**NLP Task 2: Sentiment Analysis (750 words)**

***Literature Review and Rationale: (150 words)***

Sentiment analysis is an NLP process that aims to classify language data with respect to underlying attitudes, emotions, or sentiment. A statement can be classified along a continuum from positive to negative sentiment. These data can then be used to evaluate opinions towards any number of topics. Sentiment analysis can be applied to individuals, but it is particularly useful for evaluating public sentiment on a given topic. User-generated content websites are an ideal source of sentiment data, and the results can inform decision-making in several situations.

Sentiment analysis is a natural language processing tool that is useful for monitoring Web 2.0 applications, as it can reveal public opinion about numerous issues without requiring satisfaction enquiries.2,3 According to the Oxford dictionary, sentiment analysis is the process of computationally identifying and categorizing opinions expressed in a piece of text to determine whether the writer’s attitude toward a particular topic, product, and so on is generally positive, negative, or neutral. The interest in sentiment analysis has increased significantly over the last few years due to the large amount of stored text in Web 2.0 applications and the importance of online customer opinions. As a result, more than 1 million research papers contain the term “sentiment analysis,” and various start-ups have been created to analyze sentiments in social media companies.

Multiple studies on TripAdvisor exist, but there is no complete analysis from the sentiment analysis viewpoint. This article proposes TripAdvisor as a source of data for sentiment analysis tasks. We develop an analysis for studying the matching between users’ sentiments and automatic sentiment-detection algorithms. Finally, we discuss some of the challenges regarding sentiment analysis and TripAdvisor, and conclude with some final remarks.

***Data: (200)***

*Wrangling*

VADER (Valence Aware Dictionary and Sentiment Reasoner) was used to get sentimental scores of the user reviews and convert them into three categorical sentiments (positive, negative, and neutral). VADER is a lexicon and rule-based sentiment analysis tool that is specifically attuned to sentiments expressed in social media. The following code was used create two new columns that gives the sentiment score and the sentiment category of the review:

# Creating sentimental polarity

analyzer = SentimentIntensityAnalyzer()

def compound\_score(txt):

return analyzer.polarity\_scores(txt)["compound"]

# Sentiments

def sentiment(score):

emotion = ""

if score >= 0.5:

emotion = "Positive"

elif score <= -0.5:

emotion = "Negative"

else:

emotion = "Neutral"

return emotion

# Applying compound score

polarity\_scores = df["review"].astype("str").apply(compound\_score)

df["Sentiment\_Score"] = polarity\_scores

## Applying Sentiment

df["Sentiment"] = df["Sentiment\_Score"].apply(sentiment)

Fig x displays the output of the dataframe.

Text

Description automatically generated

It was determined that no missing data was present in data frame using the **isna().sum()** function.

*Summary of data for NLP task*

After checking our data for class balance, this report found that most of the user reviews were positive (Fig. x).

*A picture containing chart

Description automatically generated*

To balance the classes, this study performed down-sampling and created a subset of the data which contained the top 1000 rows from each of the three ‘Sentiment’ categories (Fig. x). This was achieved via the following code:

# Function to retrieve top few number of each category

def get\_top\_data(top\_n = 1000):

top\_data\_df\_positive = df[df['Sentiment'] == 'Positive'].head(top\_n)

top\_data\_df\_negative = df[df['Sentiment'] == 'Negative'].head(top\_n)

top\_data\_df\_neutral = df[df['Sentiment'] == 'Neutral'].head(top\_n)

top\_data\_df\_small = pd.concat([top\_data\_df\_positive, top\_data\_df\_negative, top\_data\_df\_neutral])

return top\_data\_df\_small

# Function call to get the top 1000 from each sentiment

df = get\_top\_data(top\_n=1000)

# After selecting top few samples of each sentiment

print("After segregating and taking equal number of rows for each sentiment:")

print(df['Sentiment'].value\_counts())

df.head(10)

*Chart, bar chart

Description automatically generated*

*Visualisation of data*

Fig. x illustrates that people who have given 5-star ratings have the highest positive sentiment. Whereas at lower ratings, the sentiment is mostly negative.

*Chart, bar chart

Description automatically generated*

A pie chart using the **Plotly** library (Fig. x) displays the distribution of different ratings. The range is between 16.5% (2-star review) and 23.1% (5-star review) which demonstrates the distribution is fairly well spread across the ratings.

***Chart, pie chart

Description automatically generated***

Using the **WordCloud** package, three separate word clouds were created to understand the frequency of different words used in the reviews. The most common words used in all three Sentiments was a hotel, room and stay (Fig. x).

***Text

Description automatically generated with low confidence***

The **Gensim** package and **Counter** function package was used to find keywords and build a graph using tokens from the text (Fig. x). Apart from hotel rooms, users talked about staff and parking.

***Chart, bar chart

Description automatically generated***

Most used words were removed to improve the performance and increase accuracy of the model.

words = ["hotel","room","rooms","hotels"]

for x in words:

data["review"] = data["review"].astype(str).str.replace(x,"")

Stop words are the words which are commonly used and removed from the sentence as pre-step in different Natural Language Processing (NLP) tasks. Example of stop words are: ‘a’, ‘an’, ‘the’, ‘this’, ‘not’ etc. Every tool uses a bit different set of stop words list that it removes but this technique is avoided in cases where phrase structure matters like in this case of Sentiment Analysis. Removal of stop words removes necessary words required to get the sentiment and sometimes it can totally change the meaning of the sentence. For this reason, we have skipped the removal of stop words for our sentiment analysis.

Tokenization is the process in which the sentence/text is split into array of words called tokens. This helps to do transformations on each word separately and is also required to transform words to numbers. The **[simple\_preprocess](https://radimrehurek.com/gensim/utils.html" \t "_blank)** function allows you to convert text to lower case and remove punctuations. It has min and max length parameters as well which help to filter out rare words and most commonly words which will fall in that range of lengths. In this study, **simple\_preprocess** is used to get the tokens for the data frame as it does most of the pre-processing under the hood. The following code was used to tokenize the review text.

# Tokenisation

from gensim.utils import simple\_preprocess

# Tokenize the text column to get the new column 'tokenized\_text'

df['tokenized\_text'] = [simple\_preprocess(line, deacc=True) for line in df['review']]

Text

Description automatically generated

The Stemming process reduces the words to its’ root word. Unlike Lemmatization which uses grammar rules and dictionary for mapping words to root form, stemming simply removes suffixes/prefixes. Stemming is widely used in the application of SEOs, Web search results, and information retrieval since as long as the root matches in the text somewhere it helps to retrieve all the related documents in the search. The following code was used to stem the tokens in ‘tokenized\_text’:

# Stemming

from gensim.parsing.porter import PorterStemmer

porter\_stemmer = PorterStemmer()

# Get the stemmed\_tokens

df['stemmed\_tokens'] = [[porter\_stemmer.stem(word) for word in tokens] for tokens in df['tokenized\_text'] ]

Text

Description automatically generated

Each unique word was identified by a unique id in the dictionary object. This was intended for creating representations of texts. A Bag-of-Words (BOW) corpus was created using this method which will be required for building the TF-IDF model. A dictionary was created by the list of words. This allowed the sentences to be converted to a list of words and then fed to the **corpora.Dictionary** as a parameter:

# Building a dictionary

from gensim import corpora

# Build the dictionary

mydict = corpora.Dictionary(df['stemmed\_tokens'])

print("Total unique words:")

print(len(mydict.token2id))

print("\nSample data from dictionary:")

i = 0

# Print top 4 (word, id) tuples

for key in mydict.token2id.keys():

print("Word: {} - ID: {} ".format(key, mydict.token2id[key]))

if i == 3:

break

i += 1

After building the dictionary, the report identified a total of 7156 unique words in the ‘stemmed\_tokens’ column.

Text

Description automatically generated

***NLP Task Implementation: (200)***

*Feature normalisation*

To allow the investigation to train a model on a separate data set to the validation data set, the data was split into train (70%) and test (30%) sets. The test data is what the model will predict the classes on, and it will be compared with the original labels to check the accuracy and other model performance metrics.

# Splitting into train and test sets

from sklearn.model\_selection import train\_test\_split

# Train Test Split Function

def split\_train\_test(df, test\_size=0.3, shuffle\_state=True):

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(df[['hotelId', 'reviewId', 'date', 'stars', 'review', 'stemmed\_tokens']],

df['Sentiment'],

shuffle=shuffle\_state,

test\_size=test\_size,

random\_state=15)

print("Value counts for Train sentiments")

print(Y\_train.value\_counts())

print("Value counts for Test sentiments")

print(Y\_test.value\_counts())

print(type(X\_train))

print(type(Y\_train))

X\_train = X\_train.reset\_index()

X\_test = X\_test.reset\_index()

Y\_train = Y\_train.to\_frame()

Y\_train = Y\_train.reset\_index()

Y\_test = Y\_test.to\_frame()

Y\_test = Y\_test.reset\_index()

print(X\_train.head())

return X\_train, X\_test, Y\_train, Y\_test

# Call the train\_test\_split

X\_train, X\_test, Y\_train, Y\_test = split\_train\_test(df)

As seen below, the train and test sets both have an even distribution of all three sentiment categories. This was crucial to prevent model bias.

Text

Description automatically generated

TF-IDF is computed by multiplying a local component like term frequency (TF) with a global componenent, that is, inverse document frequency (IDF) and optionally normalizing the result to unit length. The Gensim package was used to create the TF-IDF model. The TF-IDF model was trained and TF-IDF vectors were generated using the following code:

# Created TFIDF model

# Train the tfidf Model

from gensim.models import TfidfModel

# Make sure the dictionary is created from the previous block

# BOW corpus is required for tfidf model

corpus = [mydict.doc2bow(line) for line in df['stemmed\_tokens']]

# TF-IDF Model

tfidf\_model = TfidfModel(corpus)

# Generating TFIDF vectors

import gensim

import time

start\_time = time.time()

OUTPUT\_FOLDER = '/content/drive/My Drive/A3/'

tfidf\_filename = OUTPUT\_FOLDER + 'train\_review\_tfidf.csv'

# Storing the tfidf vectors for training data in a file

vocab\_len = len(mydict.token2id)

with open(tfidf\_filename, 'w+') as tfidf\_file:

for index, row in X\_train.iterrows():

doc = mydict.doc2bow(row['stemmed\_tokens'])

features = gensim.matutils.corpus2csc([tfidf\_model[doc]], num\_terms=vocab\_len).toarray()[:,0]

if index == 0:

header = ",".join(str(mydict[ele]) for ele in range(vocab\_len))

print(header)

print(tfidf\_model[doc])

tfidf\_file.write(header)

tfidf\_file.write("\n")

line1 = ",".join( [str(vector\_element) for vector\_element in features] )

tfidf\_file.write(line1)

tfidf\_file.write('\n')

print("Time taken to create tfidf for :" + str(time.time() - start\_time))

*ML algorithm*

A decision tree classification model was used to perform the sentiment classification. Decision tree classifier is a supervised machine learning algorithm for classification problems. The **scikit-learn** package was used for implementing the decision tree classifier. The fit function is was used to fit the input feature vectors against the sentiments in the train data set. The following code shows how this was achieved:

# Training sentiment classification model using tfidf vectors

from sklearn.tree import DecisionTreeClassifier

import time

start\_time = time.time()

# Read the TFIDF vectors

tfidf\_df = pd.read\_csv('/content/drive/My Drive/A3/train\_review\_tfidf.csv')

# Initialize the model

clf\_decision\_tfidf = DecisionTreeClassifier(random\_state=2)

# Fit the model

clf\_decision\_tfidf.fit(tfidf\_df, Y\_train['Sentiment'])

print("Time to taken to fit the TF-IDF as input for classifier: " + str(time.time() - start\_time))

*Hyper-parameters*

***NLP Task Output: (200)***

*Quality metrics*

The **feature\_importances\_** attribute was used to get the most important features of the model. The value for each feature was produced with the higher the value suggesting more importance. Getting the important features was implemented via the following code:

# Find out the most important features from the tfidf classification model

importances = list(clf\_decision\_tfidf.feature\_importances\_)

feature\_importances = [(feature, round(importance, 10)) for feature, importance in zip(tfidf\_df.columns, importances)]

feature\_importances = sorted(feature\_importances, key = lambda x: x[1], reverse = True)

# print(feature\_importances)

top\_i = 0

for pair in feature\_importances:

print('Variable: {:10} Importance: {}'.format(\*pair))

if top\_i == 10:

break

top\_i += 1

Text

Description automatically generated

*Summary of outputs*

As shown below, generates an overall accuracy of 56%. The model appears to be able to predict the ‘positive’ sentiment class more accurately. This may be due to the slightly higher number of observations.

# Testing the model

from sklearn.metrics import classification\_report

test\_features\_tfidf = []

import time

start\_time = time.time()

for index, row in X\_test.iterrows():

doc = mydict.doc2bow(row['stemmed\_tokens'])

features = gensim.matutils.corpus2csc([tfidf\_model[doc]], num\_terms=vocab\_len).toarray()[:,0]

test\_features\_tfidf.append(features)

test\_predictions\_tfidf = clf\_decision\_tfidf.predict(test\_features\_tfidf)

print(classification\_report(Y\_test['Sentiment'],test\_predictions\_tfidf))

print("Time taken to predict using TF-IDF:" + str(time.time() - start\_time))

*Calendar

Description automatically generated*

*Visualise outputs*

*NLP task alignment with issue*