LONDON'S GLOBAL UNIVERSITY

Hotel Reviews Analyser System Incorporating Sentiment Analysis

Candidate Number: NSZW1[1]

BSc Computer Science

Supervisor: Anthony Hunter

Submission date: Day Month Year

Abstract

Over the past few decades, with the increasing use of internet and advancements in online applications, people's lifestyles have shifted into the new habit of "doing things online". For travelers and holiday makers, being able to plan out their route and select the appropriate hotels before travelling is crucial in their preparation stage. However, choosing the best hotels can be a daunting task. The amount of information available online may not be accurate, travellers tend to go through the reviews to see what are people's experiences were with this hotel. However, there could be hundreds of reviews for each hotel, and it becomes time consuming to read through each. Thus the system developed in this project will use an opinion-based sentiment analysis to examine the reviews related to every hotel within the database, extracting key information to help users to understand why a particular hotel has been given a good or bad rating. Furthermore, this approach combines lexical analysis, syntax analysis and supervised machine learning algorithms to understand the sentiment of the reviews.

Contents

Chapter 1

Introduction

In this chapter, the report will begin by outlining the motivations of this project. Then specify the aims that the system want to achieve, followed by an overview of how the project has been carried out. Lastly, a brief overview of the structure of the rest of the report.

1.1 Motivations

The convenience and variety of public transports available today, has made travelling easy and affordable. Around the globe, demand for good hotels is on a constant rise for the holiday makers and business travellers. Each hotel seeking individual has different needs, it is difficult for a single hotel to cater for all the different features demanded by the travellers. A highly rated hotel may be enjoyed by a group of people, but at the same time disliked by others. In modern hotel booking websites, many indicators and descriptions were given, in order for a regular individual to make a decision on which hotel to choose. Putting aside the destinations, two important factors that tends to be considered the most are the average score of the hotel and the huge number of customer reviews, which are fully loaded with useful information.

The average scores given can only provide a general idea as to whether the majority of customers think this hotel is good or bad, specific details as to why they think that way were not inferable. Thus, to understand the strengths and weaknesses of the hotels better, it is necessary to perform text analysis on the customer reviews. However, the vast number of reviews are laborious to go through manually, hence the motivation for this project. This report describes the stages of building a system that performs text analysis on a dataset of customer reviews, extracting the key features and putting together a summary report of the hotels. The reports produced can be used by hotel managers to help them understand what are the area of improvements in the hotel, or what is good about their services. At the same time, the report may also be useful to customers that want to book a hotel, because immediately, they can see a summary of what is the experience of other customers. This enables the potential customer to leverage for themselves if the less desirable aspects of the hotel are endurable, or if the hotel is really attractive to them.

1.2 Aims

The main objective of this project is to use the "Hotel_Reviews.csv" dataset from the **kaggle** website [1], and build a system that can generate a report for each hotel, outlining the key findings

extracted from both the numerical data such as ratings, and textual data involving user reviews. Thus provide a more detailed insight about each hotel.

The system also tries to produce a friendly user interface that is straightforward to use by the users. The user may be able to select any hotel that they are interested in from a list of hotels available in the dataset. When a selection is made, a request for the report about the selected hotel will be sent to the analyser system's server, which in turn has to extract relevant information from the files within the database, to put together a concise report for the user.

The system also aims to carry out sentiment analysis on the reviews, to make sure that the positive and negative aspects about the hotel within the report are true. This is also a process of validation, because the system has to validate the supposedly "positive" or "negative" reviews in the dataset against the sentiment analyser of the system.

1.3 Project Overview

This project can be divided into three parts: Front-end, Back-end and Data Analyser.

1.3.1 Data Analyser

The Data Analyser is the intelligent part of the system. It incorporates various machine learning algorithms and natural language processing techniques to analyse the user reviews within the original dataset. The main function of the Data Analyser is to breakdown the original dataset into smaller results tables, allowing the back-end server to have quick access to the contents it needs.

Another key stage in this component is to perform sentiment analysis on the reviews. This has to be done before the positive and negative feature extractions can take place. The majority of this report will be dedicated to explaining the steps taken and technical implementations of this part of the system.

1.3.2 Front-end

The front-end is a user facing dynamic website that begins by displaying a list of hotels for the users to select from. Once the user has selected a hotel that they wish to see, the site will pass on the request to the back-end server to fetch for the relevant report. The returned results are then rendered in the user's browser.

1.3.3 Back-end

Upon receiving a request from the front-end website, the back-end server will process what the targeted hotel is and goes through different result tables within the database to retrieve a set of relevant data. The data obtained are then passed forward to the front-end and be rendered into the format of a report.

1.4 Overview of report structure

Chapter 2: a general background of the problem that this project is trying to solve is explained within the context of online hotel bookings. Also a list of tools and development softwares used by this project is presented.

Chapter 3: the report will start by introducing the architecture of this hotel review analyser system. More detailed implementations of the other components in the system are also explained.

Chapter 4: this chapter focuses on the testing results and evaluation of the methods used in the development stages. Especially the performance comparison between various machine learning algorithms.

Chapter 5: a summary of the achievements and challenges faced are discussed.

Chapter 2

Context

One of the early versions of a hotel recommender system is to take simple averages of all the user ratings associated with each hotel. However, the shortcomings of this method is that only hotels with ratings close to 5-stars, which could be very expensive in the first place, are always ranked first. This results in less variations and other hotels which may be more suited to the user are being pushed to the back, because the recommender system automatically assumes that users only want 5-star rated hotels. This method is appropriate in making quick suggestions. The overall service provided by these hotels are generally satisfactory. Once the users click into a particular hotel, they can read more about the descriptions of the hotels.

However, all the textual information that describes the hotel were carefully drafted by the hotel owners themselves, which are subjective and biased towards the good aspects about the hotel. In order to get a full picture of whether the hotel is good or not, users had to read the reviews provided by other customers who have been to the hotel. Their experiences are more useful and less biased than the official descriptions. Therefore, this system will perform text analysis on all the reviews of each hotel, and summarise the key findings into a short report format that is more objective and fair.

There is a need for this type of review analysing system because for every hotel on the booking platforms, there could be hundreds of different reviews. It takes time and effort for any person to read through each review about a hotel and make a conclusion on what they think are the best and worst features. The difficulty of this task increases with more hotels joining the booking platform, which is unfit for humans to manually process them. However, this is a perfect task for machines. First of all the reviews were written in English, the wordings will therefore be written in a way that abide by the English grammars. Secondly, some reviews may be repetitive because the reviewers may have shared similar experiences during their stay. Thirdly there already exist a vast number of reviews online and new reviews are constantly generated. These three characteristics about hotel reviews has made it a perfect problem to be solved by techniques in natural language processing, and provides a large number of training data for machine learning algorithms.

2.1 Tools

Below is a list of software tools and programming languages used in the system:

- Python - For main development of system: `https://www.python.org/`
- HTML - Front-end design: `https://html.com/`
- Scikit-learn - Python based machine learning library. Used for generating classifiers that predict sentiments: `https://scikit-learn.org/stable/`
- NLTK - Python based natural language processing toolkit. Used in text analysis, information extraction and preprocessing data: `https://www.nltk.org/`
- Django - Python based tool for building web applications involving front-end and back-end components: `https://www.djangoproject.com/`

Chapter 3

Design and Implementation

This system is built based on the analysis of the 'Hotel_Reviews.csv' dataset, obtained from the **kaggle** website [1]. According to the uploader, this is a dataset crawled from `Booking.com`, where all the data were made publicly available. This dataset is suitable for the development of this project because it contains a combination of numerical and textual data, which is good for report generation.

3.1 System architecture

This analyser system works by first analysing the numerical and textual data, storing the key findings into different categories of smaller files. The files are then queried by the back-end of the system to extract the relevant information needed for producing the report of the hotels. The textual data from the dataset, mainly the customer reviews, are processed using natural language processing packages. A series of text analysis and feature extractions are performed to obtain the key aspects of the hotel that is being talked about in the review. For each hotel, the system is aiming to summarise a list of its best features, which allows the users to quickly see what are the strong points about this hotel. The reviews are also used for sentiment analysis using machine learning techniques, which helps the system to determine whether each review is truly positive or negative. The high level overview of the hotel analyser system's architecture is shown in Figure 3.1.

3.2 Text analysis

One of the important aspects in the report produced by this system is to provide a short list of the most liked and disliked features of the target hotel. For each hotel in the dataset, the system examines its corresponding positive and negative reviews and performs feature extraction.

In order to extract representative features that are meaningful to the users, the reviews were preprocessed using the NLTK (Natural Language Tool Kit). The following steps of text analysis have been carried out:

1. Lexical Analysis
2. Syntactic Analysis
3. Semantic Analysis

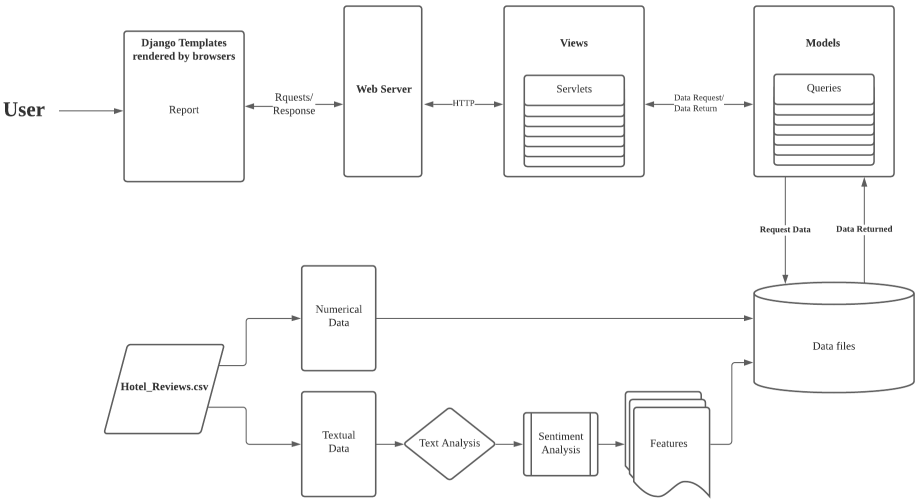

Figure 3.1: System Architecture

3.2.1 Lexical Analysis

In NLP (Natural Language Processing), lexical analysis is the study of texts at the words level. It examines the lexical meaning of each individual words. For example, the word "python" is a noun and "pythonic" is an adjective. In other words, lexical analysis is the process of taking a streams of characters as input and generate a list of tokens together with its respective lexical meanings as the output.

Tokenisation

The text reviews in the 'Hotel_Reviews.csv' dataset have already undergone some degree of pre-processing where all the punctuations have been removed. Each review is given in the form of a single long string where several sentences are concatenated together. The string is tokenised using the NLTK package's `nltk.tokenize.word_tokenize()` function, stored as a list of word tokens. An example of tokenisation is shown in Table 3.1.

Sentence	The room was big enough and the bed is good
Tokenised	['the', 'room', 'was', 'big', 'enough', 'and', 'the', 'bed', 'is', 'good']

Table 3.1: Example of tokenisation.

POS-tagging

In order to detect and correct spelling mistakes in the reviews, each token is passed through a statistical spell checker obtained from pyspellchecker [2]. The misspelled tokens are replaced with its most likely correct version. After that, the tokens are labelled with their parts of speech, called part-of-speech tagging using the NLTK package's `nltk.pos_tag()` function. This function considers each token's definition and classifies its appropriate POS-tag according to the context. An example of part-of-speech tagging is shown in Table 3.2.

Tokens	The room was big enough and the bed is good
POS-tagged	[('the', 'DT'), ('room', 'NN'), ('was', 'VBD'), ('big', 'JJ'), ('enough', 'RB'), ('and', 'CC'), ('the', 'DT'), ('bed', 'NN'), ('is', 'VBZ'), ('good', 'JJ')]

Table 3.2: Example of part-of-speech tagging.

The implementation of lexical analysis can be found in Code A.1. The list of POS-tags and its respective meaning is shown in Code A.2.

3.2.2 Syntactic Analysis

Syntactic analysis, as the name suggests is focused on the syntactic level of linguistic processing. This process makes use of the tokens that are POS-tagged in the lexical analysis and attempts to group words into short phrases. This is important because short phrases hold more meaning than just single words. In order to understand what is the main subject being talked about in a given review, the extraction of noun phrases becomes the key component that helps with finding the best features of the target hotel and feature extractions for sentiment analysis at a later stage.

Chunking

The technical term for extracting short phrases are also called chunking, where small group of words from the sentence are taken as"chunks". Therefore, determining which chunks of the sentence are going to be useful becomes a vital problem. This is achieved by parsing the POS-tagged tokens with predefined grammars that incorporates regular expressions. Taking the POS-tagged sentence in Table A.2 as an example, the aim is to extract chunks such as "room was big" and "bed is good", because they are useful for the system to perform feature extractions.

Grammar

Thus to capture these phrases, the system uses different grammars to try and match the individual word's POS-tags. As seen in Table A.2, the words "room" and "bed" both have the 'NN' tag, which stands for a noun (see Code A.2). The words "was" and "is" are labelled as 'VBD' (past tense verb) and 'VBZ' (present tense verb) respectively, followed by "big" and "good" which are both 'JJ' adjectives. Example of which grammars can capture these POS-tags are shown in Table 3.3. The full grammars used by the system can be found in Code A.3. In a way, chunking has already removed a lot of redundant data from the sentence structure, reducing dimensionality of the data to save computation power.

Grammar	Chunking
⟨ NN⟩⟨VBD⟩⟨JJ⟩	('room', 'NN'), ('was', 'VBD'), ('big', 'JJ')
⟨NN⟩⟨VBZ⟩⟨JJ⟩	('bed', 'NN'), ('is', 'VBZ'), ('good', 'JJ')

Table 3.3: Example grammars for chunking.

Lemmatization

In English, there are many words that looks similar which are derived from a single word, called derived words. The word's structure may change depending on its usage and the part of speech which it is in, in a given context. A language that has derived words in its grammar is also called an inflected language. "In linguistic morphology, inflection (or inflexion) is a process of word formation,[1] in which a word is modified to express different grammatical categories such as tense, case, voice [...] An inflection expresses grammatical categories with affixation (such as prefix, suffix, infix, circumfix, and transfix)" [3]. Thus the derived words in the tokens can be normalised by reducing them back to their root forms. See example in Table 3.4. This system makes use of

the NLTK package's `nltk.stem.wordnet.WordNetLemmatizer.lemmatize()` function to obtain the root words called lemmas, which are real words that belongs to the English language. An alternative method of normalisation is stemming, in which the reduced root forms might not be real words, thus losing the original meanings. Therefore only lemmatization is being used in this system.

Derived words	Root word
rooms	room
room's	room

Table 3.4: Example of lemmatization.

Lemmatization is only applied after the stream of tokens has been POS-tagged, so that their original POS-tags are not affected by the modification of the words. The normalisation of tokens are also useful in helping with dimensionality reduction other variations of the same lemma can be discarded.

Stop words

In each review, there are many words which do not have a lot of meaning, such as "is", "was", "the", "are", "in", they are commonly referred to as stop words in NLP. To further reduce the data sizes, these words are also removed from the chunks extracted previously. Currently, there is not a global standard in defining which words can be enlisted as a stop words because each NLP tasks is different. Hence in this project, the system uses the list of stop words provided by the NLTK package.

After going through the lexical and syntactic analysis, the example sentence in Table 3.1 is reduced into the format shown in Table 3.5.

Sentence	The room was big enough and the bed is good
Processed	[('room', 'NN'),('big', 'JJ')], [('bed', 'NN'),('good', 'JJ')]

Table 3.5: Sentence after undergone Lexical and Syntactic analysis.

3.2.3 Semantic analysis

Unlike the lexical and syntactic analysis, which follows a set of standardised procedures to achieve the results, semantic analysis is less constraint in how it proceeds and open to more interpretations. For humans, it is easy to make sense of a sentence because the brains can automatically recognise the meaning of individual words and the context in which they are used in. Machines on the other hand must be trained to understand the meanings of the text, which is the objective of semantic analysis.

Hotel features extraction

Building on top of the refined data obtained from the text analysis stages, the system is able to extract more meaningful data from the reviews, in particular the most liked and disliked features relating to the target hotel.

Positive features: Given a target hotel, its positive reviews from the dataset were fed into the the text analyser. For each review, a list of refined "chunks" were extracted. Because of the way that the grammars were defined in Code A.3, each chunk is a noun phrase. The noun is then taken as the feature with the other values stored under it as a list, forming a feature-value pair. An example output of the semantic analyser is shown in Table 3.6.

Feature	**Values**
room	['big', 'clean', 'nice']
bed	['good', 'comfortable', 'large']
staff	['friendly', 'helpful', 'nice']

Table 3.6: Semantic analysis output.

After examining each review, the values that correspond to the same features are merged together. The features are then ranked based on the number of values it has. In other words, the features that have been mentioned the most are higher ranked. Later on in the report generation stage, the top 5 features are presented. This is based on the assumption that reviewers tend to comment on the features that they genuinely liked or enjoyed. If something is mentioned more often by a collective, it infers that the hotel has done well in that particular aspect.

Negative features: The same procedures are also applied to the negative reviews to extract the top 5 most disliked features about the hotel. This is shown as potential areas of improvement for the hotel.

3.3 Sentiment analysis

Sentiment analysis is a sub branch of NLP. It is also called opinion mining, a process to determine whether a piece of text is positive, negative or neutral. This is beneficial to businesses because they could be interested in knowing what is the sentiment of the customer feedback on their products.

3.3.1 Motivation

For this project, it may first seem that sentiment analysis is not necessary, because the reviews are already classified into positive and negative columns. However, after close examination of the reviews, it turns out that when writing long reviews, users have a high tendency to provide a mixed feedback. For example in a negative review, the user also mentions something good about the hotel. This will affect the best feature selection process previously because the values captured within the "chunks" may not strictly be positive or negative. Hence the motivation for using sentiment analysis. Each sentence within the review will be passed into the sentiment classifier and be put into positive or negative columns. This approach mines deeper into the text reviews, making sure that the positive sentences hidden in a negative review is taken out and put into the positive section and vice versa for negative sentences. This reclassifying process reduces the number of mixed sentences and improves the system's ability to select the most liked and disliked features.

3.3.2 Supervised machine learning problem

In order to build a sentiment classifier, the system has to be trained using Machine Learning (ML). The main tool used for this section is the Scikit-learn package, which provides many different tools for solving ML related problems. Before diving into the detailed implementations it is important to understand the type of problem at hand so that the appropriate algorithms can be used.

The duty of the sentiment analyser is to assign "positive" or "negative" labels to the reviews. This is a binary classification problem, a sub-field of supervised ML problem, which requires the training samples to be labelled so that the ML algorithm can produce a model that maps the new reviews to a particular label.

3.3.3 Feature extraction

One of the key aspect in producing reliable ML models that generalises well on unseen data is the quality of the training samples. Unlike numerical data, textual data cannot be analysed by ML algorithms directly, they need to be converted to some sort of numerical representation first. To begin the process, the list of values obtained from the semantic analyser stage (such as in Table 3.6) are used as the base input. The results obtained from text analysis are appropriate because they have already undergone many layers of cleaning. Irrelevant words such as the stop words and even the supposedly good and bad features which are just nouns are discarded. The values left are the key words which by themselves generally reflect a certain sentiment. For example, values such as "nice", "friendly" and "helpful" are words with positive sentiment.

Bag of words

A common approach in transforming textual data into a numerical representation is by using bag-of-words (BOW). The list of "values" or tokens from every chunk are combined into a single feature vector. Then from this feature vector, the number of times each token appears in each review is counted. This produces a document term matrix, where each row is a review (also the document) and each column is the different values (the terms), which is the first numerical representation of the textual data. This is achieved by using the `sklearn.feature_extraction.text.CountVectorizer()` function from the Scikit-learn package. An example of how the document term matrix is generated is shown in Table 3.7.

Example reviews:

1. "The room was big enough and the bed is good"
2. "Comfy bed good location"
3. "Bed was extremely comfy"
4. "The room is dirty and the windows were broken"

TF-IDF

For this section, the terms *document* and *review* will be used interchangeably. The bag of words approach works fine for converting text to numbers. However, it has one drawback. It assigns a score to a word based on its occurrence in a particular document, but it does not take into account

	big	good	comfy	dirty	broke
Review 1	1	1	0	0	0
Review 2	0	1	1	0	0
Review 3	0	0	1	0	0
Review 4	0	0	0	1	1

Table 3.7: Example of document term matrix derived from the reviews.

the fact that the word might also be having a high frequency of occurrence in other documents. TF-IDF resolves this issue by multiplying the term frequency of a word by the inverse document frequency. The TF stands for "Term Frequency" while IDF stands for "Inverse Document Frequency". It weighs the importance of words in a document, "then documents with similar, relevant words will have similar vectors, which is what we are looking for in a machine learning algorithm."[4]

In this project, the TF-IDF is used to help with feature extraction. Words with high TF-IDF scores will be extracted to form feature vectors that helps the ML algorithms to learn how to classify positive and negative sentiments.

Notations:

- t is the given term in a document.
- d is a document.
- n is the total number of documents in the document set.

The Term Frequency is calculated as:

$$tf(t,d) = \log\left(1 + f(t,d)\right)$$

Here: $f(t,d)$ is the term frequency in document d.

The Inverse Document Frequency is calculated as:

$$idf(t,n) = \log\left(\frac{1+n}{1+df(t)}\right) + 1$$

Here: $df(t)$ is the document frequency of t, which is the number of documents in the document set that contain the term t.

The final TF-IDF score is calculated as:

$$tf\text{-}idf(t,d,n) = tf(t,d) * idf(t,n)$$

The implementation of Tf-IDF is shown in Code A.4 .

3.3.4 Training sentiment classifier

After undergoing the TF-IDF transformation, the document term matrix can be considered as being normalised. The extremely repetitive words are given a lower weight so that their presence in the training data are less dominating over the other word tokens.

The normalised document term matrix is now considered as the feature vectors, which will be used as the training dataset. The next step is fed the training set into different ML algorithms that can generate models for binary classification, called classifier. A trained model will be able to classify an input sentence into either the positive sentiment or the negative sentiment categories.

The approach adopted in this system is to use not one, but a collective of classifiers that forms a voting system in determining sentiment of sentences. More details on the exact implementations are clearly explained in the next chapter.

3.4 Front-end and Back-end

In order to provide the users of this system with an easy to use interface, this project makes use of the Django web framework that follows a Model-View-Template pattern. The key source code files for implementing this web framework are listed below:

- `urls.py`
- `views.py`
- `models.py`
- `Templates`

Figure 3.2 illustrates the how a user request is process by the Django system.

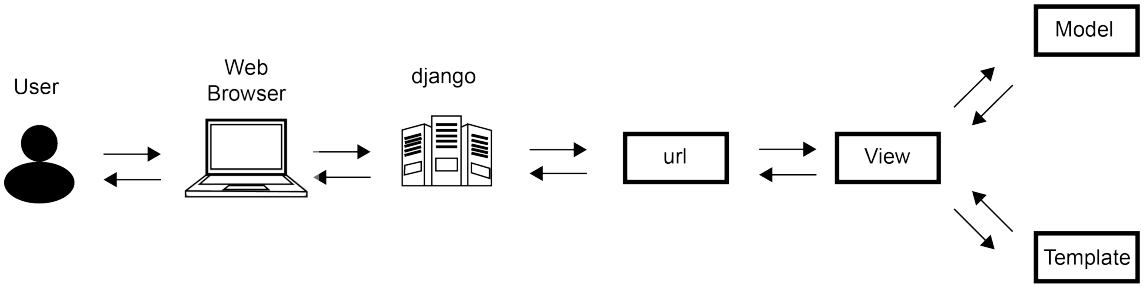

Figure 3.2: Django webframework architecture.

3.4.1 Django web framework

urls.py

This source file is relatively straightforward to follow, it contains a list of allowable URL patterns that joins the various components of the system together. When the Django server receives an incoming request, it checks whether this request is coming from a recognizable website path defined by `urls.py`. If the request is valid, the URL pattern will decide which function within the `views.py` to call and pass on the requested parameter to that function.

views.py

`views.py` holds the core functions that can be used by the back-end system. It receives the request and the corresponding parameters, more specifically the name of the hotel requested by the user,

from the `urls.py` script. In this script, the main function being called will be the `report()` function. This is the function for preparing the report because it knows which data is required. However, `views.py` does not talk directly to the database, instead, it calls a series of functions in `models.py` to fetch the data that it needs. Then it packages up a various data into a Python dictionary and sent it off to a specific template in the `Templates` folder.

models.py

`models.py` acts as the waiter serving the `views.report()` function. It receives instruction on which data is needed, then goes through the database files to extract that specific data and passing it back to `views.report()`. This is to separate the tasks between the different scripts so that the course codes are more manageable.

Templates

Unlike the first three Python scripts, the `Templates` is a folder consisting of `.html` scripts. Each `.html` script defines the layout of the web page that is going to be rendered in the user's web browser. Namely, `report.html` will render the requested report and `index.html` will render the home page with the list of hotel names.

After the `views.report()` function has acquired the necessary data for a report, it tells the Django `controller` that the packaged data should be rendered using the `report.html` script. The final rendered report page will be sent back to the user and executed by his or her browser. The reason that the controller's working mechanisms are not discussed in this report is because the controller aspect is managed by the Django framework internally.

3.4.2 User manual

1. In the command line, from the project's home directory, navigate to the `Hotel_report` folder located inside the `Django_Project` folder, which should be the directory containing the `manage.py` script.
2. execute command: `python3 manage.py runserver`
3. Open a web browser and go into `http://127.0.0.1:8000/`

(1) Having followed the steps above, the browser should boot up the home page as shown in Figure 3.3. This displays the list of hotels available from the dataset.
(2) Select any hotel available from the list to see the report. Figure 3.4.
(3) The report includes the following list of information:

- Average rating
- Location
- Nationality of customers who generally enjoyed the stay. This takes the average of all the reviewers from the same nationality.
- Nationality of customers who generally disliked the stay.
- Best features
- Worst features

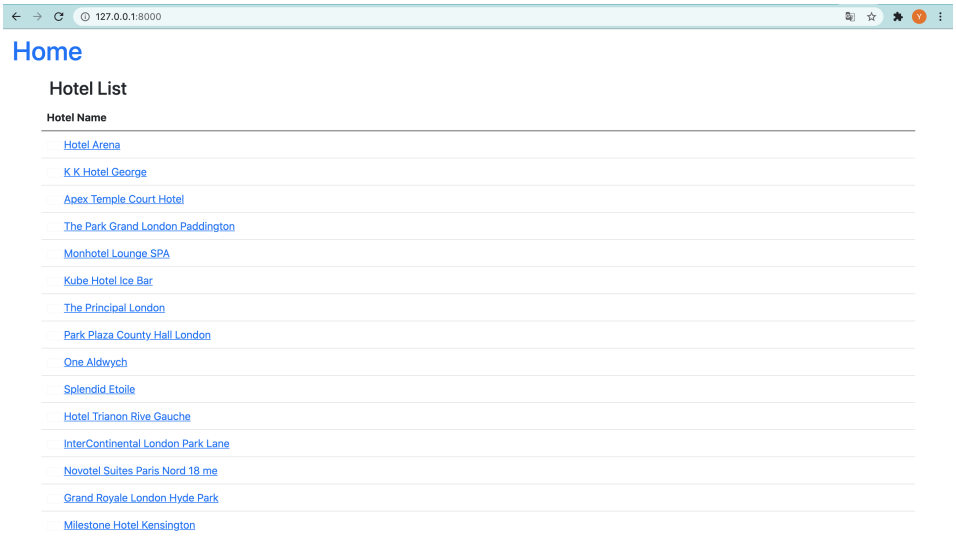

Figure 3.3: Home page

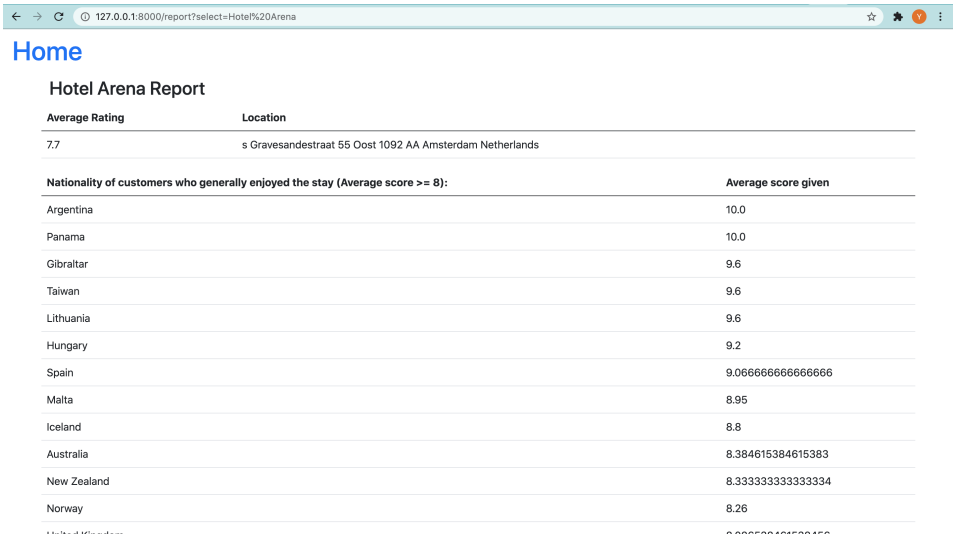

Figure 3.4: Report page of “Hotel Arena”, top section.

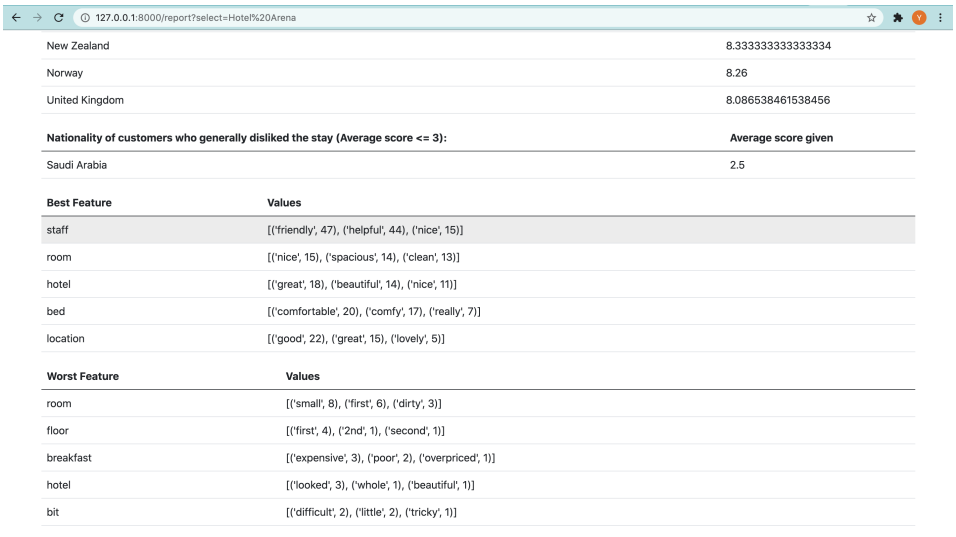

Figure 3.5: Report page of “Hotel Arena”, bottom section.

Chapter 4

Testing and Results Evaluation

Machine learning problems are challenging to solve, the best solution is rarely achieved in the first go. In order to find the best model and its combination of parameters, performance analysis is required. This involves examining the learning curves and compare prediction accuracies to eliminate problems such as overfitting and underfitting. In this chapter, the focus is on how model selection techniques such as cross validation and evaluation metrics are used to help obtain the best sentiment classifier for this system. The main tool used here is the Scikit-learn package. The following sections will start by introducing the evaluation metrics used, then move on to the testing of different ML models.

4.1 Evaluation metrics

In order to evaluate the performance of the models, the model's prediction accuracy and the precision and recall values are examined.

Notes: In the evaluation metrics, the output (0) represents the "negative" label and output of (1) represents the "positive" label.

Prediction accuracy

The prediction accuracy is straightforward. This system uses `sklearn.metrics.accuracy_score()` function from the Scikit-learn package to calculate the percentage of the test samples' labels that have been predicted correctly.

The equation for accuracy prediction is:

$$\text{accuracy}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} 1\left(\hat{y}_i = y_i\right)$$

where: $\hat{y}_i$ is the predicted sentiment label of the *i-th* sample and y_i is the true label, n_{samples} is the total number of samples tested.

Precision, Recall

Notations:

- TP stands for True Positive, this is an outcome where the model correctly predicts the positive class.
- TN stands for True Negative, this is an outcome where the model correctly predicts the negative class.
- FP stands for False Positive, an outcome where the model incorrectly predicts the positive class.
- FN stands for False Negative, an outcome where the model incorrectly predicts the negative class.

Precision: This is about calculating what proportion of the positive class predictions are actually correct. The formula of calculation is shown below:

$$\text{Precision} = \frac{TP}{TP + FP}$$

Recall: This is about calculating what proportion of the actual positive samples are classified correctly. The formula of calculation is shown below:

$$\text{Recall} = \frac{TP}{TP + FN}$$

In Table 4.1 a confusion matrix of the relationships between precision and recall are shown more clearly.

		Actual label	
		True	False
Predicted label	True	True Positive	False Positive
	False	False Negative	True Negative

Table 4.1: Confusion matrix.

When evaluating the effectiveness of a classifier, precision and recall are examined together. However, in most case, both values are often in competition with each other. Modifying the model to Improve precision may cause recall to decrease and vice versa. But for this system, obtaining a good precision value is slightly more important, because the performance of selecting the best feature and value pairs described earlier, relies on the classifier to have a high confidence in its classifying process.

F1-score

The F1-score is another measure of the accuracy performance of a binary classification model. It is defined as the harmonic mean of precision and recall, calculated as follows:

$$\mathrm{F}_1 = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

For instance, a model may have a very high precision with a low recall value, which could give a high accuracy score. It may seem that the model is working well, however in reality, this could be a case

of overfitting. Using F1-score, the value calculated will be much lower than the normal accuracy score, because it punishes extreme values in either precision or recall. Thus a high F1-score means a more balanced precision and recall values.

4.2 Reasoning of model performance

The evaluation metrics are very intuitive and straightforward in understanding how well a model has performed. However, that is not the full story, because they are simply the end results. It is essential to be able to understand the reasons that led to these results and finding the flaws in the model. To clarify, this is not about finding flaws in the ML algorithm itself, it is about finding the flaws in the generated model after undergoing training. In particular, there are two main problems faced by a model: underfitting and overfitting.

Training and testing sets

The training samples are represented as: $\mathcal{S} = \left\{\left(\mathbf{x}^{(i)}, y^{(i)}\right)\right\}_{i=1}^{n}$, where $\mathbf{x}^{(i)}$ is the feature vector extracted for each review. $y^{(i)}$ is the label of the corresponding review, which takes the values 0 (negative) or 1 (positive).

The entire sample set were then split up with 80% used as the training set and 20% used as the testing set. The training set data are used to fit the ML algorithms to generate models. The trained models are then used to predict the mappings on the testing set and evaluation metrics can then be calculated to measure the performance of the model.

Underfitting

In simple terms, underfitting, also referred to as high bias, occurs when the model has poor performance on the training set and poor generalization on the test set. This could be due to the model being too simple, it is highly biased towards certain basic assumptions that it fails to capture the other key characteristics of the positive and negative sentiment reviews. This can be overcome by increasing model complexity, enriching its representation of the training data. In other cases, it could also mean that the wrong ML algorithm has been used to generate the model.

Overfitting

Overfitting, also referred to as high variance, occurs when the model has good performance on the training set and poor generalization on the test set. This could be due to the model being overly complex, it has tried too hard in learning the key features that appears in each target class in the training set data. This allows a very small room of error. Thus when tested against the previously unseen testing set, data that may have slight differences from what it has seen in training will become unrecognisable for the model and it generalizes poorly on these new data. This can be resolved by reducing the model complexity, taking away subsets of the feature vector that the model uses. Or increase the sample space used in the training process.

4.3 Decision Tree Classifier

The first ML algorithm tested in the development of this analyser system is the Decision Tree Classifier (DTC) algorithm. DTC has a tree structure consisting of nodes and edges, starting from

a root node at the top, branching downward to two child nodes. Each node will continue to have two new child nodes until the leaf nodes have been reached. There are two types of nodes, one type is called a decision node, which is used to make a decision in how to split the data. The second type is the leaf node, which represent a final outcome. In this training exercise, the leaf node will represent either "positive" or "negative", because the objective is to determine the sentiment of a sentence after traversing through the series of decision making processes in the tree.

Criterions for decision node

At each decision node in the DTC, the algorithm calculates how well a given feature separates the target classes. In the `sklearn.tree.DecisionTreeClassifier()` function, there are two criterion available for measuring the quality of split, namely by Gini Impurity and Information Gain. More readings on their working mechanisms can be found **here**.

Tree depth

Another key parameter which can be configured is the depth of the tree. For a small depth such as 2, the algorithm will converge quickly, because only 3 features will need to be selected before reaching the leaf node. An example tree with depth set to 2 is shown in Figure 4.1.

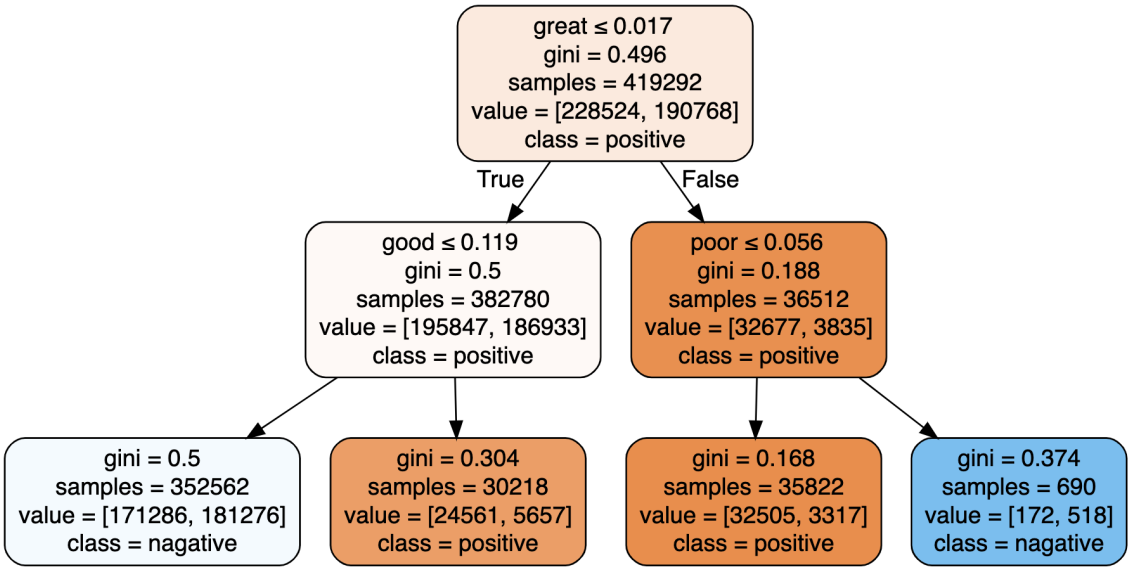

Figure 4.1: DTC with depths of 2.

As shown in Figure 4.1, the root node takes the word "great" as the best feature. It also shows that the term has a higher probability of being labelled as positive. Hence, when predicting the sentiment of a given sentence, DCT first calculates a weighting of the word "great", corresponding to the sentence. If the condition "great ≤ 0.017" is not satisfied, go down to the right branch and check if the condition "poor ≤ 0.056" is satisfied. If the answer is true, then traverse to the left, reaching a positive leaf node.

The principle which the DTC follows is therefore very intuitive. The deeper the tree is, the more words will used to from the decision nodes that sits between the root node and the leaf nodes. This will allow the model to capture more variations of the sentence being tested and increase accuracy.

Testing

Using the DTC with depth 2 as a starting point, the performance of the model is shown in Table 4.2.

	precision	**recall**	**f1-score**
0	0.51	0.95	0.67
1	0.87	0.25	0.39
Accuracy	0.5688446237943963		

Table 4.2: Accuracy and Precision Recall of DTC with depth 2.

From the evaluation metrics in Table 4.2, this model has a very high precision in classifying the positive class, so it is confident in choosing which sample has a positive sentiment. But the model also has a low recall, meaning that it tends to miss samples belonging to the positive class, which is why the F1-score is low. Hence the overall accuracy achieved is only around 57%, not generalizing well on new data.

Investigating depth

To remedy this problem, the different depth can be used when training the DTC and choose the depth with the highest accuracy.

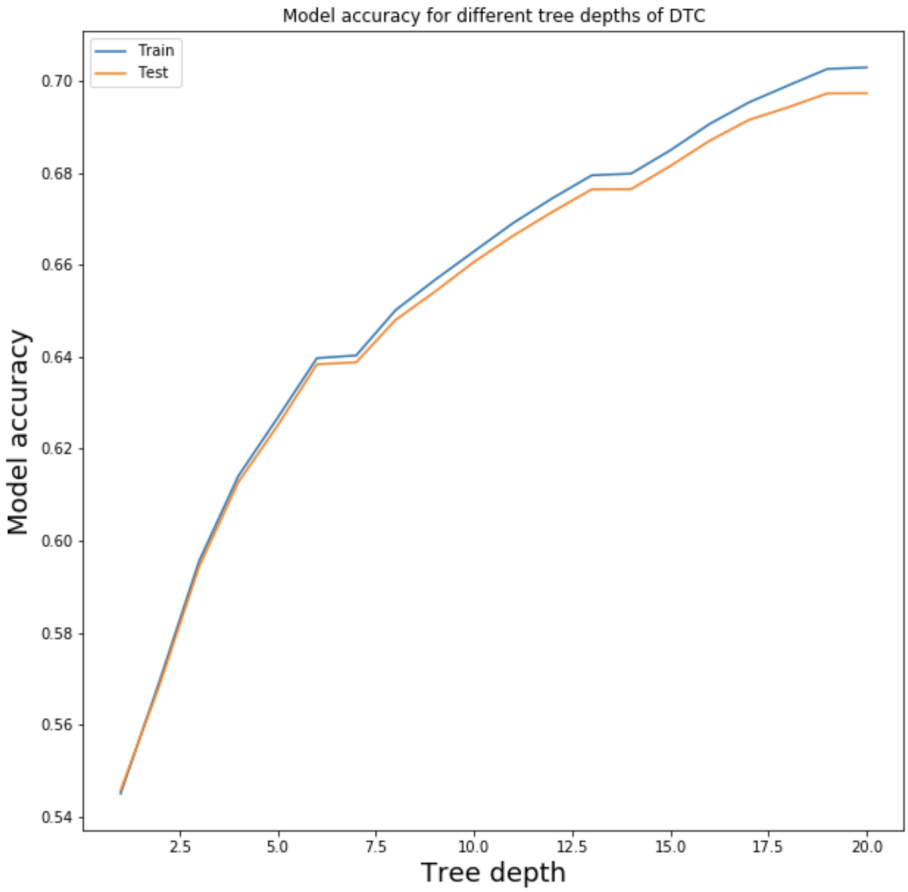

Figure 4.2: Model accuracy curve.

Figure 4.2 shows the accuracy scores of tree depths ranging from 1 to 20. From the graph, as the

tree depth used increases, the model's accuracy on classifying both the training and testing set also increases. The exact figures of some of the depths are shown in Table 4.3.

Tree depth	1	5	10	15	20
Accuracy	0.5457	0.6251	0.6606	0.6815	0.6973

Table 4.3: DTC prediction accuracies on test set with different tree depths.

	precision	**recall**	**f1-score**
0	0.61	0.93	0.74
1	0.90	0.50	0.64
Accuracy	0.6973374164067047		

Table 4.4: Accuracy and Precision Recall of DTC with depth 20.

At this point, the model generated at depths of 20 shows the best result. The evaluation metrics of this particular model is displayed in Table 4.4. Increasing depth to 20 has improved the model's accuracy by around 15%. The precision for identifying positive sentiment has now reached 90%, but the recall value is still a lot lower. Also the precision for identifying negative sentiment is much less than that of positive sentiment. This suggests that the model is only good at identifying the obvious positive sentiment reviews. It's less efficient in understanding the negative sentiments. Therefore more tuning is required.

Experiments with other tree depths have been conducted. The accuracy graph for depths values ranging from 20 to 5000 can be found in Figure 4.3, Table 4.5 has the summary of accuracies.

Tree depth	20	1000	2000	3000	4000	5000
Accuracy	0.6973	0.7807	0.7827	0.7835	0.7835	0.7834

Table 4.5: DTC prediction accuracies on test set with different tree depths.

According to the graph in Figure 4.3, tree depth of 4000 has achieved the highest accuracy. However, back at depth 1000, a similar accuracy of 78% has already been accomplished. Thus it is best to compare the evaluation metrics at both of these tree depths.

	precision	**recall**	**f1-score**
0	0.73	0.83	0.77
1	0.84	0.74	0.79
Accuracy	0.7807		

Table 4.6: Accuracy and Precision Recall of DTC with depth 1000.

	precision	**recall**	**f1-score**
0	0.74	0.81	0.77
1	0.83	0.76	0.79
Accuracy	0.7835		

Table 4.7: Accuracy and Precision Recall of DTC with depth 4000.

Despite having more depths, the model 4000, which in theory has a richer representation of the training data by using far more decision nodes did not perform any better than that of model

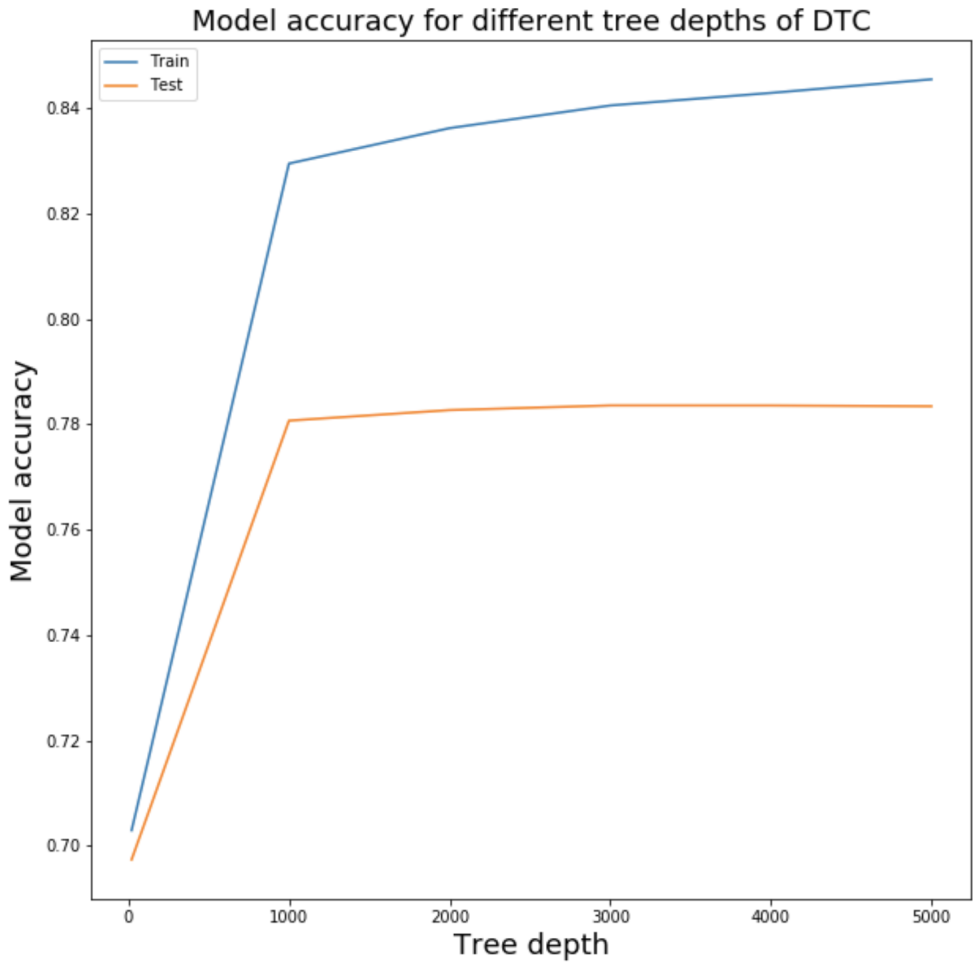

Figure 4.3: Model accuracy curve with different depth, using Gini Impurity criterion.

1000. The precision and recall values of classifying both the positive and negative sentiments are also relatively high, with F1-score similar to the actual accuracy. Therefore, the DTC model with depths of 1000 is fairly reliable and consistent.

Investigating criterion

The above model implementations has been achieved using the "Gini Impurity" criterion. The generated model has already achieved a fairly reliable results. However, it is still worth to be checking the other criterion called "Entropy", which is based on the measure of disorder.

Tree depth	20	1000	2000	3000	4000	5000
Accuracy	0.6971	0.7794	0.7817	0.7831	0.7834	0.7828

Table 4.8: DTC prediction accuracies on test set with different tree depths, using Entropy criterion.

The results are shown in Figure 4.4 and Table 4.8. It turns out that the Entropy model shares a very similar trend to the Gini Impurity model as oth models achieved around 78% accuracy at depths 1000.

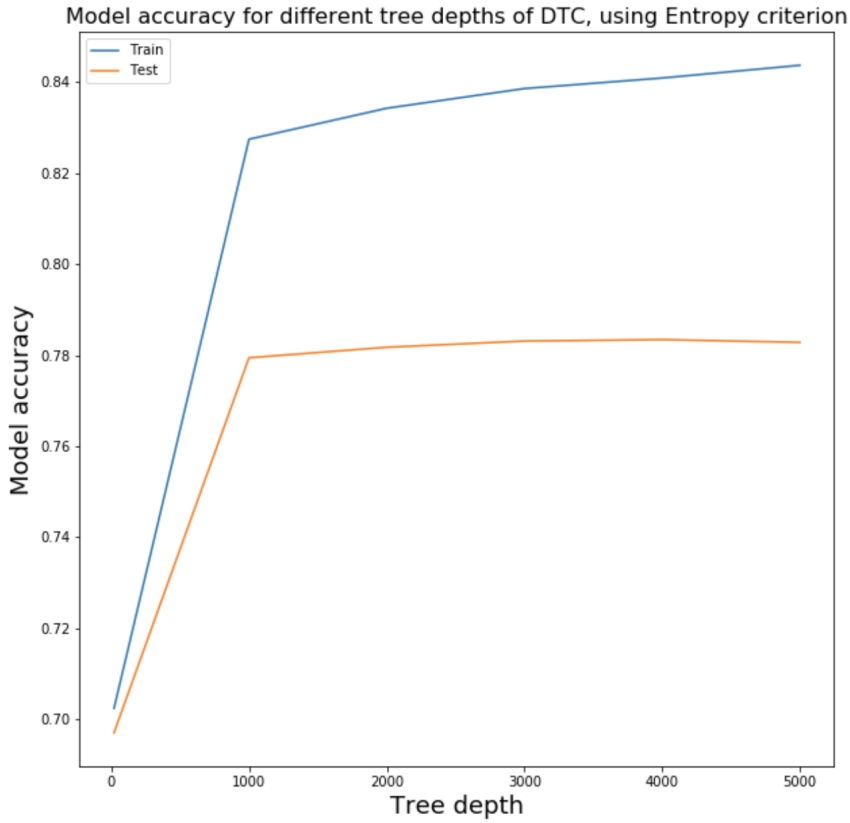

Figure 4.4: Model accuracy curve with different depth, using Entropy criterion.

4.4 Support vector machine

Support Vector Machine (SVM) is another powerful ML algorithm that can train classification models. The details of its underlying working mechanisms are not discussed in this project. More reading on how it is implemented can be found **here**. The purpose of using this algorithm is to compare the performance of different ML models when tasked with sentiment analysis. More precisely, sentiment analysis of the reviews which this analyser system is based on.

Parameters

In order to find the best model, there are many configurable parameters associated with the SVM algorithm. The main function used for this section is the `sklearn.svm.SVC()` function from the Scikit-Learn package. According to its official documentation, which can be found in this **link**, there is a list of configurable parameters. For the development of this project, only the following list of parameters will be configured:

- C the regularisation parameter.
- $Kernel$
- $Gamma$

K-Fold Cross Validation

Cross validation is a technique used when training ML models. It helps to resolve the problem of having only a limited amount of data. As for this project, the dataset is obtained directly from

`kaggle.com`. Thus the number of data samples available are fixed. To make the best out of the data, cross validation is introduced in this section.

Cross validation works by training several machine learning models on subsets of the data and validate their performance on the complementary sets of the data. For example, the dataset can be divided into 4 subsets, used to train 4 different models. Model 1 uses the first 25% of the first subset for evaluation, and the rest 75% for training. The second model will use the second 25% of the second subset of data for evaluation, and so on. A more intuitive representation is shown in Figure 4.5.

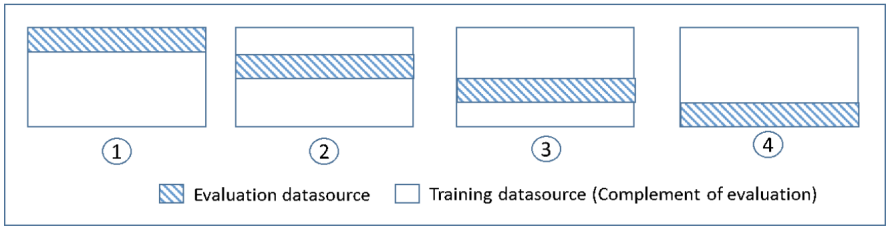

Figure 4.5: Example illustration of 4-fold cross validation.

A more advanced version of cross validation is K-Fold Cross Validation. The input data is split into k subsets, a model can be trained on all but one of the subsets, and use the last subset for evaluation. This process can be repeated for K times, each time using a different subset for evaluation. Thus all of the data samples will be used in training, the model produced will less likely to underfit the data.

Testing using GridSearchCV

The number of different parameters, together with K-fold cross validation can be cumbersome to implement. A useful tool provided by the Scikit-Learn package is the sklearn.model_selection. GridSearchCV() function. It takes a set of parameters to be configured and performs an exhaustive search over the different combinations to find the best parameters for a model.

After undergoing the exhaustive search, the best parameters found are displayed in Table 4.9. The evaluation metric results are shown in Table

C: 1	**Gamma**: 1	**Kernel**: 'rbf'

Table 4.9: Best parameters for SVC using GridSearchCV().

	precision	**recall**	**f1-score**
0	0.74	0.83	0.78
1	0.81	0.72	0.76
Accuracy	0.7715		

Table 4.10: Evaluation metric of the SVC classifier with the best parameters.

4.5 Random Forest Classifier

Random Forest Classifier (RFC) is an even more powerful tree based classifier than the DTC. When training a model, this algorithm produces many different tree classifiers, and the trees work collectively in when making the final predictions.

There are 4 main steps of the Random Forest Classifier:

1. Select random samples from a given dataset.
2. Construct a decision tree for each sample and get a prediction result from each decision tree.
3. Perform a vote for each predicted result.
4. Select the prediction result with the most votes as the final prediction.

Performance result

In Table 4.11 is the evaluation metric of the RFC using a max depth of 1000.

	precision	**recall**	**f1-score**
0	0.75	0.81	0.77
1	0.79	0.73	0.76
Accuracy	0.7678		

Table 4.11: Evaluation metric of the RFC classifier.

4.6 MLP Classifier

"MLPClassifier stands for Multi-layer Perceptron classifier which in the name itself connects to a Neural Network. Unlike other classification algorithms such as Support Vectors or Naive Bayes Classifier, MLPClassifier relies on an underlying Neural Network to perform the task of classification." For the implementation of this classifier, the `sklearn.neural_network.MLPClassifier()` function from the Scikit-Learn package is used. [9]

Testing using GridSearchCV

Below is the list of parameters being configured using the `sklearn.model_selection.GridSearchCV()` function:

- `hidden_layer_sizes`
- `activation`
- `solver`
- `alpha`
- `learning_rate`

Performance result

The final results of the MLP classifier model generated is recorded in Table 4.12.

	precision	recall	f1-score
0	0.76	0.82	0.79
1	0.81	0.75	0.78
Accuracy	0.7845		

Table 4.12: Evaluation metric of the MLP classifier.

4.7 Model Selection

The best performance of the different models generated in the previous sections are compared in Table 4.13. Other property comparisons are illustrated in Figure 4.6.

Classifiers	DTC-1000	SVM	RFC	MLPClassifier
Accuracy	0.7807	0.7715	0.7678	0.7845

Table 4.13: Accuracy comparison between different models.

Bagging

As seen from Figure 4.6, all the models have very similar performances in terms of accuracy and the various precision and recall values. The high scores achieved suggesting they are all equally reliable models that can be used in the sentiment analysis of this review analyser syste. Therefore the approach taken in this project is to create a model ensemble, also called "bagging", where all 4 models are used to produce the final prediction. This is similar to the working mechanism of the Random Forest Classifier, for every input review, its sentiment will be predicted by each of the four models, and the sentiment with the most vote will become final output. This further improves the reliability of the sentiment analysis, which in turn improves the confidence of the system in selecting the best and worst features of each hotel.

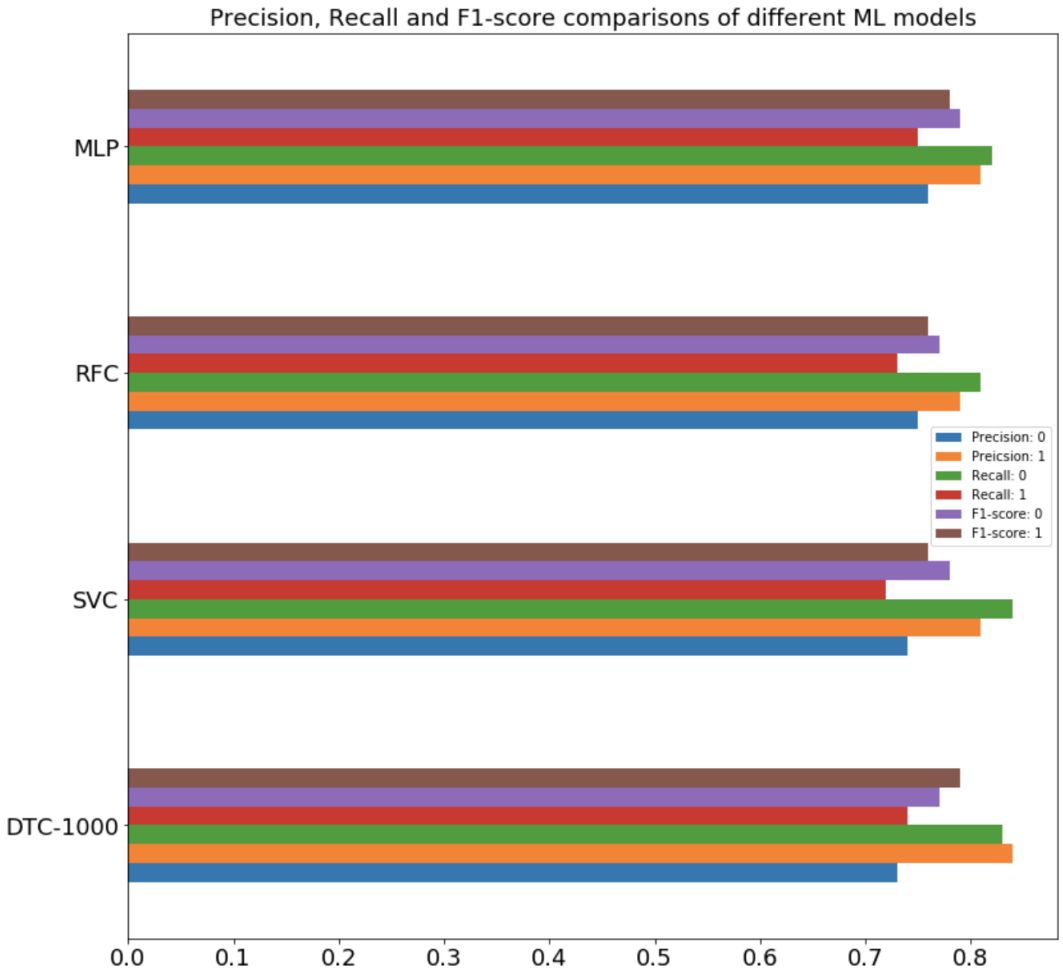

Figure 4.6: Bar chart of the evaluation metrics comparisons.

Chapter 5

Conclusions and Evaluation

5.1 Achievements

There are three key topics achieved in this project. The first one being best and worst features extraction from the hotel reviews. Second one is sentiment analysis on the textual data. Third one is hotel report generation with user accessible interface.

Best and worst features extraction

Based on the "Hotel_Reviews.csv" dataset from the **kaggle** website, a thorough analysis on the textual data has been carried out. This project seeks to incorporate various NLP techniques such as lexical, syntactic and semantic analysis to stratify long strings of data into smaller and more meaningful chunks. This allows hotel feature extractions to be more accurate, at the same time the corresponding values that describes the features were also captured by the system. Thus the end results are more understandable and interesting to the users.

Sentiment analysis

As mentioned in the earlier chapters, the motivation for sentiment analysis comes from mixed sentiments in reviews. In some negative reviews, the users may have been talking about some aspects of the hotel which they liked, and vice versa talking about the downsides in their positive comments. Hence by digging deeper into each reviews and transform the textual data into numerical representations using TF-IDF has allowed sentiment analysis to become possible in this project. Four different ML models have been trained with high accuracy in classifying the sentiment of reviews in the dataset. The strengths of these models are combined using the "bagging" technique to further strengthen the prediction of sentiments. Hence the mixed sentences in reviews can be reclassified, producing better feature extractions by the system.

User accessible interface

The majority of the heavy contents lies in the implementation of data processing and NLP, which are essential to forming a reliable backend database. However, the processed data are worthless if they cannot be utilised. Thus another main area of focus in the development of this system is building an accessible user interface. This project has used the Django web framework to build a user facing front-end and a MVT server as the back-end. Upon receiving a user request, the back-end server fetches the processed data from the database to generate the requested report and

renders it on the user's web browser. This therefore brings all the separate components together, making the system complete.

5.2 Challenges and Evaluation

Defining grammar

One of the challenges faced in this project is in the syntactic analysis stage. How to solve the problem of chunking was not immediately apparent. Various experiments have been carried out, such as extracting words within a certain distances from each other, perform Tf-IDF on raw reviews to identify the most used words. However the "features" extracted were noisy and less interpretable. Better progress was made with the realisation of finding noun phrases within the sentences. Then after knowing which parts of the reviews to extract, it became much clearer that some forms of matching to the POS-tags of the word tokens are required. The new challenge faced then was to devise the appropriate grammars for chunking. The breakthrough was in using regular expression to capture variations of the noun phrases, rather than using many predefined grammars that tries to capture the different sentence structures. However, after constructing a grammar that works well in chunking the positive reviews, it produced poor results when tested against the negative reviews. The reason behind is that in negative reviews, the sentences in the string tends to be longer, which makes it difficult to properly perform chunking. For a long string, multiple chunks can be captured at the same time with some degree of overlapping. Some words from the previous sentence might be thought to have a connection with the words in the second sentence. Therefore, it is necessary to separate the raw review from into separate sentences first to reduce ambiguity. The two approaches taken are using regular expressions and `sent_tokenize()` function from the NLTK package to automatically separate individual sentences in a review.

Positive reviews are easier to process, because reviewers tends to be in a more relaxed state. The good aspects about a certain feature are simply stated. The negative reviews on the other hand are more strenuous to deal with because they tend to be longer than the positve review. The context described is also more complicated because reviewers are most likely to be more impulsive when expressing why they disliked certain aspects about the hotel. Also the wording are not as straightforward, for example, a broken TV may be written as "TV was broken", or "TV did not work". If only "work" is matched by the grammar, and the negator "not" is not, then it changes the meaning of the sentence entirely.

ML models

Another challenging aspect arises from the ML models. The ease of training a model using the Scikit-Learn package can easily make the sentiment analysis problem look easier than it is. Underestimating the task often led to overfitting models that does not perform well in the real application. Therefore it is important to do background readings around the topic and understand how the parameters of the models should be configured, and knowing how to evaluate the performance.

After having successfully trained the 4 models that performs equally well, choosing which model to be used by the system required some considerations. The inspiration of using the "bagging" method came from the paper written by Pedro Domingos. In the paper he suggested "creating a model ensembles is now standard. In the simplest technique, called *bagging*, we simply generate

random variations of the training set by resampling, learn a classifier on each, and combine the results by voting. This works because it greatly reduces variance while only slightly increasing bias." Hence, a voting system that ensembles the 4 different classifiers were used in the sentiment analysis of this project.

There are already many sentiment analysers available on the internet, even within the NLTK package, there exist a `nltk.sentiment.sentiment_analyzer.SentimentAnalyzer()` function which can be simply imported and used. However, one of the decisions made in the development of this system was to train a dedicated sentiment classifier based on the dataset. This is because the models would be fitted better to the dataset, in particular reviews from Booking.com. Thus is capable of determining sentiments of future reviews.

5.3 Future Work

In sentiment analysis, training data can be further improved by taking into account other factors such as polarity, negator and modifier. This will require a more complex grammar being defined to capture more information from the negative reviews. The advantage would be that sentences such as "the room was not so clean" can be mapped by the grammar, values such as "not clean" can be extracted. This could increase the precision of the models correctly classifying reviews with negative sentiment. [10]

Overall, the negative reviews are more complexed to process. For future work, maybe instead of using the results from the text analysis, which relies on POS-tags, use the raw forms of the reviews, and prepare the training data with N-gram approach, maximise the chance of the system identifying the main focus of the review. Therefore, the system may be modified by training a classifier for positive reviews, and a classifier for negative negative reviews. So when reclassifying the individual sentences, they need to be examined by the positive and negative classifier, then the class with the highest confidence in its prediction will be used. This is down to the competition between the model precision scores of the positive and negative classifiers, which may produce a more robust result.

Instead of using various `.csv` files as the back-end database of the system, the processed files could be transformed into a MySql database. That way the data stored are more maintainable and scalable. The speed to fetching data and responding to user request can also be optimised.

Appendix A

Code

Link to source code: `https://github.com/1109LLL/FYP.git`

```
    # Lexical analysis
    def lexical_analysis(self,sent):
        # Tokenising the sentence into separate word tokens
        sent = sent.lower()
        tokenise = word_tokenize(sent)

        # Check spelling
        tokenise = self.spell_check(tokenise)

        # Apply parts of speech tags to the tokens
        pos_tagged = nltk.pos_tag(tokenise)
        return pos_tagged

    def spell_check(self, tokens):
        spell = SpellChecker()
        misspelled = spell.unknown(tokens)

        for word in misspelled:
            tokens[tokens.index(word)] = spell.correction(word)
        return tokens
```

Listing A.1: Lexical Analysis

```
'''
POS tags list
TAG  POS  MORPHOLOGY  DESCRIPTION
$ SYM   symbol, currency
``  PUNCT PunctType=quot PunctSide=ini  opening quotation mark
''  PUNCT PunctType=quot PunctSide=fin  closing quotation mark
, PUNCT PunctType=comm  punctuation mark, comma
-LRB- PUNCT PunctType=brck PunctSide=ini  left round bracket
-RRB- PUNCT PunctType=brck PunctSide=fin  right round bracket
. PUNCT PunctType=peri  punctuation mark, sentence closer
: PUNCT   punctuation mark, colon or ellipsis
ADD X   email
AFX ADJ Hyph=yes  affix
CC  CCONJ ConjType=comp conjunction, coordinating
CD  NUM NumType=card  cardinal number
DT  DET   determiner
EX  PRON  AdvType=ex  existential there
```

```
FW  X Foreign=yes foreign word
GW  X   additional word in multi-word expression
HYPH  PUNCT PunctType=dash  punctuation mark, hyphen
IN  ADP   conjunction, subordinating or preposition
JJ  ADJ Degree=pos  adjective
JJR ADJ Degree=comp adjective, comparative
JJS ADJ Degree=sup  adjective, superlative
LS  X NumType=ord list item marker
MD  VERB  VerbType=mod  verb, modal auxiliary
NFP PUNCT   superfluous punctuation
NIL X   missing tag
NN  NOUN  Number=sing noun, singular or mass
NNP PROPN NounType=prop Number=sing noun, proper singular
NNPS  PROPN NounType=prop Number=plur noun, proper plural
NNS NOUN  Number=plur noun, plural
PDT DET   predeterminer
POS PART  Poss=yes  possessive ending
PRP PRON  PronType=prs  pronoun, personal
PRP$  DET PronType=prs Poss=yes pronoun, possessive
RB  ADV Degree=pos  adverb
RBR ADV Degree=comp adverb, comparative
RBS ADV Degree=sup  adverb, superlative
RP  ADP   adverb, particle
SP  SPACE   space
SYM SYM   symbol
TO  PART  PartType=inf VerbForm=inf infinitival "to"
UH  INTJ    interjection
VB  VERB  VerbForm=inf  verb, base form
VBD VERB  VerbForm=fin Tense=past verb, past tense
VBG VERB  VerbForm=part Tense=pres Aspect=prog  verb, gerund or present participle
VBN VERB  VerbForm=part Tense=past Aspect=perf  verb, past participle
VBP VERB  VerbForm=fin Tense=pres verb, non-3rd person singular present
VBZ VERB  VerbForm=fin Tense=pres Number=sing Person=three  verb, 3rd person
    singular present
WDT DET   wh-determiner
WP  PRON    wh-pronoun, personal
WP$ DET Poss=yes  wh-pronoun, possessive
WRB ADV   wh-adverb
XX  X   unknown
_SP SPACE

src = https://spacy.io/api/annotation
'''
```

Listing A.2: POS Tags

```
    # Syntactic Analysis
    def syntactic_analysis_for_positives(self,sent):
        # Define chunking grammar using regular expression
        # grammar_1 captures sentence structure where the adjectives comes after
    the nouns
        grammar_1 = r"""
                    NP: {(<NN>|<NNS>)+<PRP>*<VBD>*<VBZ>*<RB>*<VBN>*<JJ>*((<CC><VBD
    >+)|(<CC><RB>+<JJ>*)|(<CC><VBN>+)|(<CC><JJ>+))*}
                    """
        # grammar_2 captures sentence structure where the adjectives comes before
    the nouns
        grammar_2 = r"""
```

```
                    NP: {<JJ>(<NN>|<NNS>)+}
                    """

        chunks_1 = self.grammar_parsing(grammar_1,sent)
        chunks_2 = self.grammar_parsing(grammar_2,sent)
        chunks = chunks_1 + chunks_2

        return chunks

    def syntactic_analysis_for_negatives(self, sent):
        # Define chunking grammar using regular expression

        grammar_1 = r"""
                    NP: {(<JJ>|<RB>|<VBD>)(<NN>|<NNS>)}
                    """
        # grammar_2 captures sentence structure where the adjectives comes after the nouns
        grammar_2 = r"""
                    NP: {(<NN>|<NNS>)<VBP>*<VBD>*<JJ>*<VB>*}
                    """

        chunks_1 = self.grammar_parsing(grammar_1,sent)
        chunks_2 = self.grammar_parsing(grammar_2,sent)
        chunks = chunks_1 + chunks_2

        return chunks

    # Parse the sentence and chunking using grammar
    def grammar_parsing(self,grammar,sent):
        cp = nltk.RegexpParser(grammar)
        result = cp.parse(sent)

        chunks = []
        for subtree in result.subtrees():
            # if subtree length smaller than 2 -> single words -> ignore
            if len(subtree) < 2 or subtree.label()=="S":
                continue

            # Check for duplicates
            if not subtree in chunks:
                chunks.append(subtree)

        return chunks
```

Listing A.3: Grammars used for chunking in reviews in Syntactic Analysis.

```
    def form_training_set_for_sentiment_clf_decision_tree(self):
        getter = Getter()
        hotel_list = getter.getHotelList()

        pos_val = []
        neg_val = []
        # Extract the positive and negative dictionaries (noun and adj) from each hotel
        # obtain a list of only the values
        for hotel in hotel_list.values:
            hotel_name = hotel[0]

```

```
            pos_dict = getter.get_pos_noun_adj_dict_of_hotel(hotel_name)
            pos_values_list = list(pos_dict.values())
            for value in pos_values_list:
                if len(value) == 1:
                    pos_val.append(value[0])
                else:
                    pos_val.append(' '.join(value))


            neg_dict = getter.get_neg_noun_adj_dict_of_hotel(hotel_name)
            neg_values_list = list(neg_dict.values())
            for value_ in neg_values_list:
                if len(value_) == 1:
                    neg_val.append(value_[0])
                else:
                    neg_val.append(' '.join(value_))

        y_pos_label = [1] * len(pos_val)
        y_neg_label = [0] * len(neg_val)

        X = pos_val + neg_val
        y = y_pos_label + y_neg_label

        return X, y

    def fit_training_set_using_tfidf_vectorizer(self):
        print("-----------------------------------------")
        now = datetime.datetime.now()
        print("{} : start fitting tfidf".format(now.strftime("%Y-%m-%d %H:%M:%S")))

        getter = Getter()
        X, y = getter.get_training_set_for_sentiment_clf_decision_tree()

        tfIdfVectorizer = TfidfVectorizer(use_idf=True)
        X_tfidf = tfIdfVectorizer.fit_transform(X)

        now = datetime.datetime.now()
        print("{} : finished fit_transform".format(now.strftime("%Y-%m-%d %H:%M:%S"
    )))
        print("-----------------------------------------")

        return X_tfidf, tfIdfVectorizer
```

Listing A.4: Document term matrix using TF-IDF.

```
print("-----------------------------------------")
now = datetime.datetime.now()
print("{} : start training".format(now.strftime("%Y-%m-%d %H:%M:%S")))

getter = Getter()
X_tfidf, y, tfIdfVectorizer = getter.
    get_X_tfidf_y_training_set_for_sentiment_clf_decision_tree()

# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X_tfidf, y, test_size=0.2,
    random_state=42)

# Create SVC object
```

```
svc = SVC(random_state=42)

# Train Decision Tree Classifer
svc = svc.fit(X_train,y_train)

param_grid = {'C': [0.1, 1, 10, 100, 1000],
              'gamma': [1, 0.1, 0.01, 0.001, 0.0001],
              'kernel': ['rbf', 'poly']}

grid = GridSearchCV(SVC(), param_grid, cv=5, refit = True, verbose = 3, n_jobs=-1)

# fitting the model for grid search
grid.fit(X_train, y_train)

now = datetime.datetime.now()

# Create predictions

print("best estimator = {}".format(grid.best_estimator_))
print("best parameters = {}".format(grid.best_params_))

grid_predictions = grid.predict(X_test)


# Model Accuracy
print(metrics.classification_report(y_test,grid_predictions))
print("Accuracy: {}".format(round(metrics.accuracy_score(y_test, grid_predictions)
    ,4)))
print("-------------------------------------")
```

Listing A.5: GridSearchCV to find best SVC classifier.