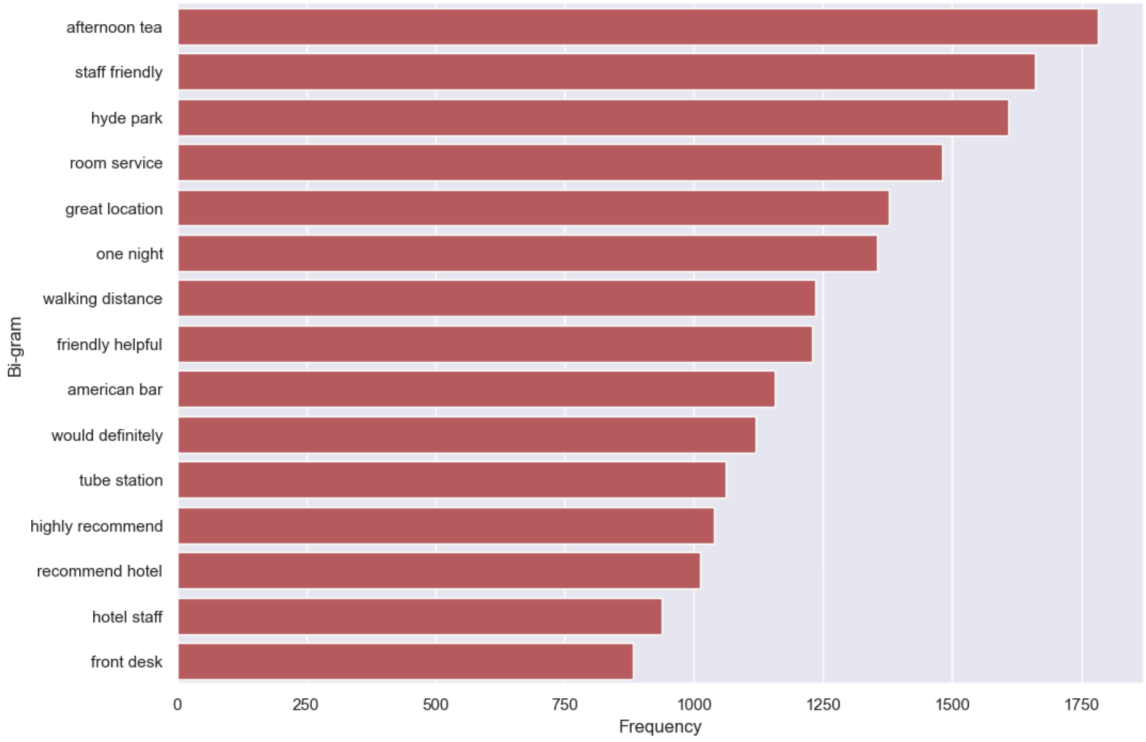


The bigram and trigram is created in a similar way as the unigram.

```
# Create an empty list to store all bi-grams in the corpus
all_bigrams = []

# Iterate over each review in the corpus and add all bi-grams to the all_bigrams list
for review in df['Corpus_Reviews']:
    bigrams = ngrams(review, 2)
    all_bigrams.extend(list(bigrams))

# Count the frequency of each bi-gram and get the 10 most common bi-grams
bigram_freq = Counter(all_bigrams)
top_bigrams = bigram_freq.most_common(15)

# Print the 15 most common bi-grams
print("The 15 most frequent bi-grams in the corpus are:")
for bigram, freq in top_bigrams:
    print(f"{bigram}: {freq}")
```

```
The 15 most frequent bi-grams in the corpus are:
('afternoon', 'tea'): 1782
('staff', 'friendly'): 1660
('hyde', 'park'): 1608
('room', 'service'): 1480
('great', 'location'): 1378
('one', 'night'): 1355
('walking', 'distance'): 1234
('friendly', 'helpful'): 1228
('american', 'bar'): 1156
('would', 'definitely'): 1119
('tube', 'station'): 1062
('highly', 'recommend'): 1040
('recommend', 'hotel'): 1013
('hotel', 'staff'): 937
('front', 'desk'): 883
```

The plot indicates that bigrams provide more valuable sentiment than unigrams. This is because bigrams provide more context and help to capture the nuances and complexities of language, such as sarcasm or irony, that may be missed by analyzing individual words in isolation. Additionally, bigrams can capture more specific phrases and expressions used in reviews, allowing for a more accurate understanding of the sentiment being expressed. This information can be useful for hoteliers to identify specific areas of improvement in their services and amenities, as well as to better understand the overall guest experience.

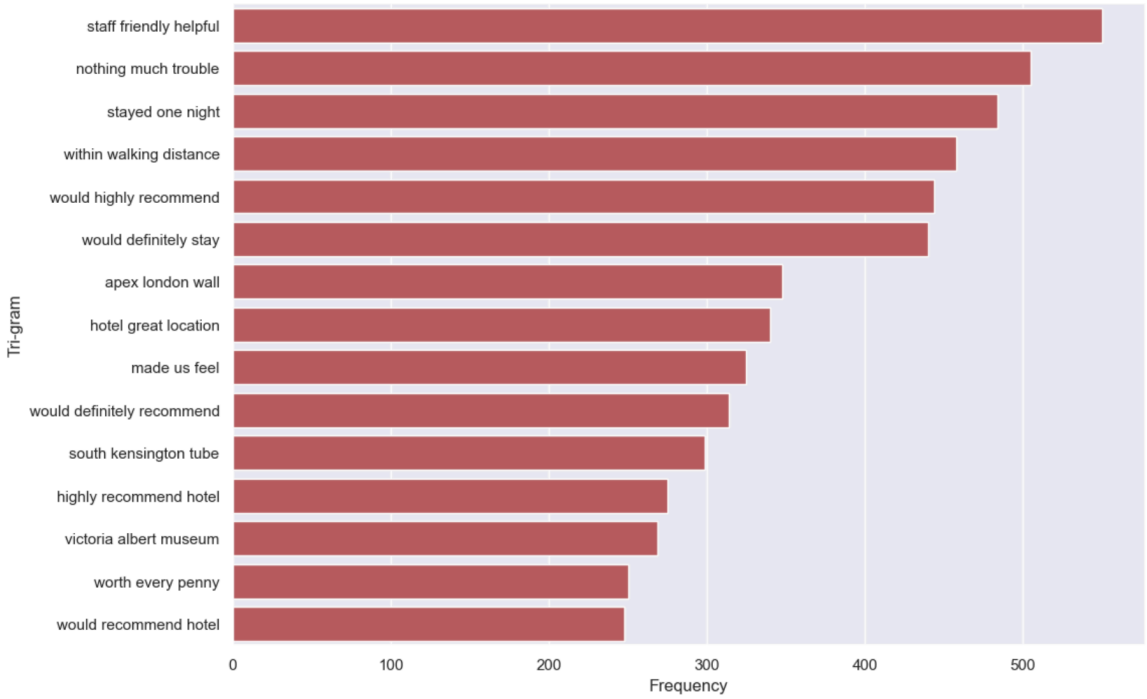


The trigram is also created in a similar way as bigram.

```
# Create an empty list to store all tri-grams in the corpus
all_trigrams = []

# Iterate over each review in the corpus and add all tri-grams to the all_trigrams list
for review in df['Corpus_Reviews']:
    trigrams = ngrams(review, 3)
    all_trigrams.extend(list(trigrams))

# Count the frequency of each tri-gram and get the 10 most common tri-grams
trigram_freq = Counter(all_trigrams)
top_trigrams = trigram_freq.most_common(15)

# Print the 15 most common tri-grams
print("The 15 most frequent tri-grams in the corpus are:")
for trigram, freq in top_trigrams:
    print(f"{trigram}: {freq}")
```

```
The 15 most frequent tri-grams in the corpus are:
('staff', 'friendly', 'helpful'): 550
('nothing', 'much', 'trouble'): 505
('stayed', 'one', 'night'): 484
('within', 'walking', 'distance'): 458
('would', 'highly', 'recommend'): 444
('would', 'definitely', 'stay'): 440
('apex', 'london', 'wall'): 348
('hotel', 'great', 'location'): 340
('made', 'us', 'feel'): 325
('would', 'definitely', 'recommend'): 314
('south', 'kensington', 'tube'): 299
('highly', 'recommend', 'hotel'): 275
('victoria', 'albert', 'museum'): 269
('worth', 'every', 'penny'): 250
('would', 'recommend', 'hotel'): 248
```

The use of trigrams in sentiment analysis provides a deeper understanding of the tonality of reviews. In this case, it's evident that the trigrams capture the sentiment of the reviews well, with most of them being positive. This is not surprising given that the dataset was heavily skewed towards high ratings. The trigrams offer valuable insight into what aspects of the hotel experience customers find particularly positive, which can be useful for hotel managers seeking to improve their services and enhance customer satisfaction.



Since the data is mostly skewed towards positive reviews, bigrams and trigrams will mostly consist of positive words. To plot unigrams, bigrams, and trigrams with negative sentiments, the NLTK library's Vader and SentimentIntensityAnalyzer, along with the WordNet package, are used. WordNet can be used for allocating negative words as it contains a list of synsets (groups of synonymous words) with antonyms (opposites) labeled as such. In fact, the NLTK library's WordNet module contains methods for finding antonyms for a given word or synset, which can be useful for sentiment analysis tasks.

Firstly, the cleaned text is converted back into a string using the join() method. Then, the SentimentIntensityAnalyzer() function is used to initialize the VADER sentiment analyzer.

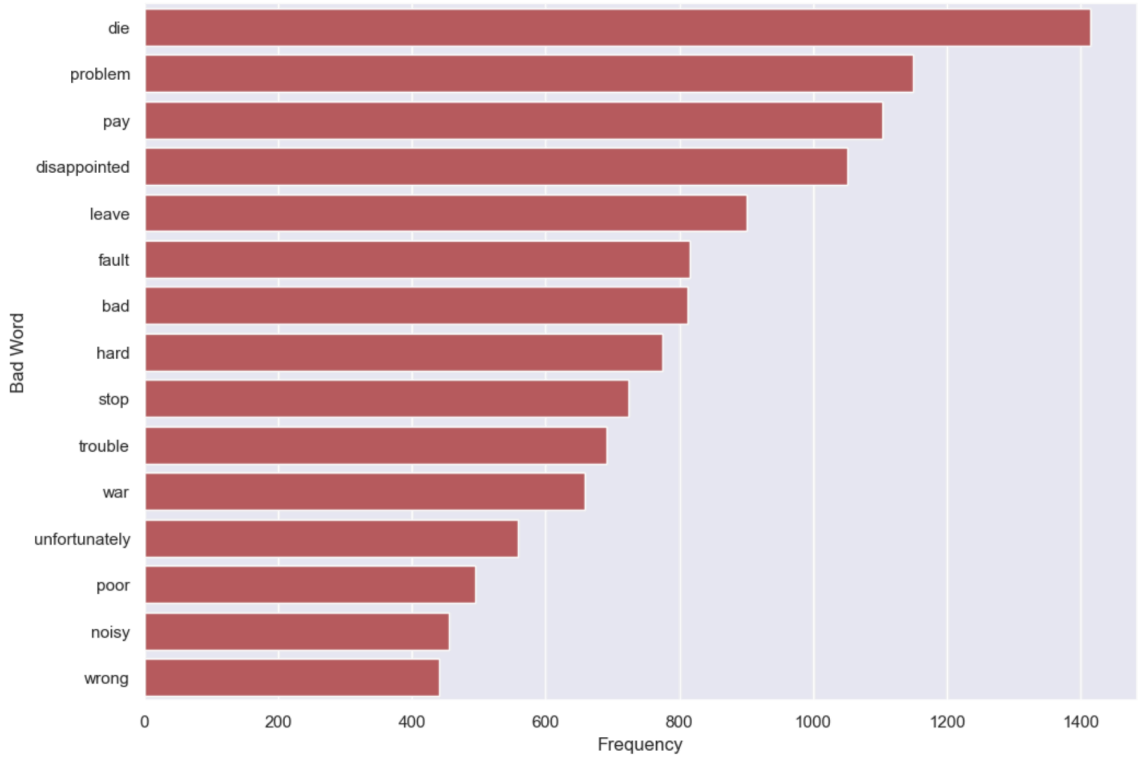
Next, the word_tokenize() function from the NLTK library is used to tokenize the string into individual words. Then, a for loop iterates over each word and assigns a sentiment score using the polarity_scores() method of the analyzer. A threshold of 0.5 is used to determine if the word is negative or not. If a word is found to be negative, it is added to the negative_words list.

Finally, the Counter() function from the collections module is used to count the frequency of each negative word, and the 15 most common negative words are stored in the top_words variable.

```
text = ' '.join(clean_text)  # join the list of strings using a space as separator

# Initialize the sentiment analyzer
analyzer = vader.SentimentIntensityAnalyzer()

# Tokenize the text into individual words
words = word_tokenize(text)

# Get the negative words
negative_words = []
for word in tqdm(words):
    score = analyzer.polarity_scores(word)['neg']
    if score > 0.5:  # Threshold for negative sentiment
        negative_words.append(word)

word_freq = Counter(negative_words)
top_words = word_freq.most_common(15)
```

```
100%|██████████| 1880113/1880113 [00:28<00:00, 66999.57it/s]
```

As we can see from the plot, it only displays the frequency of negative sentiment words. As mentioned earlier in the case of unigrams, they do not provide much useful information regarding the specific negative sentiment being expressed. However, by analyzing bigrams and trigrams, we can gain more insight into the context and tone of the negative sentiment.

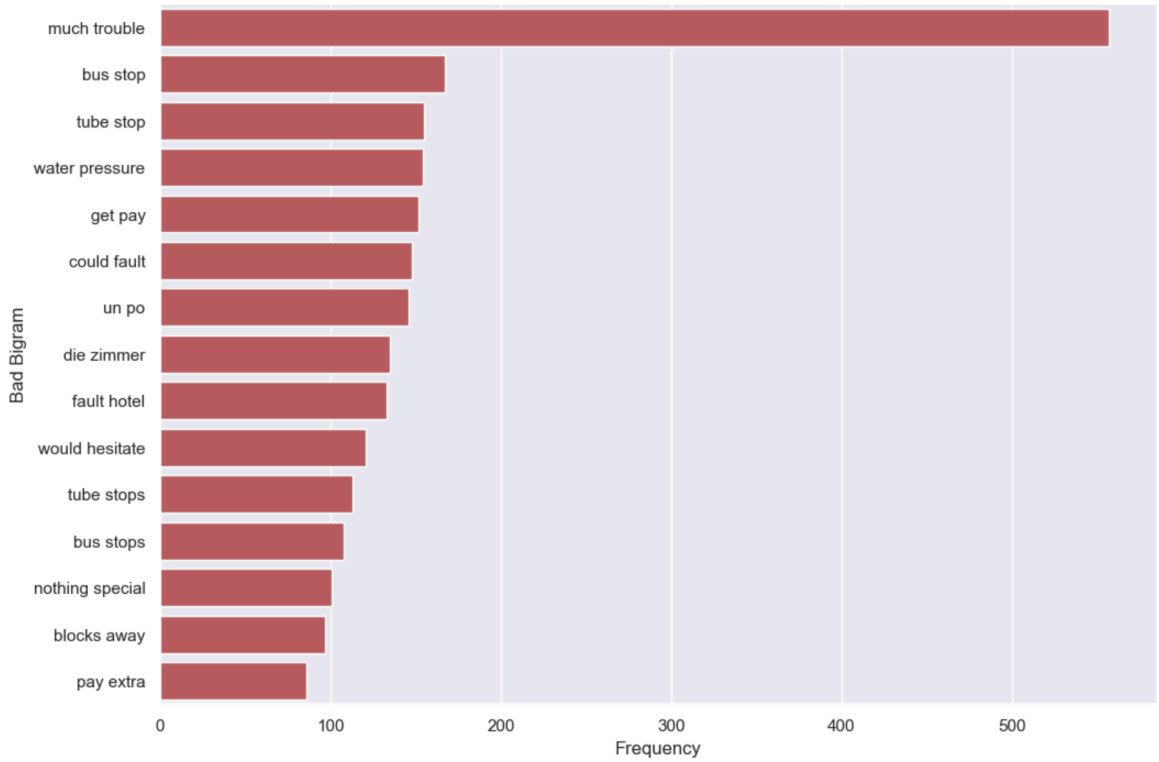


This code creates bigrams (pairs of words) from the tokenized words, converts them to strings, and then finds the negative bigrams (with a negative sentiment score above 0.5) using the VADER sentiment analyzer. The frequency of these negative bigrams is counted and the top 15 are stored in the variable top_bigrams.

```
# Create a bi-gram of the words
bigrams = ngrams(words, 2)

# Convert the bigrams into strings
bigram_strings = [' '.join(bigram) for bigram in bigrams]

# Get the negative bigrams
negative_bigrams = []
for bigram in tqdm(bigram_strings):
    score = analyzer.polarity_scores(bigram)['neg']
    if score > 0.5:  # Threshold for negative sentiment
        negative_bigrams.append(bigram)
bigram_freq = Counter(negative_bigrams)
top_bigrams = bigram_freq.most_common(15)
```

```
100%|██████████| 1880112/1880112 [00:44<00:00, 42060.74it/s]
```

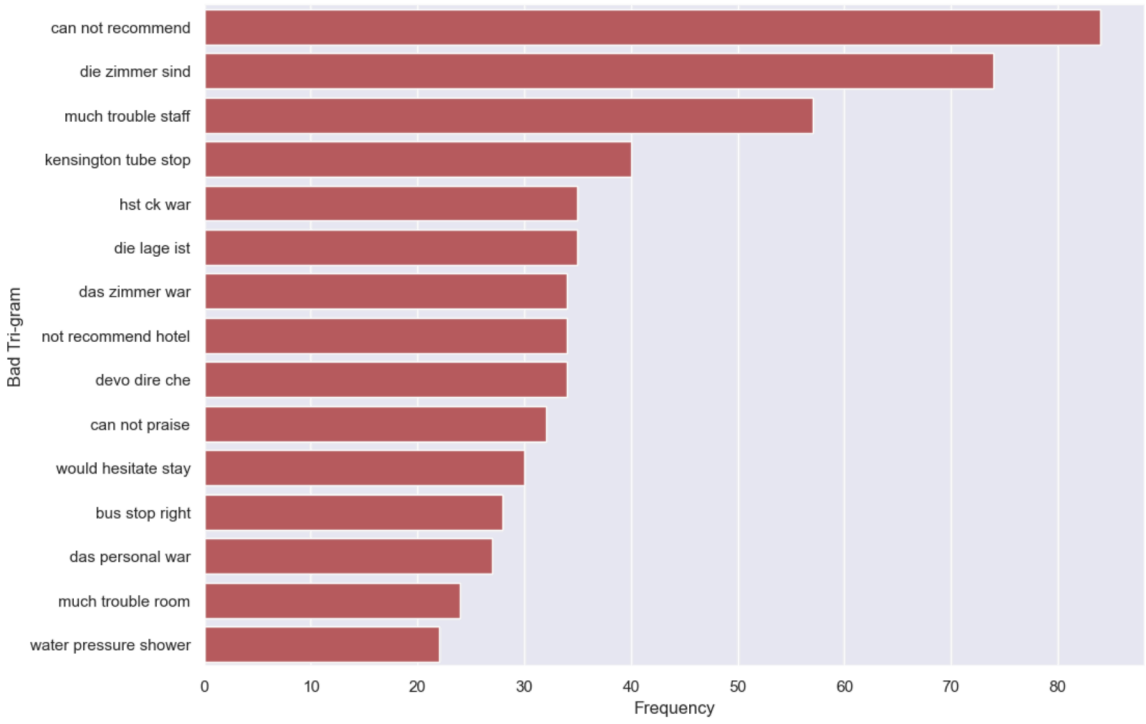
Analyzing these bigrams gives us some context about the negative sentiment in the reviews, but it can still be difficult to fully understand the context of the text e.g much trouble, could fault, would hesitate.




```
# Create tri-grams
trigrams = ngrams(words, n=3)

# Convert the trigrams into strings
trigram_strings = [' '.join(trigram) for trigram in trigram]

# Get the negative words
negative_trigrams = []
for trigram in tqdm(trigrams):
    score = analyzer.polarity_scores(' '.join(trigram))['neg']
    if score > 0.5:  # Threshold for negative sentiment
        negative_trigrams.append(' '.join(trigram))


# Get the top 15 bad tri-negative_trigrams
trigram_freq = Counter(negative_trigrams)
top_trigram = trigram_freq.most_common(15)
```

The trigram is created in a similar way to the bigram, except for the value of the n parameter which is set to 3 to create a trigram instead of a bigram (which had a value of 2).

This trigram analysis provides a greater level of insight into the negative sentiments expressed in the reviews, with a more contextual understanding of the issues customers are dissatisfied with. Hoteliers can utilize this information to identify areas for improvement and focus on addressing these specific issues.



Model Preparation

To prepare the data for the model, the dataset is split into training and testing, with the test size being 20% of the entire data, which in this case is 5466 data points.

```
# Cleaning review text which will then be saved to x

x = []

sentences = list(df['Review Text'])
for sen in tqdm(sentences):
    x.append(preprocess_text(sen))
```

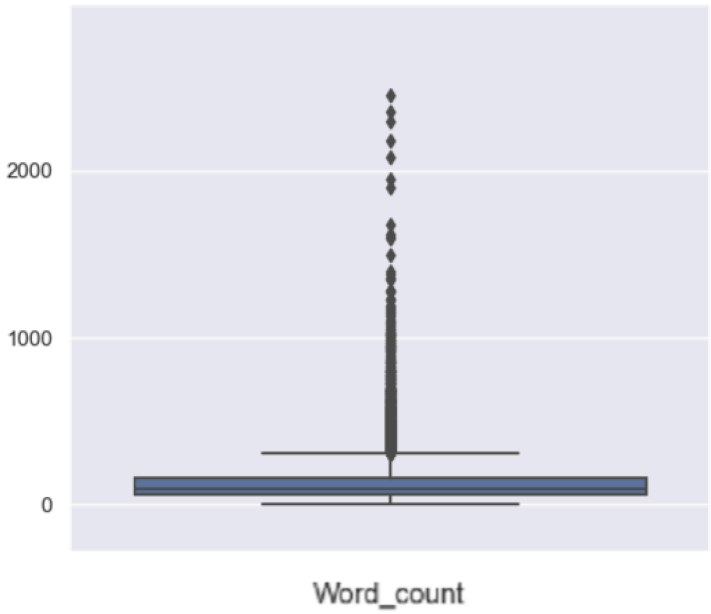
```
100%|██████████| 27330/27330 [00:07<00:00, 3517.67it/s]
```

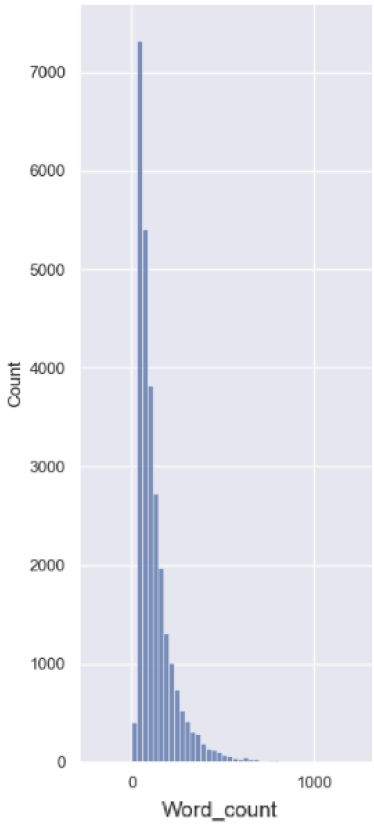
```
#x = df['Review Text']
x = np.array(x)
y = df['Positive']
```

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.20)

print("shape of x_train:", x_train.shape)
print("shape of x_test:", x_test.shape)

print("shape of y_train:", y_train.shape)
print("shape of y_test:", y_test.shape)
```

```
shape of x_train: (21864,)
shape of x_test: (5466,)
shape of y_train: (21864,)
shape of y_test: (5466,)
```

Next, we need to determine the number of words in our vocabulary to determine the size of the padding. This can be achieved through visualizing boxplots and distribution plots. It appears that a vocabulary size of around 300-350 is the sweet spot, as it allows us to capture most of the words in the dataset.





To ensure the validity or proximity of the observation a method called IQR (Interquartile Range). This method is a statistical technique used to detect outliers in a dataset. It involves calculating the difference between the 75th percentile (Q3) and the 25th percentile (Q1) of the data, known as the IQR. Any data points that lie more than 1.5 times the IQR below Q1 or above Q3 are considered outliers and can be removed from the dataset.

```
q1 = df["Word_count"].quantile(0.25)
q3 = df["Word_count"].quantile(0.75)
iqr = q3-q1

upper_limit = q3 + (1.5 * iqr)
lower_limit = q1 - (1.5 * iqr)

upper_limit, lower_limit

print(f'The upper limit, which is {upper_limit}, closely aligns with my observation. Therefore, I will round it u
```

```
The upper limit, which is 307.5, closely aligns with my observation. Therefore, I will round it up to 350.
```

And then, the reviews in the x_train and y_train datasets are tokenized. Here is an example of what a review looks like before being tokenized.

```
# This is what the words look like before getting tokenized
print(x_train[4])
print(y_train[4])
```

```
wife stayed aplex london wall november come one night stay special treat anniversary hotel close liverpool station
bank tube station heart city really nice welcome reception anh glass chilled prosseco given room balcony nice view
city room verybig great facilitiesandalovely coffee machine realcoffeeforonce thebed nicely presented swans made to
wels heart shape made red petals also received aplex branded box chocolates bathrobes fluffy slippers romantic inde
ed room clean huge comfortable bed white crisp sheets nice feather ducks pillows impressed man point view condition
room items paint ac mastic shower drain etc pristine condition impressive normally see hotel room nowadays declined
service secondhand thoughts accepted see wife enjoyed massive stand bath decorated rose petals nicetoiletries elemi
s brand white fluffy towels food delicious service quick overall amazing experience worth every penny paid definite
ly return longer stay also recommend friends family
1
```

```
# Let's check what our vocabulary size is
tokenizer = Tokenizer()

tokenizer.fit_on_texts(x_train.copy())

VOCAB_SIZE = len(tokenizer.word_index)
print("Vocabulary size:", VOCAB_SIZE)
```

```
Vocabulary size: 52005
```

```
VOCAB_SIZE = 52005

#, lower=True
tokenizer = Tokenizer(VOCAB_SIZE)
tokenizer.fit_on_texts(x_train)

# Providing each token an integer
x_train = tokenizer.texts_to_sequences(x_train)
x_test = tokenizer.texts_to_sequences(x_test)
```

The code first initializes a Tokenizer object and fits it on the training data to create a vocabulary of words. The size of the vocabulary is then printed as VOCAB_SIZE. The tokenizer is then reinitialized with the same vocabulary size and fitted on the training data again with the texts_to_sequences method used to convert the text data into sequences of integers. The resulting

sequences are then stored in x_train and x_test.

And here is an example of what a review looks like after being tokenized.

```
# This is what the words look like after beeing tokenized
print(x_train[4])
print(y_train[4])
```

```
[204, 14, 19813, 4, 397, 1725, 154, 11, 21, 5, 67, 224, 387, 1, 92, 1060, 78, 360, 68, 78, 768, 129, 24, 18, 214, 8
1, 26801, 518, 2428, 8457, 217, 2, 472, 18, 69, 129, 2, 26802, 7, 26803, 128, 785, 26804, 26805, 713, 649, 5721, 4
8, 559, 768, 3405, 48, 1546, 2274, 26, 487, 19813, 5393, 1351, 987, 2225, 1747, 936, 1578, 931, 2, 32, 239, 31, 43,
932, 2602, 1726, 18, 4308, 7620, 670, 478, 353, 481, 69, 2021, 2, 828, 3135, 952, 19814, 75, 4238, 165, 4154, 2021,
686, 1296, 133, 1, 2, 5563, 2429, 6, 26806, 5101, 3010, 133, 204, 127, 1160, 1165, 166, 391, 1660, 2274, 26807, 202
2, 1269, 932, 1747, 559, 29, 196, 6, 521, 167, 47, 42, 153, 63, 1004, 483, 66, 125, 961, 5, 26, 54, 346, 185]
1
```

And next step is to add padding to the text. Padding is used to ensure that all sequences in a dataset have the same length. LSTM models require input sequences to be of equal length, and so padding is necessary to make sure all sequences have the same length. This is important because LSTM models work by maintaining a hidden state across a fixed number of time steps, and so all input sequences need to have the same length to be processed by the model.

This is an example where the first five reviews length is shown before adding any padding.

```
# Word lenghts before padding
len(x_train[0]), len(x_train[1]), len(x_train[2]), len(x_train[3]), len(x_train[4])
```

```
(27, 73, 66, 57, 137)
```

The code sets the maximum length of the sequences to 350, which was determined in an earlier step, and then pads the sequences using the pad_sequences() function from Keras. Padding ensures that all sequences have the same length of 350, which is necessary for training the LSTM model. The padding argument is set to 'post', which adds padding to the end of the

sequences. The resulting padded sequences are stored in x_train and x_test.

```python
# Max length of words
MAXLEN = 350

x_train = keras.utils.pad_sequences(x_train, padding='post', maxlen=MAXLEN)
x_test = keras.utils.pad_sequences(x_test, padding='post', maxlen=MAXLEN)
```

If we look at same five reviews as earlier we see that all of them have the length of 350.

```python
# Word lenghts after padding
len(x_train[0]), len(x_train[1]), len(x_train[2]), len(x_train[3]), len(x_train[4])
```

```
(350, 350, 350, 350, 350)
```

And if we look closer into a review we can see the padding. In this case the zeroes represents the padding which is added after after the tokenized review.

```python
# And this is what it looks like if we take a closer look
x_train[4]
```

```
array([  204,    14, 19813,     4,   397,  1725,   154,    11,    21,
           5,    67,   224,   387,     1,    92,  1060,    78,   360,
          68,    78,   768,   129,    24,    18,   214,    81, 26801,
         518,  2428,  8457,   217,     2,   472,    18,    69,   129,
           2, 26802,     7, 26803,   128,   785, 26804, 26805,   713,
         649,  5721,    48,   559,   768,  3405,    48,  1546,  2274,
          26,   487, 19813,  5393,  1351,   987,  2225,  1747,   936,
        1578,   931,     2,    32,   239,    31,    43,   932,  2602,
        1726,    18,  4308,  7620,   670,   478,   353,   481,    69,
        2021,     2,   828,  3135,   952, 19814,    75,  4238,   165,
        4154,  2021,   686,  1296,   133,     1,     2,  5563,  2429,
           6, 26806,  5101,  3010,   133,   204,   127,  1160,  1165,
         166,   391,  1660,  2274, 26807,  2022,  1269,   932,  1747,
         559,    29,   196,     6,   521,   167,    47,    42,   153,
          63,  1004,   483,    66,   125,   961,     5,    26,    54,
         346,   185,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0])
```

Now y is one-hot encoded. The reason for encoding y is because machine learning models require numerical inputs and outputs. In this case, the sentiment labels (positive or negative) are represented as text labels ('positive' and 'negative'). To use these labels as outputs for the model, they need to be converted into a numerical format.

In the code, the to_categorical() function from Keras is used to perform one-hot encoding of the sentiment labels. This converts each label into a binary vector of length num_classes, where each element of the vector corresponds to a different class. For example, if num_classes is set to 2 (as it is in this code), each label will be converted into a binary vector of length 2, with a value of 1 in the element corresponding to the correct sentiment and a value of 0 in the other element. This encoding allows the model to understand the sentiment labels as numerical values that it can use for training and prediction.

```
# The assigned number is dependent on the type of rating being used, which is currently set to 0 or 1 in this ins
# However, it is possible to modify this if you have multilabel.
num_classes = 2

y_train = to_categorical(y_train, num_classes)
y_test = to_categorical(y_test, num_classes)

print(y_train.shape)
print(y_test[0])
```

```
(21864, 2)
[0. 1.]
```

Lastly it's time to reshape the data from 2D into 3D. Recurrent Neural Networks (RNNs) require 3D input because they are designed to operate on sequences of data. The 3D input tensor consists of a batch size, sequence length, and feature dimensions.

- Batch size: The number of sequences in a batch, typically represented as the first dimension of the 3D tensor.

- Sequence length: The number of time steps in each sequence, represented as the second dimension of the 3D tensor.
- Feature dimensions: The number of features or variables in each time step, represented as the third dimension of the 3D tensor.

The 3D input tensor allows RNNs to process a sequence of data over time, where each time step contains a set of features. By using a 3D input, RNNs can learn and remember patterns and dependencies across multiple time steps, which is useful in a variety of applications such as natural language processing, speech recognition, and time series forecasting.

```
x_train = np.array(x_train).reshape((x_train.shape[0], x_train.shape[1], 1))
x_test = np.array(x_test).reshape((x_test.shape[0], x_test.shape[1], 1))

print(x_train.shape)
print("Sentences:", x_train.shape[0], "Words:", x_train.shape[1], "TimeSteps:", x_train.shape[2], "\n")
print(x_test.shape)
print("Sentences:", x_test.shape[0], "Words:", x_test.shape[1], "TimeSteps:", x_test.shape[2])
```

```
(21864, 350, 1)
Sentences: 21864 Words: 350 TimeSteps: 1

(5466, 350, 1)
Sentences: 5466 Words: 350 TimeSteps: 1
```

Model Building

The code defines a neural network model for sentiment analysis using LSTM layers. The model has an embedding layer with an input dimension of VOCAB_SIZE, an output dimension of 50, and an input length of MAXLEN. This is followed by two LSTM layers with 32 and 16 units, respectively, and both with return_sequences=True to output a sequence. Dropout layers are used after each LSTM layer to prevent overfitting. The output from the second LSTM layer is flattened and passed through a dense layer with num_classes units and sigmoid activation function. The model is compiled and summarized using the summary() function.

```
# NN Model
model = tf.keras.Sequential([
tf.keras.layers.Embedding(input_dim=VOCAB_SIZE, output_dim=50, input_length=MAXLEN),
tf.keras.layers.LSTM(32, return_sequences=True),
tf.keras.layers.Dropout(0.2),
tf.keras.layers.LSTM(16, return_sequences=True),
tf.keras.layers.Dropout(0.2),
tf.keras.layers.Flatten(),
tf.keras.layers.Dense(num_classes , activation="sigmoid"),
])

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 350, 50)           2600250

 lstm (LSTM)                 (None, 350, 32)           10624

 dropout (Dropout)           (None, 350, 32)           0

 lstm_1 (LSTM)               (None, 350, 16)           3136

 dropout_1 (Dropout)         (None, 350, 16)           0

 flatten (Flatten)           (None, 5600)              0

 dense (Dense)               (None, 2)                 11202

=================================================================
Total params: 2,625,212
Trainable params: 2,625,212
Non-trainable params: 0
_________________________________________________________________
```

Before training the model we need to deal with the imbalanced data. It's important to adjust the weights of the classes to reflect their distribution. This can be done by calculating the class weights and passing them to the model during training.

The class weights can be calculated using various techniques such as inverse frequency, inverse square root frequency, or user-defined weights. Inverse frequency assigns a weight to each class that is inversely proportional to the number of samples in that class. Inverse square root frequency assigns a weight that is the inverse square root of the frequency of the class. User-defined weights allow the user to manually assign weights to each class based on their domain knowledge.

The chosen method used in this project is inverse frequency as you can see below. The result shows that the weight assigned to the negative (0) class has to be around 9 times that of the positive (1) class to balance the data.

```
# Frequency
freq = pd.value_counts(df['Positive'])
freq
```

```
Positive
1    24347
0     2983
Name: count, dtype: int64
```

```
# Inverse frequency
weights = {0: freq.sum() / freq[0], 1: freq.sum() / freq[1]}
weights
```

```
{0: 9.161917532685216, 1: 1.122520228364891}
```

The last thing before training the model is to add and uses three different callbacks to improve the performance of the LSTM model during training.

The first callback is ModelCheckpoint, which saves the best model based on the validation loss. The saved model can later be used to make predictions on new data or to resume training. The save_best_only parameter is set to True, which means that only the best model will be saved.

The second callback is EarlyStopping, which stops the training process if the validation loss does not improve after a certain number of epochs (patience). In this case, the training will stop after 4 epochs if the validation loss does not improve. This helps to prevent overfitting and saves computation time.

The third callback is CSVLogger, which creates a CSV file and saves the model metrics in it. This allows for easy tracking and visualization of the model performance over time.

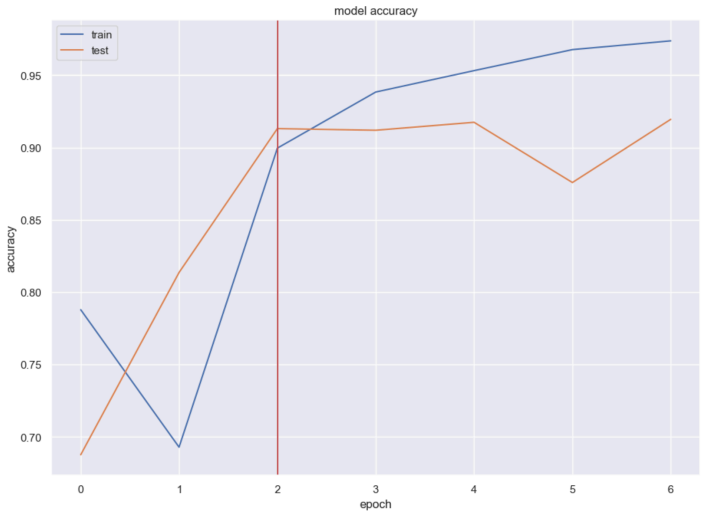
All three callbacks are put into a list called callback_list, which is passed as an argument to the fit() method of the LSTM model.

```
# Creating checkpoint that saves the best model
checkpoint = ModelCheckpoint('model/', save_best_only=True)
#, monitor='val_accuracy', mode='max'

# Stop the training if the val_loss does not improve on next 4 epochs
early_stop = EarlyStopping(monitor='val_loss', patience=4, verbose=1)

# Creates a csv file and saves the model metrics in it
log_csv = CSVLogger('my_logs.csv', separator=',', append=False)

# A lsit with the 3 instances that will be put into callbacks in model.fit()
callback_lsit = [checkpoint, early_stop, log_csv]
```

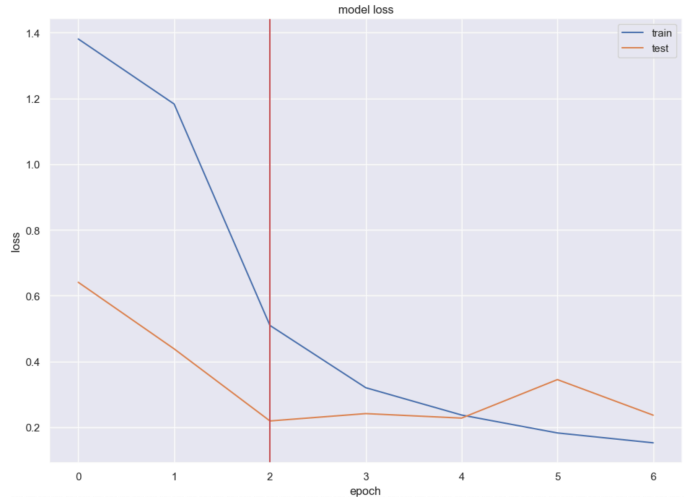
Now it is time to train the model. The training stopped at epoch 7 because the model loss score did not improve after epoch 2.

```
# Creating an instance of BinaryCrossentropy to pass into model.compile()
loss_fn = tf.keras.losses.BinaryCrossentropy()

# Configure the model for training.
model.compile(optimizer=Adam(learning_rate=(0.0001)), loss=loss_fn, metrics=['accuracy', AUC(name='AUC')])

# Train a model
history = model.fit(x_train, y_train, batch_size=150, epochs=20, validation_split=0.20, class_weight=weights, cal

# print the best epoch
print(f"The best epoch is {early_stop.best_epoch}.")
```

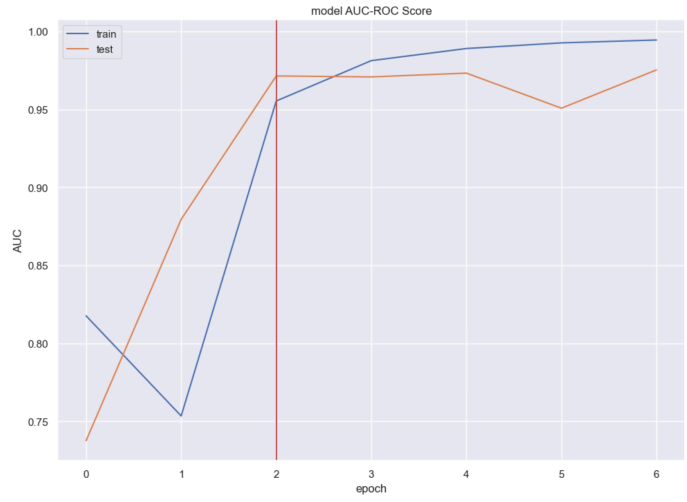
```
Epoch 1/20
117/117 [==============================] - ETA: 0s - loss: 1.3810 - accuracy: 0.7879 - AUC: 0.8176
WARNING:absl:Found untraced functions such as _update_step_xla, lstm_cell_layer_call_fn, lstm_cell_layer_call_and_r
eturn_conditional_losses, lstm_cell_1_layer_call_fn, lstm_cell_1_layer_call_and_return_conditional_losses while sav
ing (showing 5 of 5). These functions will not be directly callable after loading.
INFO:tensorflow:Assets written to: model\assets
INFO:tensorflow:Assets written to: model\assets
117/117 [==============================] - 66s 537ms/step - loss: 1.3810 - accuracy: 0.7879 - AUC: 0.8176 - val_los
s: 0.6408 - val_accuracy: 0.6876 - val_AUC: 0.7376
Epoch 2/20
117/117 [==============================] - ETA: 0s - loss: 1.1831 - accuracy: 0.6929 - AUC: 0.7534
WARNING:absl:Found untraced functions such as _update_step_xla, lstm_cell_layer_call_fn, lstm_cell_layer_call_and_r
eturn_conditional_losses, lstm_cell_1_layer_call_fn, lstm_cell_1_layer_call_and_return_conditional_losses while sav
ing (showing 5 of 5). These functions will not be directly callable after loading.
INFO:tensorflow:Assets written to: model\assets
INFO:tensorflow:Assets written to: model\assets
117/117 [==============================] - 58s 496ms/step - loss: 1.1831 - accuracy: 0.6929 - AUC: 0.7534 - val_los
s: 0.4382 - val_accuracy: 0.8136 - val_AUC: 0.8795
Epoch 3/20
117/117 [==============================] - ETA: 0s - loss: 0.5095 - accuracy: 0.8996 - AUC: 0.9554
WARNING:absl:Found untraced functions such as _update_step_xla, lstm_cell_layer_call_fn, lstm_cell_layer_call_and_r
eturn_conditional_losses, lstm_cell_1_layer_call_fn, lstm_cell_1_layer_call_and_return_conditional_losses while sav
ing (showing 5 of 5). These functions will not be directly callable after loading.
INFO:tensorflow:Assets written to: model\assets
INFO:tensorflow:Assets written to: model\assets
117/117 [==============================] - 63s 539ms/step - loss: 0.5095 - accuracy: 0.8996 - AUC: 0.9554 - val_los
s: 0.2190 - val_accuracy: 0.9131 - val_AUC: 0.9714
Epoch 4/20
117/117 [==============================] - 52s 443ms/step - loss: 0.3201 - accuracy: 0.9384 - AUC: 0.9813 - val_los
s: 0.2412 - val_accuracy: 0.9120 - val_AUC: 0.9708
Epoch 5/20
117/117 [==============================] - 54s 462ms/step - loss: 0.2369 - accuracy: 0.9532 - AUC: 0.9890 - val_los
s: 0.2276 - val_accuracy: 0.9174 - val_AUC: 0.9733
Epoch 6/20
117/117 [==============================] - 54s 464ms/step - loss: 0.1824 - accuracy: 0.9677 - AUC: 0.9927 - val_los
s: 0.3447 - val_accuracy: 0.8758 - val_AUC: 0.9508
Epoch 7/20
117/117 [==============================] - 55s 469ms/step - loss: 0.1524 - accuracy: 0.9738 - AUC: 0.9945 - val_los
s: 0.2364 - val_accuracy: 0.9195 - val_AUC: 0.9753
Epoch 7: early stopping
The best epoch is 2.
```



Looking at this output at best epoch, the accuracy score for the training set is 0.90%, while the accuracy score for the validation set is 0.91%. This suggests that the model is performing well and not overfitting too much to the training data.



The loss scores are also decreasing with each epoch until epoch 2, which is a good sign that the model is learning from the data.



The ROC AUC score on the best epoch on the training set is 0.96, while the validation set is 0.97 which indicates that the model is performing well. A value of 0.5 indicates random guessing, while a value of 1 indicates perfect

performance. A score between 0.7 to 0.9 is considered good, while a score above 0.9 is considered excellent. This model is performing well in terms of its ability to distinguish between positive and negative samples.

After training the LSTM model it is time to test how good it is. The test data is now used to predict on the best model in this case epoch 2.

```
best_model = load_model('model/')

test_predictions = (best_model.predict(x_test) > 0.5).astype(int)
```

```
171/171 [==============================] - 7s 37ms/step
```

	precision	recall	f1-score	support
0	0.60	0.75	0.67	579
1	0.97	0.94	0.95	4887
micro avg	0.92	0.92	0.92	5466
macro avg	0.78	0.85	0.81	5466
weighted avg	0.93	0.92	0.92	5466
samples avg	0.92	0.92	0.92	5466

```
------------------------------------------------------------
The best epoch: 2
------------------------------------------------------------
```

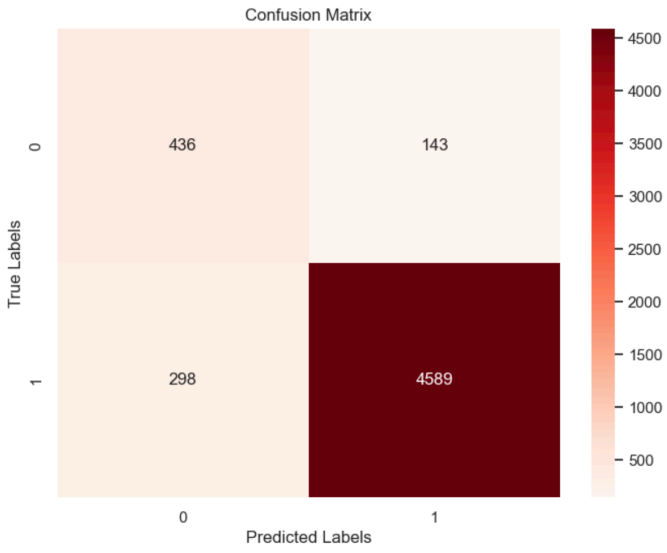
This model with the best epoch has a precision of 0.60 for class 0 (low recall) and 0.97 for class 1 (high precision), meaning that when the model predicts class 0, it is correct 60% of the time, while when it predicts class 1, it is correct 97% of the time.

The recall score is 0.75 for class 0 and 0.94 for class 1, meaning that the model is able to identify 75% of the true class 0 cases and 94% of the true class 1 cases.

The F1-score is a weighted average of precision and recall, with values ranging from 0 to 1, where 1 indicates the best performance. In this case, the F1-score is 0.67 for class 0 and 0.95 for class 1.
The weighted average of precision, recall, and F1-score indicates an overall performance of the model across both classes, where the performance of each class is weighted by the number of true cases in that class. In this case, the weighted average precision, recall, and F1-score are 0.93, 0.92, and 0.92, respectively, indicating a good overall performance of the model.

The micro and macro averages are additional ways to aggregate the performance scores across the classes, with the micro average being weighted by the total number of true cases, and the macro average being unweighted.

Based on the output, the confusion matrix has four values - true negatives (TN), false positives (FP), false negatives (FN), and true positives (TP). Here, the model predicted 436 true negatives and 4589 true positives. It also predicted 143 false positives and 298 false negatives.



Model Comparison

A completely new dataset will be used containing hotel reviews to test the performance of our LSTM model. We will use the pre-trained model from the best epoch (in this case, epoch 2) to predict the sentiment of the new reviews, and save the results in a new column. However, before training the model with this new data, we will need to clean, tokenize, and pad the reviews using the functions and variables that we have defined earlier. For more details on these steps, please refer to the notebook.

After predicting the new dataset, a new column named 'LSTM_Prediction_Positive' is created and saved, which contains ones and zeroes indicating the predicted sentiment of the hotel reviews.

```
# Let's evaluate the model to see how well it did
print(classification_report(data['Positive'], data['LSTM_Prediction_Positive']))
```

```
              precision    recall  f1-score   support

           0       0.73      0.80      0.76      5398
           1       0.93      0.89      0.91     15093

    accuracy                           0.87     20491
   macro avg       0.83      0.85      0.84     20491
weighted avg       0.87      0.87      0.87     20491
```

```
data.head()
```

	Review	Rating	Positive	LSTM_Prediction_Positive
0	nice hotel expensive parking got good deal sta...	4	1	0
1	ok nothing special charge diamond member hilto...	2	0	0
2	nice rooms not 4* experience hotel monaco seat...	3	0	1
3	unique, great stay, wonderful time hotel monac...	5	1	1
4	great stay great stay, went seahawk game aweso...	5	1	1

Then a function is created to check if two columns contain the same values which is saved inte a new column called ‘Positive_VS_LSTM_Positive'. If True, both have 1 or 0, and if False, they don't match.

```
# function to compare values in two columns and return boolean
def check_cols(df:str, col1:str, col2:str):
    return df[col1] == df[col2]
```

```
# apply function to each row using the apply method
data['Positive_VS_LSTM_Positive'] = data.apply(lambda x: check_cols(x, 'Positive', 'LSTM_Prediction_Positive'), a

data.head()
```

	Review	Rating	Positive	LSTM_Prediction_Positive	Positive_VS_LSTM_Positive
0	nice hotel expensive parking got good deal sta...	4	1	0	False
1	ok nothing special charge diamond member hilto...	2	0	0	True
2	nice rooms not 4* experience hotel monaco seat...	3	0	1	False
3	unique, great stay, wonderful time hotel monac...	5	1	1	True
4	great stay great stay, went seahawk game aweso...	5	1	1	True

When analyzing the performance of the model, it was found that the percentage of True values when comparing the LSTM predictions with the original Positive column is 86.82%, while the percentage of False values is 13.18%. This means that the model is around 87% accurate.

```
Positive_VS_LSTM_Positive
True      17790
False      2701
Name: count, dtype: int64


LSTM & VADER - Percentage of true values: 86.82%
LSTM & VADER - Percentage of false values: 13.18%
------------------------------------------------------------
```

Next is to predict the sentiment on new dataset using VADER model and have it as a benchmark against the LSTM model.

First, it creates an empty list called data_clean_text and then applies the preprocess_text function to each review of the data_clean_sentences list, storing the cleaned reviews in data_clean_text.
Second, it initializes the sentiment analyzer SentimentIntensityAnalyzer() from the NLTK library and uses it to predict the sentiment of each review in data_clean_text. The compound score obtained from the polarity_scores method is used to classify a review as positive or negative, with a threshold of 0.
Third, it creates a new DataFrame data_predictions_df with the predicted class labels and concatenates it with the original DataFrame data. The new column containing the VADER predictions is called VADER_Prediction_Positive.

```
# Empty list where the cleaned text will be stored
data_clean_text = []

data_clean_sentences = list(data['Review'])
for sen in tqdm(data_clean_sentences):
    data_clean_text.append(preprocess_text(sen))
```

```
100%|██████████| 20491/20491 [00:05<00:00, 3920.93it/s]
```

```
# Initialize the sentiment analyzer
analyzer = SentimentIntensityAnalyzer()

# Use the analyzer to get the sentiment for each review
vader_predictions = []
for text in tqdm(data_clean_text):
    score = analyzer.polarity_scores(text)
    if score['compound'] >= 0:
        vader_predictions.append(1)  # Positive sentiment
    else:
        vader_predictions.append(0)  # Negative sentiment
```

```
100%|██████████| 20491/20491 [00:17<00:00, 1202.64it/s]
```

```
# Create a DataFrame with the predicted class labels
data_predictions_df = pd.DataFrame(vader_predictions, columns=['VADER_Prediction_Positive'])

# Concatenate the original DataFrame with the new predictions DataFrame
data = pd.concat([data, data_predictions_df], axis=1)
```

```
# apply function to each row using the apply method
data['Positive_VS_VADER_Positive'] = data.apply(lambda x: check_cols(x, 'Positive', 'VADER_Prediction_Positive'), axis=1)

data.head()
```

	Review	Rating	Positive	LSTM_Prediction_Positive	Positive_VS_LSTM_Positive	VADER_Prediction_Positive	Positive_VS_VADER_Positive
0	nice hotel expensive parking got good deal sta...	4	1	0	False	1	True
1	ok nothing special charge diamond member hilto...	2	0	0	True	1	False
2	nice rooms not 4* experience hotel monaco seat...	3	0	1	False	1	False
3	unique, great stay, wonderful time hotel monac...	5	1	1	True	1	True
4	great stay great stay, went seahawk game aweso...	5	1	1	True	1	True

When analyzing the performance of the model, it was found that the percentage of True values when comparing the VADER predictions with the original Positive column is 78.44%, while the percentage of False values is 21.56%. This means that the model is around 78% accurate.

```
Positive_VS_VADER_Positive
True     16074
False     4417
Name: count, dtype: int64


LSTM & VADER - Percentage of true values: 78.44%
LSTM & VADER - Percentage of false values: 21.56%
------------------------------------------------------------
```

A new column called LSTM_Positive_VS_VADER_Positive is created where their prediction is compared against each other.

LSTM_Positive_VS_VADER_Positive
False
False
True
True
True

The evaluation shows that out of 20491 rows, there are 15525 rows where the LSTM and VADER predictions match (True), and 4966 rows where they do not match (False). The percentage of True values is 75.76%, while the percentage of False values is 24.24%.

```
LSTM_Positive_VS_VADER_Positive
True     15525
False     4966
Name: count, dtype: int64


LSTM & VADER - Percentage of true values: 75.76%
LSTM & VADER - Percentage of false values: 24.24%
------------------------------------------------------------
```

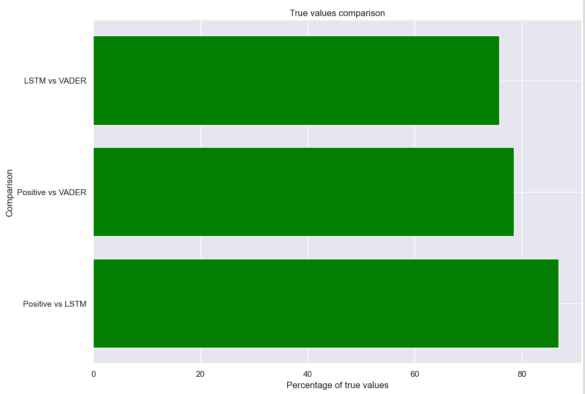
Results

Positive_VS_LSTM_Positive: Out of 20491 data points, 17790 (86.82)% were classified as true by LSTM, and 2701 (13.18)% were classified as false.

Positive_VS_VADER_Positive: Out of 20491 data points, 16074 (78.44)% were classified as true by LSTM, and 4417 (21.56)% were classified as false.

LSTM_Positive_VS_VADER_Positive: Out of 20491 data points, 15525 (75.76%) were classified as true by both LSTM and VADER, and 4966 (24.24%) were classified as false.

These results suggest that the LSTM model performs slightly better than VADER in identifying positive sentiment, but when both models agree, they are correct about 76% of the time.




```
Positive_VS_LSTM_Positive
True     17790
False     2701
Name: count, dtype: int64


LSTM & VADER - Percentage of true values: 86.82%
LSTM & VADER - Percentage of false values: 13.18%
------------------------------------------------------------
Positive_VS_VADER_Positive
True     16074
False     4417
Name: count, dtype: int64


LSTM & VADER - Percentage of true values: 78.44%
LSTM & VADER - Percentage of false values: 21.56%
------------------------------------------------------------
LSTM_Positive_VS_VADER_Positive
True     15525
False     4966
Name: count, dtype: int64


LSTM & VADER - Percentage of true values: 75.76%
LSTM & VADER - Percentage of false values: 24.24%
------------------------------------------------------------
```

According to the classification report, the LSTM model has an overall accuracy of 0.87, with an F1-score of 0.76 for predicting negative reviews and 0.91 for predicting positive reviews.
The precision score for negative reviews is 0.73, while it is 0.93 for positive reviews.

On the other hand, the VADER model has an overall accuracy of 0.78, with an F1-score of 0.32 for predicting negative reviews and 0.87 for predicting positive reviews.
The precision score for negative reviews is 0.96, while it is 0.77 for positive reviews.

The macro average F1-score for the LSTM model is 0.84, while it is only 0.59 for the VADER model. This indicates that the LSTM model performs better overall.

In summary, based on the provided classification report, the LSTM model outperforms the VADER model in terms of accuracy and F1-score. While the VADER model has a higher precision score for negative reviews, it performs poorly in predicting negative reviews overall, whereas the LSTM model has a more balanced performance.

LSTM Prediction Score:

	precision	recall	f1-score	support
0	0.73	0.80	0.76	5398
1	0.93	0.89	0.91	15093
accuracy			0.87	20491
macro avg	0.83	0.85	0.84	20491
weighted avg	0.87	0.87	0.87	20491

VADER Prediction Score:

	precision	recall	f1-score	support
0	0.96	0.19	0.32	5398
1	0.77	1.00	0.87	15093
accuracy			0.78	20491
macro avg	0.87	0.59	0.59	20491
weighted avg	0.82	0.78	0.73	20491

Discussion

The use of natural language processing (NLP) tools such as LSTM and NLTK has become increasingly popular in recent years for analyzing text data. In this research, the effectiveness of these tools for sentiment analysis of hotel reviews was evaluated. The findings of this study demonstrate that LSTM and NLTK are effective NLP tools for sentiment analysis of hotel reviews.

The insights gained from sentiment analysis can be valuable in identifying and addressing issues that impact guest experiences. By analyzing the most common word combinations and phrases used in negative reviews, hoteliers can identify the most pressing problem areas and improve their services. This can result in increased customer satisfaction and loyalty.

Improving the effectiveness of LSTM and NLTK for sentiment analysis can also make it easier for hoteliers to conduct sentiment analysis on large amounts of data and quickly identify areas for improvement in their services. The findings of this research are consistent with previous research in the field, which has also demonstrated the effectiveness of NLP tools for sentiment analysis.

However, it is important to acknowledge the limitations of this study, such as the restricted timeframe and the use of only text data for analysis. Further research could explore the impact of other factors, such as the use of multimedia content, on sentiment analysis of hotel reviews.

The implications of these results are significant for both theory and practice. From a theoretical perspective, this research contributes to our understanding of the effectiveness of NLP tools for sentiment analysis. From

a practical perspective, the findings of this study can be applied in hotel management to improve the guest experience and increase customer satisfaction and loyalty.

There are still unanswered questions that emerged from this research, such as the potential biases in the dataset and the impact of different training parameters on the accuracy of the models. These questions could be addressed in future studies to further improve the effectiveness of sentiment analysis in the hotel industry.

In conclusion, the effectiveness of LSTM and NLTK for sentiment analysis of hotel reviews has been demonstrated. The insights gained from this analysis can be used to improve the guest experience and increase customer satisfaction and loyalty. The findings of this study have important implications for both theory and practice, and future research could further improve the effectiveness of sentiment analysis in the hotel industry.

References

PromptCloudHQ. (2018). *Reviews of London-based hotels*. https://www.kaggle.com/PromptCloudHQ/reviews-of-londonbased-hotels [2023-04-24]

LARXEL. (2020). *Trip Advisor Hotel Reviews*. https://www.kaggle.com/datasets/andrewmvd/trip-advisor-hotel-reviews [2023-04-24]

Appendices

Here is a link to the Jupyter Notebook the project was worked on: Github