Hansard Historical Sentiment Analysis and Comparison

Final Report for CS39440 Major Project

Author: Adam Neaves (adn2@aber.ac.uk)
Supervisor: Dr. Amanda Clare (afc@aber.ac.uk)

4th March 2018
Version: 1.1 (Draft)

This report was submitted as partial fulfilment of a BSc degree in Artificial Intelligence and Robotics (GH7P)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, UK

Declaration of originality

I confirm that:

- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for Unacceptable Academic Practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the regulations on Unacceptable Academic Practice from the University's Academic Quality and Records Office (AQRO) and the relevant sections of the current Student Handbook of the Department of Computer Science.
- In submitting this work I understand and agree to abide by the University's regulations governing these issues.

Name: Adam Neaves

Date: 04/05/2018

Consent to share this work

By including my name below, I hereby agree to this dissertation being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Name: Adam Neaves

Date: 04/05/2018

Acknowledgements

I would like to express my thanks to Amanda Clare, my supervisor on this project, without whose guidance I would've gotten lost in the details.

I would also like to thank Tom Doyle, my friend, roommate, and fellow Comp Sci student for being a good listener when I needed to talk through a problem. He was the best rubber duck a programmer could ask for.

I'm grateful to my mother and sister for their support during my career as a student. Without their pushing I could never have achieved as much as I have.

Finally, I'd like to thank my partner, River. Without their patience I would not have known how to deal with the stress, and whose support was invaluable despite often not knowing what I was talking about.

Abstract

Politics affects all aspects of a person's life. The results of debates in Parliament may have a profound influence on an individual's life, and knowing how your local MP speaks and the opinions they express during these debates could prove useful. Websites, such as They Work For You, show a user how their local MP votes, and how often they attend debates, ask questions, and other information that may allow the user to make informed decisions when voting during elections.

The aim of the Hansard Sentiment Analysis Tool is to provide a tool to automatically detect the sentiment of statements made during political debates, and to compare it with sentiment expressed about the topic previously, or compare it with sentiment expressed by the same MP. The sentiment analysis will use a machine learning approach, where a model will be trained on labelled datasets generated from the source data.

Being able to view how sentiment changed over time, especially about certain topics, could prove useful for a user who wants to not only know how an MP votes, but also how they represent themselves and their constituency in parliament. If an MP is seen to change opinion on a topic as time passes, this information could be used to keep voters informed.

This report will document the process of designing and developing the Hansard Sentiment Analysis Tool, highlighting any challenges encountered during development, and also the results of the technical work.

CONTENTS

1 Background & Objectives **1**
- 1.1 Background . . . 1
  - 1.1.1 Hansard Dataset . . . 1
  - 1.1.2 Natural Language Processing . . . 2
  - 1.1.3 Related Work . . . 2
  - 1.1.4 Technical Research . . . 4
- 1.2 Analysis . . . 4
  - 1.2.1 Project Aims . . . 5
- 1.3 Process . . . 5
- 1.4 Background Conclusion . . . 6

2 Design **7**
- 2.1 Overall Architecture . . . 7
  - 2.1.1 Data Downloader . . . 8
  - 2.1.2 Data Parser . . . 8
  - 2.1.3 Manual Annotation Tool . . . 9
  - 2.1.4 Sentiment Analyzer . . . 10
  - 2.1.5 Search Tool . . . 10
- 2.2 Data Design . . . 10
  - 2.2.1 Parsed Data . . . 10
  - 2.2.2 Annotated Data . . . 12
- 2.3 User Interface . . . 13
- 2.4 Algorithm Design . . . 13
  - 2.4.1 AI Algorithm Choice . . . 13
  - 2.4.2 Parsing . . . 14
- 2.5 Design Conclusion . . . 14

3 Implementation **16**
- 3.1 Data Downloading . . . 16
  - 3.1.1 Online Connection . . . 16
  - 3.1.2 Dataset Size . . . 17
- 3.2 Data Parsing . . . 17
  - 3.2.1 Name Recognition and Disambiguation . . . 18
  - 3.2.2 Name Matching . . . 19
  - 3.2.3 Saving Parsed Data . . . 20
- 3.3 Manual Annotation Tool . . . 21
  - 3.3.1 Sentence Splitting . . . 22
- 3.4 Sentiment Analyser . . . 23
  - 3.4.1 Naive Bayes Classifier . . . 24
- 3.5 Implementation Conclusion . . . 27

4 Testing **28**
- 4.1 Overall Approach to Testing . . . 28
- 4.2 Unit Tests . . . 28
  - 4.2.1 Natural Language Processing Module . . . 29
  - 4.2.2 Parser Tests . . . 29

4.3 Testing Conclusion 30

5 Evaluation **31**
5.1 Requirements Comparison 31
5.1.1 Data Parser Difficulties 32
5.1.2 Annotating Data 32
5.1.3 AI Algorithms 33
5.2 Future Work 33
5.2.1 Testing 34
5.2.2 More AI Algorithms 34
5.2.3 Develop Search Tool 34
5.2.4 Work with Additional Data Sources 34
5.3 Project Conclusion 34

Appendices **35**

A Testing Log **36**

B Ethics Submission **40**

Annotated Bibliography **42**

LIST OF FIGURES

1.1 Screenshot of a Trello Board showing work in progress, done, and not yet started 6

2.1 Functional block diagram of the overall system. Arrows represents data movement 8
2.2 Screen capture from the Hansard Archive website, showing the number of files available 9
2.3 A sample of data from the file *S5CV0021P0.xml*, a file from the fifth series of data 11
2.4 Diagram showing the structure of the parsed data, in XML. The *Date* tag is the root of the file 12
2.5 A mock-up design of the Search tool, which would be the main user interface for the project 13
2.6 Bayes Theorem 14

3.1 An example from the original dataset, showing that names exist only within *member* tags 18
3.2 Extract of code from the Name Extractor Method. 19
3.3 Screenshot from the Annotation Tool running on the Command Line 22
3.4 Diagram showing how K-fold validation works, when $k = 5$. Rows represent training rounds, columns are the chunks of data. 25
3.5 Probability of correct classification based on random selection. $P(X)$ is the Probability of X 27

LIST OF TABLES

3.1 Table showing the different ways names might or might not refer to the same person 19
3.2 Excerpt from the log of the original parser parsing Series 6, showing time it took to parse each file 21
3.3 Size comparison between the Original Hansard Dataset and the Parsed Dataset generated by the Parser 21
3.4 Weather training data as an example of features 24
3.5 Table of results from training the Naive Bayes algorithm using 10-fold cross validation. 26
3.6 Table of results from training the Naive Bayes algorithm using 10-fold cross validation. 27

5.1 Comparison between required functions and the functions that were developed . . 31

LIST OF ALGORITHMS

2.1 Earlier version of the Parsed Data design 11
2.2 Data Parser Pseudo-code 15
3.1 Snippet of the Downloader Code 17
3.2 Pseudocode representing the name matching method 20
3.3 Sentence Splitting method, with included regular expressions 23
3.4 Snippet of code that trains and tests the classification algorithm 26

Chapter 1

Background & Objectives

1.1 Background

1.1.1 Hansard Dataset

The Hansard is a set of documents produced by the British Parliament, which began in the 18th and 19th century. These documents contain reports and details of debates in the House of Commons, going back to the year 1803. Eventually, in 1907, these reports were made official and started being produced by Parliament itself, becoming The Official Report, though still unofficially known as Hansard. Along with becoming official, a report was officially defined as being one:

> which, though not strictly verbatim, is substantially the verbatim report, with repetitions and redundancies omitted and with obvious mistakes corrected, but which on the other hand leaves out nothing that adds to the meaning of the speech or illustrates the argument" [1]

Hansard is available in a variety of versions. The most commonly used and best known version is the Daily Hansard, which appears each morning and reports the previous days proceedings. However, access to this is via an API that only provides the most recent 7 days. For this project, most training and processing will be done on the Historical Hansard dataset, which is a dataset containing all 6 series of Hansard, between 1803 to 2004, though it is expected that some of the older documents will be less useful for this project due to the likelihood of them using outdated speech that would no longer be relevant.

The Historical Hansard Dataset is available online in an XML format. Multiple documents per series are available, each document covering a few days debates at most. It is a large dataset, reaching around 10Gb in size in total. Most of the documents available are scanned from hard copies, rather than typed up directly, meaning there is a possibility of small errors from the scanning process that may have to be dealt with. Additionally, from preliminary investigation, the data itself appears to be only loosely formatted, and each of the six series appear to be formatted in a slightly different way, so any system designed to read this data will have to be capable of dealing with any changes to formatting.

1.1.2 Natural Language Processing

Natural Language Processing (NLP) is the exploration of how computers can be used to understand and manipulate natural language text or speech to do useful things [2]. Computers are very good at dealing with numbers and performing complex calculations at high speed, but they are not as good at understand spoken or written language. Because of this, a large part of NLP is the act of processing the data, or text, to make it easier for the computer to understand and work with, as well as gathering knowledge on how human beings understand and use language so that appropriate tools and techniques can be developed [2]. NLP covers multiple topics, such as *Named Entity Recognition*, *Part of Speech Tagging*, and *Sentence Boundary Disambiguation*. However, the part this project is mainly interested is the act of *Sentiment Analysis*.

Sentiment Analysis, also known as Opinion Mining, is the process of identifying and extracting the opinions expressed in a piece of text [3]. It aims to determine the attitude of a speaker or writer towards a topic, or the overall polarity of a piece of text. This can be a judgment made by the writer or speaker, in the case of reviews, or the emotional state of the speaker or writer.

A basic version of Sentiment Analysis classifies the polarity of a piece of text, classifying it as either positive, negative or neutral. A more advanced version would be, for example, looking at emotions expressed in the text, classifying it as angry, happy, or sad, as some examples. A basic method used can be to compare a piece to two lists of words, one a list of words that usually denote a positive polarity, and one that usually denotes a negative polarity. A system can then simply count the number of positive and negative words in a piece of text, account for any negation (Saying *not great* would change the word *great* from a positive to a negative word, for instance) and whichever type of word was most common would denote the piece of texts sentiment. However, this method is likely only useful for those pieces of text where it is known that strong sentiment is likely to be expressed in a simple enough manner, in text such as a review of a product or film.

Stance Detection is another aspect of Natural Language Processing, similar to Sentiment Analysis. However, the difference here is that Stance Detection sets out to classify the stance of text towards a particular target, or entity. Whereas Sentiment Analysis is the overall mood or polarity of text, Stance Detection specifically looks to see if the text is for, against, or neutral towards something [4]. An example would be detecting the stance of a news article towards the subject it's being written about. This may be more applicable to the Hansard Dataset than Sentiment Analysis as the members of parliament are likely to be expressing some form of stance on a topic that they are debating, but also somewhat more complicated to do, as the subject of the stance must also be discovered from the text, which is not guaranteed to be explicitly stated.

1.1.3 Related Work

In researching the potential design of project, a few relevant pieces of work done by others were discovered, some of which had a useful impact on the design of this project.

1.1.3.1 Towards Sentiment Analysis on Parliamentary Debates in Hansard

Towards sentiment analysis on parliamentary debates in Hansard [5] is a paper which discussed the progress made by **Onyimadu *et al*** towards applying classic sentiment analysis techniques to

the Hansard dataset, such as word association. The paper details the proposed approach to sentiment analysis, by using *heuristic classifiers based on the use of statistical and syntactic clues in the text* and using a sentiment lexicon base known as the MPQA corpus (Multi Perspective Question Answering) to identify sentences containing known positive or negative words. They first classify a sentence using this lexicon, annotating sentences as positive or negative depending on the number of positive or negative words, before then applying syntactic clues to improve the classification, such as the presence of negations, such as *not* or *never*, and the inclusion of intensifying adverbs such as *very*. The paper reports an average of 43% correctly annotated sentences, claiming that the correctly annotated sentences were those *without compound opinions, sarcasm and comparative sentences,* showing that the style of debate speech renders their syntactic and lexical based approach insufficient for the task.

This paper demonstrates the need to customize sentiment analysis for the Hansard Dataset, as it shows that the standard techniques used are not applicable to the debate speech. Any design for the project should therefore acknowledge the need to be specifically trained on the Hansard Dataset.

1.1.3.2 They Work For You

They Work For You [6] is a website that allowed the user to search for their local MP via post code, and the site can then display the voting patterns for that MP, along with information about how often their votes align with their parties' votes, and shows examples of appearances made by that MP and what they said. The source code is publicly available on Github and uses python for a large part of their code base. Whilst it does not appear that they use any form of sentiment analysis, it is still a good example of the sort of thing that can be done using the parliamentary data, and would likely be well supplemented by my project, allowing them to also show how an MP might speak in debates, as well as how they vote.

This website shows what sort of information tends to be extracted from debates and shows what is usually considered interesting. In addition, it's code is open source and hosted on A GitHub Repository. In addition, it also uses the daily form of Hansard, and so can provide an insight into how to go about parsing data from that source.

1.1.3.3 The Fake News Challenge

The Fake News Challenge [7] is a challenge set up to explore *"how artificial intelligence technologies could be leveraged to combat fake news."* and aims to eventually produce a tool that can help human fact checkers tell if a news story is a hoax, or intentionally misleading. The first part of the challenge involved the use of Stance Analysis on a series of news articles, comparing the contents of the article with the headline, to tell if the article contents agree with the claims made in the headlines. As the project is set up as a competition, multiple teams submitted solutions to the problem, showing a variety of techniques in solving this problem.

This project is built on the work of **Ferreira et al.** [8], who built a dataset based around articles, the claims they are based on, and the stance of the article, where a *For* Stance means the article reports the claim as true, an *Against* Stance means the article reports the claim as false, and an *Observing* stance means the article reports on the claim, but does not report of its veracity.

Stance detection is a potential alternative to Sentiment Analysis for the project. Instead of the basic

Positive or *Negative* output of Sentiment Analysis, based on just the contents of the text being classified, Stance Analysis compares the contents of two pieces of text to see if they are related to the same topic, and if so, if they agree or disagree. This could be applied to the speech found in Hansard, by comparing the speech of the MP with the topic being debated, to automatically see if they are speaking for or against the topic, or have gone off topic.

1.1.4 Technical Research

Before any form of planning could begin for the project, some research on the kinds of technologies available was required. Three topics had to be researched, namely the language to be used, what methods were available for sentiment analysis, and what was available to extract the data from its original form to something more usable.

For sentiment analysis, and other required NLP tools, the Natural Language Toolkit (NLTK) [9] was found. This Python package provides methods and classes for a majority of NLP tasks, including everything required for this project. Additionally, its well documented, as it is commonly used by other projects that require some form of NLP. This means it would be easy to find solutions to any problem encountered during development, as it is highly likely someone else using the same package has encountered a similar issue and documented a solution online. Due to this, it was quickly decided that the NLTK package would be used for all NLP requirements, which also meant the language to be used would be Python.

There was also an NLP package available for Python called SpaCy, which is similar to NLTK. It is, however, newer, and less well documented than NLTK, simply because it is not yet as popular. This means that, should any issues arise, it may be harder to find a solution or any help on the subject. It is possible that in the future SpaCy may prove to be a more useful NLP package, but for the purposes of this project NLTK's commonness wins out.

Once a language was selected, some form of parsing tool had to be discovered. As the data is provided in XML, a commonly used semi-structured database format, the chosen parser solution had to be able to parse XML. An often used package for Python that could do this is lxml, a package which could read in XML data structures and translate them into its own set of classes to be used in Python. However, lxml expects well structured data, whereas the hansard dataset is not as well organised. For this reason, it was decided that BeautifulSoup4 [10] would be used, a module designed to parse less structured data, in HTML or XML. It makes use of the lxml module, but provides methods that allow the parsing of data whose structure is unknown.

1.2 Analysis

Following along from the background research, some decisions were made on how this project would proceed, and what challenges were expected. Additionally, the design of the overall system had to be developed, and the developmental process.

Due to the size and complexity of some of the source data, it was decided that the system would not be able to work directly with the original data without some form of intermediary parsing system. Thus, part of the project was to develop a parser that would get all relevant data from the original files, and reorganize them into a more useful format. It was therefore also necessary to

decide on the structure of the data once parsed, and how this data would be stored and accessed by the rest of the system. It would also be necessary to decide what exactly from the original data was relevant to the project. Additionally, it was important that speech be attributed to the correct member of parliament. It often appeared that the way an MP was referenced in the data would change, going from a full title, honorific and name to just surname. It was important that these different forms of name be recognized as the same person, otherwise speech attributed to just one person would be seen as being said by different people, reducing the accuracy of any searching. This could be done using Named Entity Recognition, an aspect of NLP.

Any form of supervised machine learning method requires training and testing datasets. Due to the nature of the data being used, there are no pre-existing sets of annotated data available, and thus the datasets used must be hand annotated. A tool designed to do this must therefore be produced that can assist in this lengthy process, allowing a user to generate a set of annotated data that can then be used to train an Artificial Intelligence.

1.2.1 Project Aims

From the analysis of the problem, a decision needed to be made about what exactly the project will aim to do. This can then be developed further into a proper design later on the developmental process.

The main target of the project is to be able to automatically extract the sentiment expressed in Parliamentary Debates, and track trends in the sentiment expressed by an individual or about a topic. In order to manage this, the project will have to undergo the following tasks:

- Download the Hansard Dataset
- Extract the relevant information from the dataset
- Extract Sentiment from the speech extracted, and save it in a way that can be searched for.

These tasks will likely need to be broken down further in order to be efficiently developed, but they represent the overall aims and goals of the project.

1.3 Process

The process selected for this project was Feature Driven Development (FDD), an agile styled methodology designed to focus on delivering working blocks of software repeatedly. Though this methodology is not usually designed for a single developer, it was chosen for its ability to break down the larger tasks of *Develop a fully functioning system* into smaller, easier to digest tasks of *Develop the part of the parser that searches for speech* or *Develop a method of comparing two names*. It began with an overall design of the system, and then this design was broken down into a list of ”features”, by breaking down the overall design into functional sections, represented by the blocks in the diagram. Each of those is decomposed further into a list of features for each one, which can then be targeted as milestones during development. Because these features are relatively small, completing them is a small task, and keeping track of the progress on each feature can give a decent report on the progress of the project as a whole.

For version control, git, and Github.com was used to maintain versions of code. This can then also be used as a backup service, with the online version of the code hosted on Github being a form of backup, and allowing the code to be transferred between multiple machines with relative ease.

A to-do page was also maintained on Trello, which is an online resource for producing Kanban style to-do lists. There were four columns in the Kanban Board:

- A to-do column for tasks that needed doing that were not yet complete,
- A Doing column for tasks currently in progress,
- A Blocked column for tasks that cannot be started until another task has been completed, or tasks that have been started but cannot continue until another task is done or some event is complete,
- A Done column for tasks that have been completed.

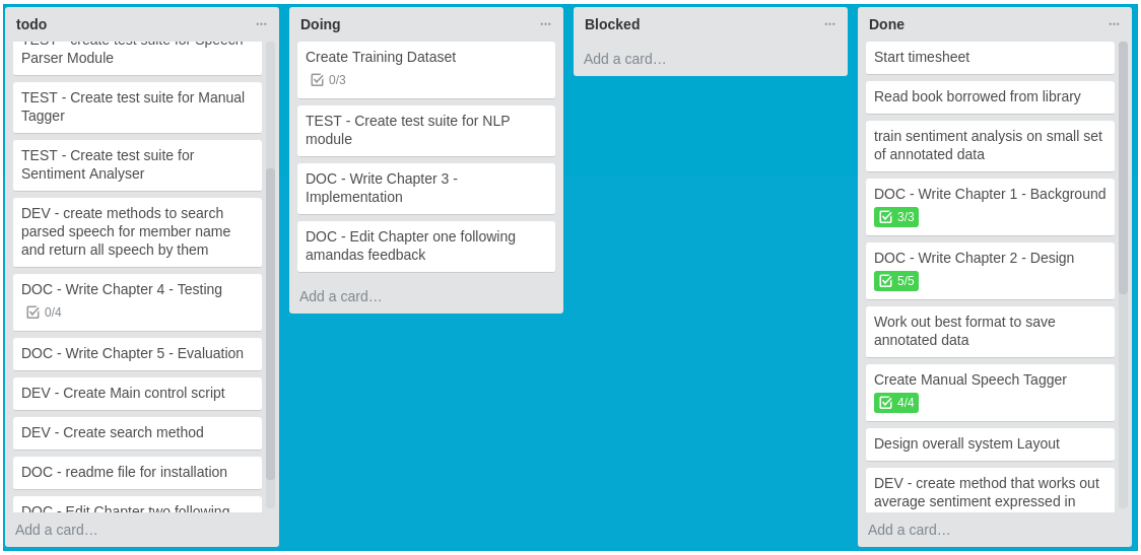

Figure 1.1: Screenshot of a Trello Board showing work in progress, done, and not yet started

In utilizing this tool for the organization of the project, development was planned out. Each ”Card” of the board represented one task, and could be moved between columns. More complicated tasks were broken down into separate tasks, or a checklist could be added to a card, should the task need to remain as a single task for whatever reason.

1.4 Background Conclusion

Now that the background work has been documented, this report will move on to look at the design process of the project, documenting the choices made for each part of the project and the reasoning behind them, and showing some of the design work undertaken.

Chapter 2

Design

In FDD, large, detailed design documents are not necessary. A simple overview of desired features and a breakdown of the overall system are sufficient, and are what will be detailed in this chapter.

2.1 Overall Architecture

As the project was developed using Feature Driven Development (FDD), the initial design of the system focused on the desired features of the system. These features were:

1. Download the dataset.
2. Parse the original data into a more useful format.
3. Allow a user to annotate the data with sentiment, to generate training and testing datasets.
4. Train an AI algorithm on the datasets generated.
5. Search the parsed data to find speech about a certain topic, or by a particular Member of Parliament.
6. Use the trained AI algorithm to extract sentiment from the speech found.
7. Display a comparison of the sentiment expressed about a topic or by an MP over time.

Using these features, the overall system can be designed, and an unofficial class diagram can be created that can help guide implementation.

As seen in Figure 2.1, the system is broken down into functional blocks, each representing a distinct part of the system. They were split this way to maintain readability in the source code. Whilst it would have been possible to keep all functionality in a single file, it would have made maintaining the code very difficult. Each functional block is therefore a separate class in the Python source code, so any interaction between them is easy to manage.

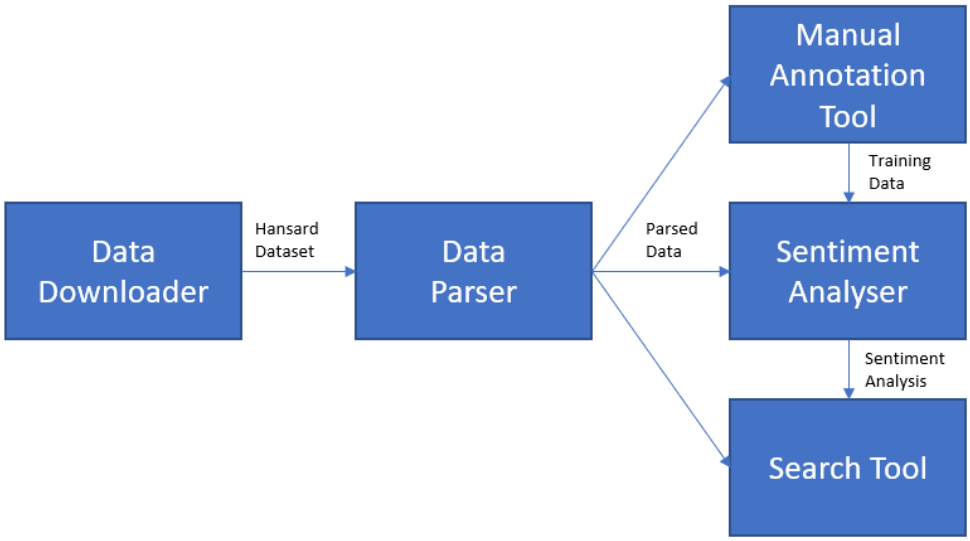

Figure 2.1: Functional block diagram of the overall system. Arrows represents data movement

2.1.1 Data Downloader

The Data Downloader is designed to download all of the Hansard Dataset from the site on-line, and store it locally for use by the rest of the system. It is likely that this first block will only need to be run a single time, but still proves useful in ensuring all the data is downloaded.

The files are to be downloaded from the Hansard Archive page. Each of the six series has a sub-directory under that root page, that contain all the data files. The data files have a distinct naming pattern that can be referenced using something like Regular Expressions in order to automate the downloading and searching of the files.

Originally, this was not going to be a part of the main system, as it is possible to download the data manually. However, only ten files are displayed on the web page each time, and some of the series have up to a thousand files to download. This can be seen in Figure 2.2. Therefore, an automated downloader needs to be implemented that can download the data automatically. The Design should allow the user to select which series they want to download and where it should be saved, and then it should automatically connect to the web page and attempt to download all the files.

2.1.2 Data Parser

The Parser is designed to read the original files downloaded by the Data Downloader, and save all relevant data in separate location, in a more consistent layout than that of the original dataset. This should allow any usage of the data from other parts of the system to be much simpler, and thus faster and less prone to error. It should ensure speech is always attributed to the correct person, even if their names are presented differently. This means the Data Parser will have to utilize a part

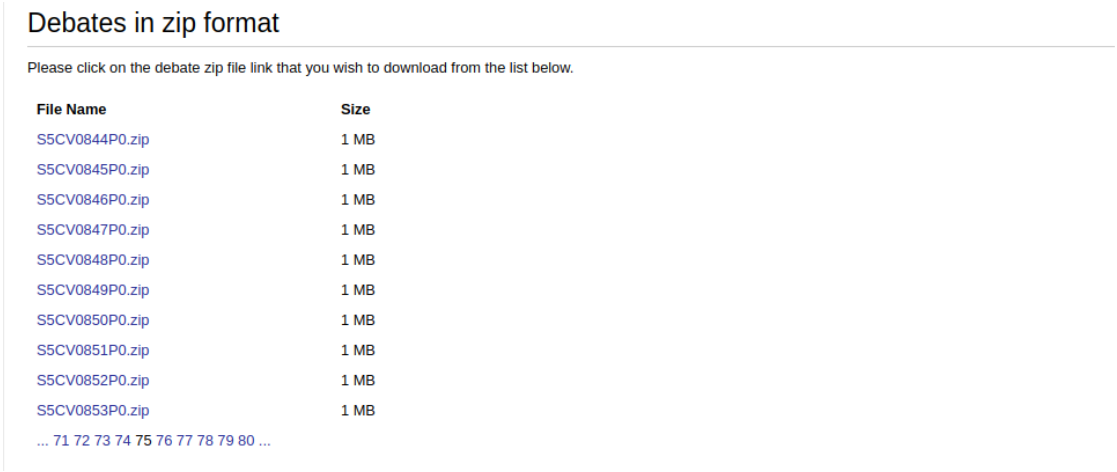

Figure 2.2: Screen capture from the Hansard Archive website, showing the number of files available

of NLP known as Name Disambiguation, which is not a simple task, and is discussed in further detail in Section 3.2.1.

Additionally, it was decided early on during the design that only Spoken speech would be used in this project. This was because written words were expected to be less likely to express any strong sentiment, as the written word gives the Member of Parliament time to edit and decide upon exactly what they wanted to respond to a question with. The spoken word was thought to be more likely to display the actual opinions and sentiments of the speaker, as they must respond in real time to questions and rebuttals.

2.1.3 Manual Annotation Tool

In Artificial Intelligence development, there is a concept known as *Ground Truth*, which is a representation of the reality that an algorithm wants to predict; such as the changes to the stock market or the sentiment expressed by a piece of text. It's often commonly represented by the training data used to train an AI algorithm when using a supervised or semi-supervised algorithm. Thus, for this project, the ground truth must be decided upon and represented by a set of training and testing data.

The Manual Annotation Tool (the MAT) is designed to allow a user to create said testing and training datasets from the parsed data. It should show a selection of speech to the user, and ask them to annotate it as either a positive sentiment, negative sentiment, or neutral sentiment. These choices will be recorded, and can then be used to train a machine learning algorithm to extract sentiment from the remaining data. It would also provide the option to edit the text, the MP's name, or the topic title, to allow the user to correct typos transferred from the original dataset.

Originally, the MAT would display an entire paragraph of speech, displaying everything a Member of Parliament said about a topic on a particular day. However, this would often display too much text, with parts showing positive sentiment and other parts of the same text showing negative sentiment. Additionally, it was realized that attempting to train an AI algorithm on these large blocks of text would be far too complex. For those reasons, the MAT was modified so that it would

only display a single sentence at a time, allowing a more accurate way of annotating sentiment.

The MAT is designed to accept the parsed data from the parser described in Section 2.1.2, and should break the speech found in the data down into sentences, then output the sentence and sentiment annotated to it as described in Section 2.2.2

2.1.4 Sentiment Analyzer

The Sentiment Analyzer will use the datasets generated by the MAT and produce a model based off the data. It should also provide the ability to load a previously trained model, rather than spend time retraining every time the system is run. Once trained or loaded, it should be able to accept blocks of text, which it can then extract sentiment from, and return it.

As AI algorithms are trained on ”Features” rather than plain text, the Sentiment Analyzer must have some method of converting a sentence or paragraph into a set of features for the algorithm to use. Following on from a tutorial by *sentdex* [11], it appears that a good way to turn the text into a set of features is to use a dictionary, where the keys are the most common words from the training set, and the value is a True or False, for if the word is contained in the sentence or paragraph given. The potential issue with this is the loss of context, as separating the words completely will lose any negation, such as in the sentence ”This is not good”.

2.1.5 Search Tool

The Search Tool will be used by the user to search through the data for a particular Member of Parliament, to get their speech and its sentiment, or for a particular topic, to get the sentiment expressed about that topic.

The user would be presented with an option to search for either a Member of Parliament by name or a topic by title. It should then display the results of the search, showing examples of what was said about the topic, or if searching by MP, showing the topics they discussed and the things they said. It should also display an average sentiment expressed by that person or about that topic.

This average would have to be calculated in some way, likely by attributing a score of -1 for negative sentiment and +1 for positive sentiment. It would then likely be displayed using some sort of descriptive value rather than just the number; such as *Neutral* for a score around the 0 mark, or *Very Positive* for something close to 1.

2.2 Data Design

As parts of the system were designed to modify the layout of the original data at times, the formatting of said data had to be designed as well.

2.2.1 Parsed Data

The original Hansard Data was a set of XML files generated from the official Hansard Report. Some of the files referring to older reports, such as those from the 1800s, appear to have been

scanned in using some form of Optical Character Recognition, and thus contained some slight errors in some of the text. This also means that the data is not structured as well as might be expected from XML data.

As can be seen in figure 2.3, the data files contain data that is not useful to the project. For instance, there are references to images that were not included as a part of the download, presumably a scan from the original written report, though this is not made clear. In addition, there are ¡p¿ tags, representing paragraphs, which include ID numbers that are also not needed by this project.

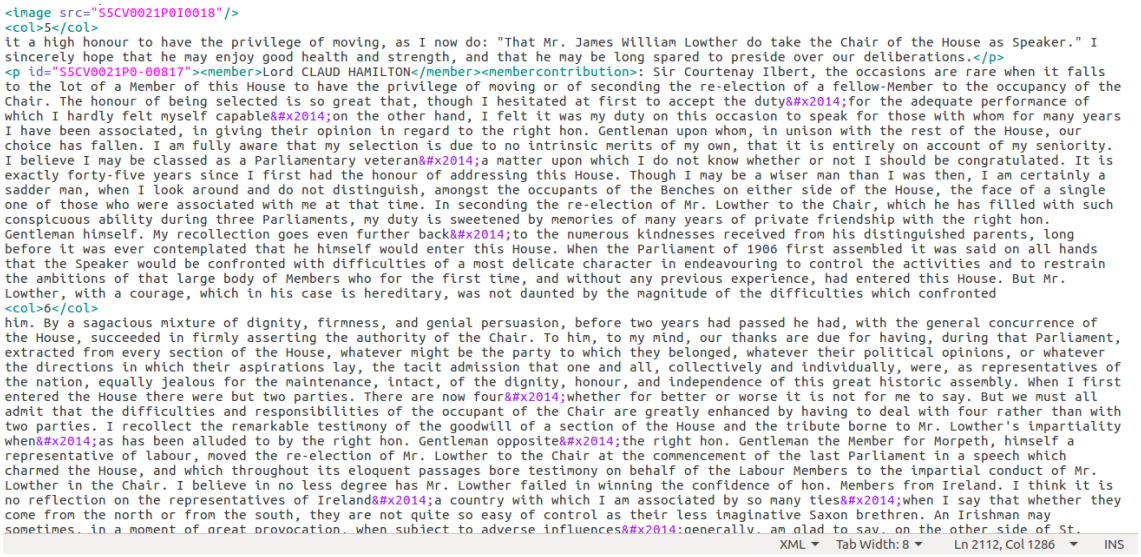

Figure 2.3: A sample of data from the file *S5CV0021P0.xml*, a file from the fifth series of data

For the parser, the format of the data files that it produces was originally designed as shown in Algorithm 2.1. The ellipses represent repeats, so inside the *Date* tags, multiple *speech* tags, all

```
<date dateformat="1984/04/13">
    <speech>
        <member>memberName</member>
        <topic>topicTitle</topic>
        <stance>POS/NEG</stance>
        "Actual Text of Speech would go Here"
    </speech>
    ....
</date>
....
```

Algorithm 2.1: Earlier version of the Parsed Data design

formatted like the one shown, can exist. A file may also contain multiple *Date* tags.

This, however, was causing some issues during development based around the size of the produced files, which were growing to thousands of lines long and several Gigabytes in size, as each entire series of data was being parsed into the same file. In order to combat the slow speeds of accessing such large files, and other issues caused by the layout, the data format was changed. Now, each

unique date found within the source is saved in a separate file, and within this file the data was formatted in its final form. Each date file is an XML file, the structure of which can be represented as a tree, shown in Figure 2.4.

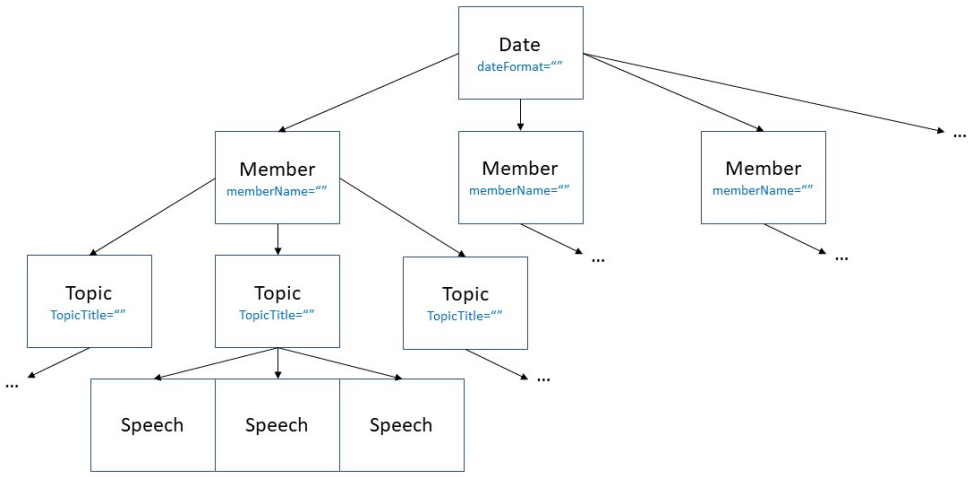

Figure 2.4: Diagram showing the structure of the parsed data, in XML. The *Date* tag is the root of the file

Each file can contain multiple member tags, which each have the member of parliaments name as an attribute. Each member tag can contain multiple tags for the topics they discuss, each of which have the topic title as an attribute. Each topic tag can have multiple speech tags, each speech tag containing the verbatim copy of what the member said about the particular topic.

2.2.2 Annotated Data

The formatting of the annotated data files was also designed at this stage. It was decided that the files would be saved as CSV files (comma separated values), which is a simplistic database style where a line in the file represents a single record, each value of the record being separated by a comma, hence the name. The layout was designed as the following:

Sentence, Sentiment, Member, Topic

As shown, it was decided that the best way of annotating full speech was to split it by sentence, so that each sentence would be annotated with its own sentiment. This would simplify the training process for the machine learning algorithms, as it was thought that trying to train it on too large a block of text would result in sub-par results, as these larger blocks of text can often express multiple sentiments interwoven into the speech. Attempting to train on, or even annotate, these larger more complicated blocks of speech would mean having to decide what the most important sentiment expressed would be, or work out the average sentiment. Due to this, any AI algorithm attempting to train on this would be training on data that expressed a varied sentiment, rather than a more polarized sentiment expressed in a single sentence. Of course, it is still possible that a single sentence might express positive sentiment towards one thing and simultaneously a negative sentiment towards something else, but dealing with these issues would take a lot more work than can be done on this project.

2.3 User Interface

Due to the complexity of the project, it was decided that the system would only be interacted with via the command line. This meant that, beyond basic menus, no GUI had to be designed for use. It was though that, should there be time during or after the project development, a GUI could be designed which worked with the existing functions, but it was not considered system critical to have a GUI.

The search tool described in Section 2.1.5 is designed to be the part used by most users, and so a mock-up of the command line interface is shown in Figure 2.5

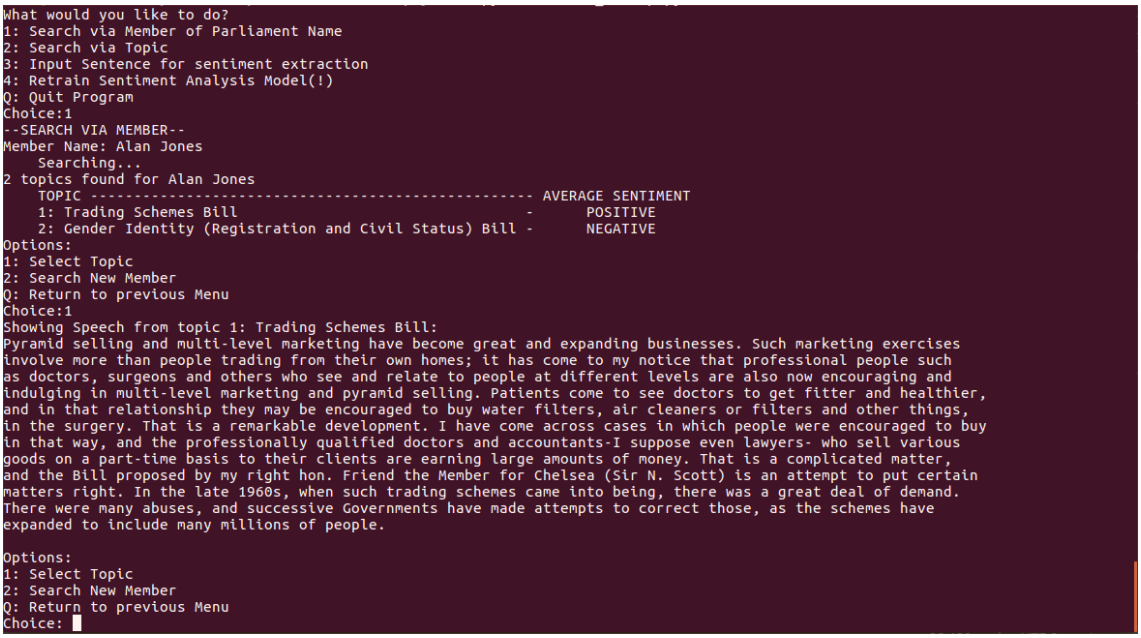

Figure 2.5: A mock-up design of the Search tool, which would be the main user interface for the project

2.4 Algorithm Design

2.4.1 AI Algorithm Choice

There are many potential algorithms that may be chosen for this project. It may prove difficult to choose one, or more, without simply trying them. The design calls for the use of Naive Bayes at first, mainly as its one provided by the NLTK package. However, multiple other algorithms may be tested, and the results compared to decide on the final choice. Should multiple algorithms prove useful, a voting system could be implemented, which applies each algorithm, and they cast a vote to what sentiment they calculate the text as. The votes would then be counted and the result shown.

2.4.1.1 Naive Bayes

Naive Bayes is a supervised algorithm with uses the statistical analysis of Bayes Theorem as to classify data. It treats each feature as completely distinct, which isn't accurate for NLP, but will may suffice for development of the system.

Bayes Theorem provides a way to calculate the probability of something occurring, for this project text being positive or negative, given its prior probability, and the probability of it given the Data provided. [12]. Text being classified as one thing or the other is usually called a *Hypothesis* in this regard. The mathematical representation of Bayes Theorem can be seen in Figure 2.6.

$$P(h \mid D) = \frac{P(D \mid h)\, P(h)}{P(D)}$$

Figure 2.6: Bayes Theorem

Where $P(h \mid D)$ is the probability of the Hypothesis (in the case of this assignment the prediction that a sentence is either Positive or Negative) holding true given the data provided. $P(D \mid h)$ is the probability that the data is true, given the hypothesis; and $P(h)$ and $P(D)$ are the probabilities of the Hypothesis and the Data respectfully.

$P(h)$ is usually called the *Prior Probability* of the Hypothesis, and reflects any background knowledge about the chance that the hypothesis is correct. In this project, this is likely to represent any form of bias in the training data, in that the prior probability of text being classified as positive would be equal to the percentage of training data that is labelled positive.

The Naive Bayes Classifier uses Bayes Theorem in a learning step in which the various probabilities of a class, such as *positive* or *negative*, are estimated, given the different attributes or the data [12]. These estimations correspond to the learned hypothesis, which is then applied to the new instances of sentences to classify them.

2.4.2 Parsing

The Data Parser needs to find and extract useful speech from the original data. Though said data is somewhat disorganized, in that speech can exist down almost any route through the XML tree and occur at any level, there are exploitable patterns that the Parser can use to find as much speech as possible. The overall method is documented in the pseudo-code Algorithm 2.2:

This shows that, so long as the parser can find a tag in the source XML for the date, all other relevant information can be found in relation to the position of this date tag in the XML. It also shows the method of ensuring all speech by one Member of Parliament is correctly attributed to them, to avoid repeated mentions of the same MP in the same file.

2.5 Design Conclusion

Now that the design of all the major parts of the project had been finalized, this report can now move on to the implementation section, to discuss how implementing these designs went, what

```
for EACH FILE
    for EACH DATE TAG
        if(DATE_FILE FOR DATE EXISTS)
            LOAD DATE_FILE CONTENTS INTO DATE_XML
        else
            CREATE FILE
            CREATE DATE_XML
        for EACH CONTRIBUTION in DATE_TAG
            GET CONTRIBUTION PARENT_TAG
            GET MEMBER NAME AS CHILD OF PARENT_TAG
            GET TITLE AS SIBLING OF PARENT_TAG
            GET SPEECH FROM CONTRIBUTION
            if MEMBER HAS BEEN SEEN BEFORE in CURRENT DATE_FILE
                if TOPIC HAS BEEN DISCUSSED BY MEMBER BEFORE
                    ADD SPEECH TO TOPIC
                else
                    CREATE NEW TOPIC
                    ADD SPEECH
            else
                CREATE NEW MEMBER TAG
                ADD TOPIC TO MEMBER TAG
                ADD SPEECH TO TOPIC
            ADD SPEECH TO DATE_XML

    SAVE XML TO DATE_FILE (OVERWRITE)
```

Algorithm 2.2: Data Parser Pseudo-code

became more difficult than expected and what, if anything, had to be changed, as FDD means that once a design is completed, it can still be modified and changed depending on implementation.

Chapter 3

Implementation

3.1 Data Downloading

One of the first parts of the system to be implemented was the Data Downloader. This was done first in order to get the full dataset downloaded as fast as possible, and also as a form of Python practice.

As mentioned in Section 2.1.1, the files that needed to be downloaded from the online archive had a specific format that made finding all the files required easier. Each series had a specific URL, and all files for that series would be stored as children of that URL in zip files. The naming pattern for the files was as follows:

S*s*V*vvvv*P*p*.zip

where *s* is the series from 1 to 6, *vvvv* is 4 digits representing the volume of the series, starting at 0001, the max value depending on which series it is a part of, and *p* is a part number, as some files were split into multiple parts. This could be zero, if the file is not split, or a one or two if it has been split. For instance, *S6V0141P1.zip* would represent the *1st* part of the *141st* volume in series *6*.

There was one difference to this pattern, discovered during development. Series 5 of Hansard was separated into records for the *House of Commons* and the *House of Lords*. For these, the letter *C* or *L* was added before the V to differentiate the two series. Afterwards, series 6 also contained the letter *C* to show that it was also only for the house of commons. So, for Series 6, volume 141, part 1, the file format would actually be *S6CV0141P1.zip*.

3.1.1 Online Connection

Python has a commonly used library for HTTP connections called *Requests*. It provides an API (*Application Programming Interface*) allowing easy connection to a provided URL, simply by providing a URL. Using this API, the downloader can check to see if a file exists, using the code snippet Algorithm 3.1.

Effectively, this code snippet loops through the files following the described pattern, and gets the status of the constructed URL. the *header* part returns a string that describes the type of file re-

```
for i in range(0, 100):  # change depending on how many files it
    looks like exist
    for j in range(0, 4):  # just to make sure. Not seen any
   files with P3 at the end but you never know
        temp_url = '{}{}/{}'.format(url_base, url_add[series-1],
    file_name_format.format(series, i, j))
        print("Checking: {}".format(file_name_format.format(
   series, i, j)))
        r = requests.get(temp_url)
        status = r.headers['content-type']
# if the zip file exists, the type will appear like this:
        if status == 'application/zip; charset=utf-8':
            print("FILE FOUND")
            zip_ref = zipfile.ZipFile(io.BytesIO(r.content))
            zip_ref.extractall(save_loc)  # extract the file
            zip_ref.close()

        else:  # if the zip file doesn't exist an error page is
   returned instead
            print("FILE NOT FOUND")
```

Algorithm 3.1: Snippet of the Downloader Code

turned, as if the file does not exist, it still returns a 404 error page rather than not returning anything. If the file does exist, however, it returns a string that says it is a ZIP file type (*"application/zip"*).

If the file does exist, the downloader also makes use of a library called *zipfile*, which allows it to handle zip files internally. It can then extract the data file from the compressed zip file, and save it to the designated directory on the local machine.

3.1.2 Dataset Size

The total Historical Hansard Dataset that was downloaded by the project is *14.4GB*, and consists of *2503* files, demonstrating the need for the automated downloading. Due to this size, the dataset could not be saved to the University file system, and so was saved to an external hard drive instead. This meant that some parts of the later development were slowed due to the USB 3 connection used by said hard drive.

3.2 Data Parsing

During development of the system, the Data Parser and associated methods were where the most time was spent, due to many unforeseen complications.

3.2.1 Name Recognition and Disambiguation

Part of the difficulty in parsing the data accurately was ensuring speech spoken by one person was always attributed to that person, even though the references to that person may use multiple versions of their name. In order to correctly attribute speech to the speaker, without duplicating references to them, a method of NLP called ”Name Disambiguation” must be employed.

Name Disambiguation is the method of identifying proper names in text, and recognizing when two proper names refer to the same subject or person. [13] Many difficulties exist for such a task, as names can come in many forms. Proper nouns can often refer to multiple things. *Johnson and Sons* might refer to a business by that name; two people with the surnames *Johnson* and *Sons*; or someone called *Johnson* and their male children. In this example, the context of the sentence the name appears in may provide clues to which it is, but this is not always an option.

Thankfully, this project only needs to be able to recognize the names of the Members of Parliament mentioned in the Hansard Dataset, which displays more structure in the way it names people than normal human speech would.

```
<section>
<title>Disabled Persons (Air Travel)</title>
<p id="S6CV0001P0-00373">1. <member>Sir David Price</member> <membercontribution>asked the Secretary of State for Trade, in view of the fact
that 1981 is the International Year of Disabled People, if he will take steps with British airports and British airlines, respectively, to
improve facilities for air travel by disabled pepole, especially those confined to wheelchairs.</membercontribution></p>
<p id="S6CV0001P0-00374"><member>The Under-Secretary of State for Trade (Mr. Reginald Eyre)</member><membercontribution>: The provision of
facilities for disabled air travellers is a matter for the airports and airlines concerned, and a good dealhas been, and is being, done to
improve the lot of the handicapped when they travel by air. Airport and airline representatives have participated in the meetings reviewing the
special meeds of disabled passengers held under the auspices of my right hon. Friends the Secretary of State for Transport and the last
Minister for Social Security. We encouraged the widest possible airport and airline representation at last Thursday's conference on "Transport
without Handicap".</membercontribution></p>
```

Figure 3.1: An example from the original dataset, showing that names exist only within *member* tags

As can be seen in Figure 3.1, the names needed for this project are always contained within a *member* tag. They still contain the full job title at times, but this means that the area the project needs to look for a name is well defined. It can be guaranteed that, if a member tag is encountered, it will contain the name of the member.

The first attempted method of extracting the name from the *member* tag was using the *Named Entity Chunker* from NLTK, which is designed to extract the Named Entities from a piece of text and return them in a list. Using this, the name could be extracted, potentially along with other parts of the title (*The Under-Secretary of State for Trade (Mr. Reginald Eyre)* would be returned as a list of two names, *State for Trade* and *Mr. Reginald Eyre*). The thought was that the actual name desired could be chosen from that list by looking for the item that started with an Honorific, such as *Mr.*, *Mrs.* and other such titles. However, the Named Entity Chunker did not work consistently enough for it to be a viable solution for the name extractor, as it often separated the honorific from the name. Due to this, it could not be reliably used to extract the names, because there was no way to recognize, when it returned multiple names, which referred to the actual person, and which was a part of their job title.

The second method attempted, which was the one settled on for the project, was the use of Regular Expressions to search for the expected name format. This method relies on a lot of assumptions about the organization of the original dataset, but appears to be good enough for the current dataset. The assumptions are as follows:

- All Names start with an Honorific (Mr, Mrs, Sir etc)

- All Names End with a Surname
- The first time a person is seen, their full name and job title, if relevant, are presented

Following these assumptions, Regular Expressions could be used to find the honorific at the start of the name, then get everything following it until either the end of the text, or a piece of punctuation not commonly used in names, such as a bracket.

```
31  def extract_name(text):
32      # find a name in a sentence or piece of text
33      # will assume the first name given that includes an honorific (mr., Mrs, etc) is the correct name
34      punctuation = string.punctuation  # we want to get all the text from Mr/Mrs etc till it finds some punctuation
35      punctuation = punctuation.replace("-", "")  # however, - and . are valid parts of a name, so we still want them
36      punctuation = punctuation.replace(".", "")
37      name_rex = re.compile(r'(\bMr[ .]|\bMrs[ .]|\bSir[ .]|\bDr[ .])[^{}\\\n]+'.format(punctuation), re.IGNORECASE)
```

Figure 3.2: Extract of code from the Name Extractor Method.

Figure 3.2 shows the regular expression used to extract a name from a piece of text. It searches for an Honorific to start it off, either *Mr*, *Mrs*, *Dr* or *Sir*. Once it finds one of these honorifics it gets everything after it that is not a punctuation mark listed, as shown by the `[^{}\\\n]+` part at the end of the regular expression. The `{}` brackets in that part are replaced with the string called *punctuation*, generated by the string package, so that it becomes a list of all punctuation possible, including new line characters.

3.2.2 Name Matching

Once names can be reliably extracted from the source text, a method of comparing two names to see if they referred to the same person had to be developed. This becomes a very complicated issue, simplified only slightly by the assumptions mentioned previously.

Name One	Name Two	Same Person?
Mr. Adam Neaves	Mr. Adam Neaves	True
Mr. A. Neaves	Mr. Adam Neaves	True
Dr. Adam Neaves	Mr. Adam Neaves	False, wrong Honorific
Mr. B. Neaves	Mr. Adam Neaves	False, wrong First Name
Adam Neaves	Mr. Adam Neaves	True, despite missing honorific
Mr. A. B. Neaves	Mr. Adam Neaves	True, though slightly ambiguous
Mr. A. B. Neaves	Mr. Neaves	True, though again ambiguous

Table 3.1: Table showing the different ways names might or might not refer to the same person

For instance, as shown in Table 3.1, first names might be shortened to just an initial, or even fully removed. Honorifics are not guaranteed to be present, but are very important if they are, as two different names may differ from only that.

The implemented algorithm for this project follows the same assumptions mentioned in Section 3.2.1, and is designed to allow for some false negatives (wherein two names that *do* refer to the same person might not appear as the same person) to avoid any false positives at all. This is because it was decided that speech accidentally attributed to the wrong person would be worse than accidentally having duplicate references to a person, since a human reader can likely see the connection between *Mr. Adam Neaves* and *Mr. A Neaves*, but there would be no way for them to

know that some speech was attributed to the wrong person. The basic algorithm implemented is as shown in Algorithm 3.2.

```
if BOTH NAMES ARE EXACTLY THE SAME
    return True
SPLIT BOTH NAMES INTO COMPONENT WORDS
for EACH NAME
    if NAME CONTAINS 3 WORDS
        FIRST WORD IS HONORIFIC
        SECOND WORD IS FORENAME
        THIRD WORD IS SURNAME
    else
        FIRST WORD IS HONORIFIC
        LAST WORD IS SURNAME
        IGNORE ANYTHING ELSE
if FORENAME ONE and FORENAME TWO BOTH EXIST
    FORNAME_MATCH = FORNAME ONE == FORNAME TWO
else
    FORNAME_MATCH = True
SURNAME_MATCH = SURNAME ONE == SURNAME TWO
HONORIFIC_MATCH = HONORIFC ONE == HONORIFIC TWO

if SURNAME_MATCH, HONORIFIC_MATCH and FORENAME_MATCH ARE True
    return true
else return False
```

Algorithm 3.2: Pseudocode representing the name matching method

This method does work so long as the assumptions in Section 3.2.1 hold true. It would not be a valid method of matching names in a more generalized setting where the formatting and appearance of names would vary wildly.

3.2.3 Saving Parsed Data

The parser would save the information extracted from the original data files in newly created XML files. Due to some character use in the original dataset, these files had to be opened by the Python script in *Binary Mode*, meaning it would treat the data as bytes of data rather than strings of text. Due to this, when editing a file, the Parser could not simply append data to the file, but rather would have to overwrite the whole file. Therefore, when editing a parsed file to add more data to it, it was necessary to load the whole file into memory, edit that data, then overwrite the whole set of data to the XML file again. This was one of the main reasons for the change in data structure part way through implementation described in Section 2.2.1. When the Parser was designed to write an entire series to a single file, it would take an increasingly long time to read the whole file into memory, then save the data back to the file once modified. At the start of a session of parsing, it would take an average of *3 seconds* to parse the first file from the original dataset, but as it parsed more of a series, each additional file would increase the time it took substantially. Table 3.2 shows the time taken on a subset of one series.

Source Data File Number	Time to Parse (s)
0001	2.564
0002	3.245
0003	4.632
...	...
0250	302.452
0251	306.321

Table 3.2: Excerpt from the log of the original parser parsing Series 6, showing time it took to parse each file

As can be seen, the additional time taken for each file meant that, in order to parse an entire series, the parser would take up to a few hours to finish. Additionally, as the large files had to be loaded into memory in order for them to be edited, the Parser would use a large amount of RAM, at one point during development getting up to around 10GB of RAM used.

Therefore, instead of saving an entire series worth of data to a single file, the decision was made to split the files by date instead. The parser would search through each file, creating a new file for each date it encountered, so that everything that was debated on that date would be saved to that file. This meant that the parsed data files were smaller, and meant that the data that had to be loaded into memory in order to be modified was much smaller, so the Parser didn't use anywhere near as much RAM. It also meant that the time taken per file parsed was constant, and only depended on the size of the original file, rather than the number of files already parsed. This was a much better solution, and also encouraged a better design for the parsed data layout, as described in Section 2.2.1.

Additionally, though the parsed data consisted of more files than the original dataset, the set of parsed data was usually smaller than the original, due to the parser stripping out unneeded information. The exact values can be seen in Table 3.3

	Original Dataset		Parsed Dataset	
Series	**Size (GB)**	**Number Of Files**	**Size (GB)**	**Number Of Files**
1	0.0943	27	0.0691	1245
2	0.0877	22	0.0755	926
3	1.4	305	1.2	6888
4	0.6461	133	0.3842	1551
5	5.5	900	3.4	9742
6	3.5	446	1.7	3745

Table 3.3: Size comparison between the Original Hansard Dataset and the Parsed Dataset generated by the Parser

3.3 Manual Annotation Tool

With all the source data successfully parsed, the Manual Annotation Tool (MAT) could be developed to use that data to create the training dataset discussed in Section 2.2.2.

Originally, as discussed in Section 2.1.3, this tool was designed to print everything a single MP said about a particular topic, but this often meant far too much text was printed on the screen at any one time. Additionally, this tool was also designed to allow the user to edit the name of the MP, or the topic, in case of typos in the text. However, it was felt that the tool was overcomplicated by this added functionality.

Instead, the Annotation tool now only displays one sentence at a time, and allows the user to quickly annotate it by entering only a single character, as shown in Figure 3.3. That helped to speed up the process of annotating data, which was already a lengthy process, in order to create the training set needed for the Artificial Intelligence Algorithm.

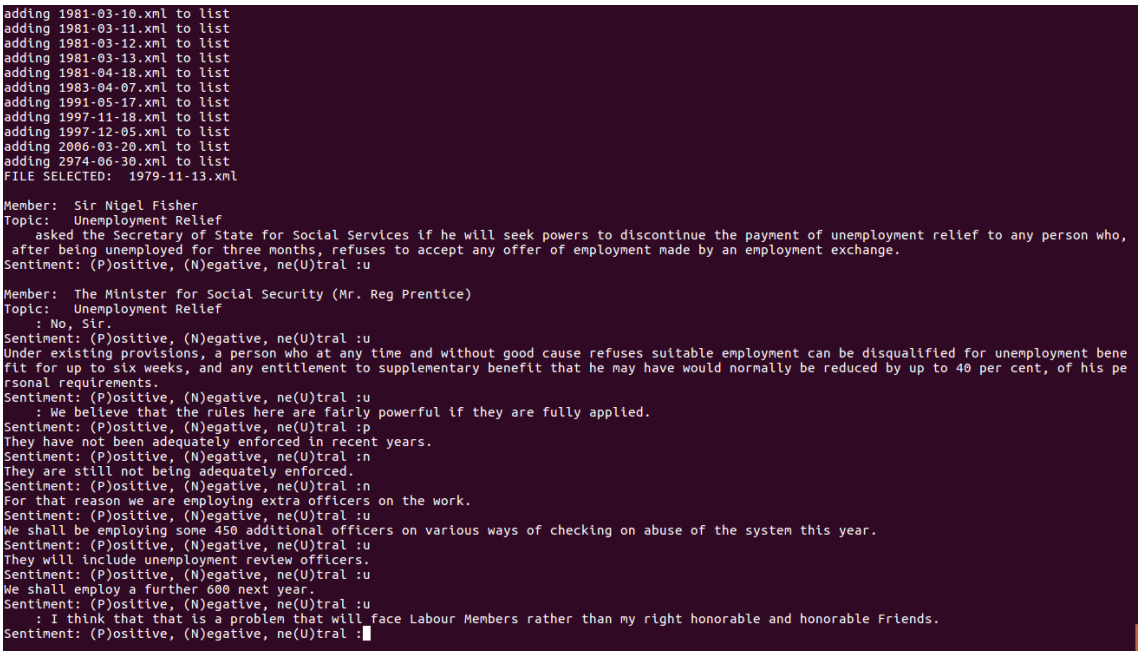

Figure 3.3: Screenshot from the Annotation Tool running on the Command Line

3.3.1 Sentence Splitting

An important part of the update to the Annotation tool was the ability to separate a block of text by sentence. This is not as simple as it sounds, as the text can't just be split whenever a "." character is encountered, as there are acronyms in some of the text, and some sentences might include a quote with a period in it, which must be included in the sentence structure.

NLTK does include a sentence tokenizer as a part of its library. This method is designed to receive a block of text as an input, and returns an array of sentences, split by the tokenizer. However, this function did not work perfectly for the data used, as there were a few special cases, such as typos, that the tokenizer provided was not prepared for. In order to improve the quality of the splitting, some parts of the text had to be modified, so that the tokenizer would either stop seeing sentence boundaries where there were none, or missing sentence boundaries. These changes were as follows:

- The word *Honourable* had to replace the word *Hon.*, which was often used as a short hand.

NLTK's tokenizer did not recognise this and therefore split sentences on the period of that word.

- If the Members of Parliament discussed some form of percentage, it was commonly referred to as *per cent.*, which the sentence tokenizer saw as the end of a sentence due to the period. This was replaced with the word *percent*.
- The Parser would strip out any new line characters from text before saving it. This, combined with issues from the original data, meant that when a sentence ended, it was common for it to miss out any sort of space between the period that marked the end of the sentence, and the start of the next sentence. Due to this missing space, the tokenizer wouldn't realize that it was supposed to be a sentence boundary and therefore wouldn't split the text. Regular expressions were used to find areas of text where it looked like this had occurred, and would input the space.

Following these corrections, the code that performed the sentence splitting became that shown in Algorithm 3.3.

```
def sentence_split(text):
    # because NLTK's sentence tokenizer recognises hon.
    # as the end of a sentence, unlike mr. or mrs., we need to
    # replace that with something it can handle
    regex_hon = re.compile(r'\bhon\.', re.IGNORECASE)
    regex_percent = re.compile(r'\bper cent\.', re.IGNORECASE)
    # some sentences are missing the space. Add it back in.
    regex_sentence_end = re.compile(r'([a-z])\.([A-Z])')
    text = re.sub(regex_hon, "honorable", text)
    text = re.sub(regex_percent, "percent", text)
    text = re.sub(regex_sentence_end, r'\1. \2', text)
    text = text.replace('\n', ' ')
    # once all regex is done, send text to the tokenizer
    return sent_tokenize(text)
```

Algorithm 3.3: Sentence Splitting method, with included regular expressions

As can be seen by this function, each replacement required is handled using regular expressions, which allow it to quickly search the whole text given to it for these issues described, and fix them, before the text is then sent to the sentence tokenizer of NLTK to be split into an array. This method works well, though it is not the most maintainable, as it would have an issue if many more special cases were found.

3.4 Sentiment Analyser

The sentiment analysis tool builds upon the methods provided by NLTK. The tool provides a method to train the AI algorithm, and also methods to analyze the sentiment for sentences and paragraphs using the trained algorithm. Though NLTK does provide a sentiment analyser as part of its package, this was a pre-trained algorithm that had been trained on a Movie Review corpus.

It was therefore expected that such an algorithm would not be suitable for this project, as no way to retrain on custom data was found. Therefore, a custom sentiment analyser was created using the Naive Bayes Algorithm.

This Sentiment Analyser was first trained on only a small set of training data designed only to be enough to allow development. Without a small dataset, the Sentiment Analyser module could not be developed, as there would be no way of testing that it functioned without anything to base the AI model on.

3.4.1 Naive Bayes Classifier

Naive Bayes, as described in Section 2.4.1.1, was the AI algorithm provided as a part of the NLTK library.

3.4.1.1 Creating Features

Features, in the case of AI training, are a representation of the thing an AI is trying to learn to predict [12]. For instance, if an AI algorithm was being trained to predict the weather based off current conditions, the current conditions would be the set of features, each separate condition such as current temperature, barometric pressure, humidity etc. would be a feature. In the case of a supervised algorithm, such as Naive Bayes, it is provided with a list of features, and the class that is attributed to that set of features. So, for the example given, Table 3.4 would be an example set of training data.

Temperature	Pressure	Humidity	Weather Class
Low	Medium	High	**Storm**
High	Low	Low	**Sunshine**
High	Low	High	**Storm**

Table 3.4: Weather training data as an example of features

In order to be able to use the training data to train the algorithm, the data must first be turned into a set of *Features* that the algorithm could use to base its hypothesis on. Following the Tutorial by **Sentdex** [11], it was decided that the feature set would be the set of 3000 most commonly used words from the training dataset, after removing *Stop Words*, which are words too common in the English language to be of any use to the classifier, such as *The*, *for*, or *to*. The set of features would be represented by a *Key Value dictionary*, where the word acts as the unique key for each value, and the value is a Boolean, representing whether or not the word is present in the text. This method has its limitations, discussed in Section 5.1.3. However, it does limit the effect of uncommonly used words, as they won't be a part of the feature set and can therefore be ignored by the algorithm.

3.4.1.2 Training the Algorithm

To train the algorithm, the set of features created is first split into a set of training data, and a set of testing data. The algorithm is trained on the training data, using the combination of features

and annotated classes, then it classifies the testing data using the hypothesis generated by the training. The accuracy of the algorithm is then represented as the percentage of these training cases correctly classified.

For this project, the set of annotated features was shuffled, so that they would be in a random order, in an attempt to reduce the changes of bias. It then, in the original iteration of the project, split the set of feature/annotation pairs so that the first 90% were the training data, and the remaining 0% were the testing set. This, however, often gave very mixed results, in part due to the randomization, as the resulting AI model would have a claimed accuracy of anywhere between 40% and 90%.

Instead, it was decided to use *K-Fold Cross Validation* [14], in which the set of annotated features was split into K chunks. For each chunk, the algorithm was trained on the rest of the data, and tested on the single chunk, as shown in the diagram of Figure 3.4. It was then trained on the entire set of features, the model from such training was the one used and saved.

Round One	Test	Train	Train	Train	Train
Round Two	Train	Test	Train	Train	Train
Round Three	Train	Train	Test	Train	Train
Round Four	Train	Train	Train	Test	Train
Round Five	Train	Train	Train	Train	Test
	Chunk One	Chunk Two	Chunk Three	Chunk Four	Chunk Five

Figure 3.4: Diagram showing how K-fold validation works, when $k = 5$. Rows represent training rounds, columns are the chunks of data.

The partitioning of the data was done after the feature list was shuffled to ensure the chunks were random. The code that trained and tested the model is in Algorithm 3.4. the variable *fold* is the number of folds to make, and the variable *features* is the list of annotated features, having been

randomly shuffled.

```
1
2 print("Splitting feature set into {} folds".format(fold))
3 folds = numpy.array_split(features, fold)  # split into folds
    for training
4 # for each fold, train it on all other folds, test it on this
    fold
5 for i, test_fold in enumerate(folds):
6     train_folds = folds[0:i] + folds[i+1:fold]
7     train_folds = [item for sublist in train_folds for item in
    sublist] # flatten list
8     print("Fold {}. Training on {} Instances.".format(i+1, len(
    train_folds)))
9     classifier = NaiveBayesClassifier.train(train_folds)
10    print("Fold Accuracy: {}%".format(nltk.classify.accuracy(
    classifier, test_fold)))
```

Algorithm 3.4: Snippet of code that trains and tests the classification algorithm

3.4.1.3 Testing Results

The algorithm was trained and tested using both *10-fold* and *5-fold* validation. Table 3.5 and 3.6 show the results from this training and testing. The total number of annotated sentences was 201.

Fold Number	Training set Size	Reported Accuracy (%)
1	160	85.366
2	161	65
3	161	72.5
4	161	65
5	161	67.5

Table 3.5: Table of results from training the Naive Bayes algorithm using 10-fold cross validation.

As can be seen by these tables, the average accuracy when trained using 5-fold validation was *71.1%* and when using 10-fold validation it was *70.1%*. The similarities might well be due to the small number of annotated sentences that were used in training. The problems that caused this are discussed in Section 5.1.2.

To compare the results, a baseline called *ZeroR* [15] was used. This was used instead of a baseline of random guessing, as it takes into account potential bias in the annotated data. Effectively, *ZeroR* classifies all instances as the most common class, hence the name *Zero Rule*. In the annotated data used, *106* of the sentences are classified as Positive, and *95* are classified as Negative. This means that ZeroR classified all of the instances as Positive, as that is the class with the most instances. The maths of working out the probability of random guessing is shown in Figure 3.5, showing that random guessing gives a theoretical accuracy of **50.1%**, whereas the Zero Rule baseline has an accuracy of **52.7%**. Whilst this difference might not be a large one, it helps improve the analysis

Fold Number	Training set Size	Reported Accuracy (%)
1	180	66.666
2	181	75
3	181	85
4	181	70
5	181	65
6	181	60
7	181	70
8	181	60
9	181	80
10	181	70

Table 3.6: Table of results from training the Naive Bayes algorithm using 10-fold cross validation.

of the Naive Bayes algorithm.

$$(P(ClassIsPositive)*P(ClassAsPositive))+(P(ClassIsNegative)*P(ClassAsNegative))$$

$$(0.527 * 0.527) + (0.473 * 0.473)$$

$$0.501$$

Figure 3.5: Probability of correct classification based on random selection. $P(X)$ is the Probability of X

3.4.1.4 Saving the Model

In order to be useful, the algorithm could not be retrained every time. As the training dataset grows, the time taken to train the algorithm also grows, so it would be inefficient to retrain each time. Thankfully, a python module called Pickle allows for the saving of Python classes as a file on the computer. This module can allow the trained model to be saved and loaded from a file created by the pickle module, and avoids the need to retrain it each time the software is run.

3.5 Implementation Conclusion

Moving on from implementation, this report will now move on to discuss the methods of testing used during and after development, as well as discussing any areas where more rigorous testing, or just more testing in general, would have been useful.

Chapter 4

Testing

4.1 Overall Approach to Testing

The main tactic for testing was to use Unit Tests. Unit testing involves breaking the software being tested down into pieces, called *Units*, and then testing each piece *In Isolation* [16]. The isolation is important, as it ensures that any failing tests are caused by the code being tested, rather than some other part of the system that might not be working. This isolation also means that tests should be able to be run in any order, and therefore makes sure that the code and test are written to avoid *Side Effects*, where running a test might affect another part of the software.

For these reasons, a package for Python called Unittest was used for writing the tests. It provides a framework for testing, including methods for *Setting up* and *Tearing Down* before and after the tests run. The set up method ensures that, before the tests are run, anything that needs to be initialised can be. The *Tear Down* method is used to make sure that, after the tests are run, the state of the system has been returned to how it was before testing. Using these two methods, as well as the other parts of the *Unittest* module, the unit tests can be run easily, and repeatably, without causing tests that pass to start failing, or vice versa.

4.2 Unit Tests

A framework to implement Unit testing for each module was created. However, due to time constraints, the cause of which is discussed in Section 5.1.1, not all tests were implemented. Each suite of tests was created as a python script, named *test_XXX* where *XXX* is the name of the module being tested. Each test script would import the *unittest* module, as well as the module to be tested.

In order to run the tests, a package called *py-test* was installed. When running this package from the command line, it can search the directory for the suite of tests, and set up a virtual environment for the tests to run in, to ensure that references to the different modules would still work. This was used as the test scripts were stored in a separate folder, for organizational sake, but this meant that the references to the modules didn't work when trying to run the test scripts as normal python scripts.

4.2.1 Natural Language Processing Module

The only module for which a reasonably full set of tests was written was the Natural Language Processing module. This module is the one which contains all the NLP methods required by the system, including the Sentence tokenization, and name matching methods. A test suite for each method was created, and a set of tests were created that tested not only normal functionality of the method, but tested edge cases that could cause the method to crash. For instance, leaving a string empty when the method expected contents.

4.2.1.1 Name Extraction

One of the methods tested was the Name Extraction method. This method was designed to accept a string as input, which contained a name and potentially other things, such as job title, and return a new string that was just the name.

Most of the tests for this method provided a string with a name, and then compared the returned string with one that represented the expected result. Additionally, a test was written that tested what happened if the method was given an empty string, or a variable that was not a string.

4.2.1.2 Name Matcher

The name Matching method compares two strings that contain names, and returns a boolean value of *True* if the names appear to refer to the same person, of *False* if not.

This method received extensive testing, due to the complexities of this task. Table 3.1 in Section 3.2.2 shows the sort of things that must be tested for. Most of these tests used two strings, each representing a name. The tests would assert if the function returned the expected boolean value. The tests included matching names of the exact same format, some where the forename was absent from one name, and some where the only difference in the name was the honorific used. Additionally, a test for each possible edge case, such as empty strings and non-string objects, were also written.

4.2.1.3 Sentence Splitter

The Sentence Splitter method took a paragraph of test as an input, and returned an array of strings, in which each string is a separate sentence. This follows the rules set out in Section 3.3.1.

The testing of this method involved a test string that contained one or more sentences, and an expected output to compare to the output of the method. It also tested all the special cases as mentioned in Section 3.3.1.

4.2.2 Parser Tests

Attempts were made to write tests for the parser module. However, comparing XML files can prove difficult, and in the time remaining on the project, no solution for this problem was found.

A test file was created, with known format and contents, and the expected output of this file was written. The plan was to compare these files with the one generated by the Parser after parsing the test file.

The only part of the Parser that was tested was the collection of files to parse. The parser had to create a list of references to each file it needed to parse. This functionality, when run on the test file mentioned, is tested by a single Unit Test.

4.3 Testing Conclusion

Now that the testing methods have been described, this report will now move on to critically evaluate the progress of this report. The next chapter will highlight the difficulties encountered with the development of the project, and what, with hindsight, could be improved or done differently in order to produce a better result.

Chapter 5

Evaluation

5.1 Requirements Comparison

Table 5.1 shows the comparison between what required features were set out in Section 2.1, and what features made it in to the developed system.

Feature Num	Required Function	Developed Function	Notes
1	Download the Dataset	Fully Developed	
2	Parse the Original data	Fully Developed	Was far more work than anticipated
3	Provide Ability to Annotate	Fully Developed	
4	Train AI on Annotated Data	Fully Developed	Had not annotated enough data
5	Search parsed data for MP or Topic	Not Developed	
6	Use AI to extract Sentiment	Partially Developed	Only uses naive bayes
7	Display comparison of sentiment	Not Developed	

Table 5.1: Comparison between required functions and the functions that were developed

As shown, the project did not manage to complete every requirement that was set out. However, this project did manage to develop an automated sentiment tool that could be trained on the Hansard records, then extract sentiment from sentences or paragraphs given to it, which was one of the main aims of the project.

5.1.1 Data Parser Difficulties

One part of the project that was a much larger part of development than expected was the Data Parser, the development of which is described in Section 3.2. As described in that section, parsing the dataset turned out to be a much more complex job than anticipated. Some difficulties came from a lack of prior knowledge on Natural Language Processing. It wasn't realised until a good part way through the development of the parser that Name Disambiguation, discussed in section 3.2.1 was such a complex topic of research, or that it would be such a crucial part of the parsing process. Due to the unexpected complexities of developing the parser, other parts of the project suffered. The *Search Tool* and the *Comparison Tool* were never developed, as the time planned for the development of those modules was instead spent on the Parsing Tool. Additionally, trying to understand the layout of the original Hansard Dataset, especially the earlier series, took a lot of time in the beginning of the project, due to a lack of formatting. Attempting to visualise the large and complex XML files caused a delay in the development of the Data Parser, as there was no way to develop the parser until the data structure was understood.

However, despite these difficulties, once they were overcome the Parser itself works very well. It can accurately pull the relevant speech out of any of the Historical Hansard data files provided, and parse it into files that are much easier to handle. The Name Disambiguation and matching functions, discussed in Section 3.2.1 and 3.2.2, might not work for more general name matching, but are tailored towards the Hansard data, and work very well for the project itself.

5.1.2 Annotating Data

It was expected at the beginning of the project that a large amount of time would have to be devoted to annotating data to train the AI algorithm that would be used. The tool developed, discussed in Section 3.3, was designed to make this job as easy as possible. However, despite attempts, it still took a large amount of time to annotate any data. Part of the problem is likely as the tool discards any sentences labelled as *Neutral*, since it was thought training on neutral data would not help the task at hand. However, A large part of the speech by Members of Parliament is neutral, when they are either stating facts, or just speaking neutrally. This meant that, if the tool was used to check through 100 sentences to annotate, up to around 40 of those sentences might be neutral, and thus discarded. This means that the training and testing dataset is too small to accurately train the AI algorithm. The tool itself is fully functional and works well for the purpose but, due to time constraints caused in part by the difficulties mentioned in Section 5.1.1, not enough data was annotated, and therefore the performance of the Sentiment Analyser suffered.

There is also a potential for bias in the data annotated. It's possible that, as the data was only annotated by a single person, that there could be some bias in what was claimed to be *Positive*, *Negative* or *Neutral*. It's possible that what was annotated as a *Negative* sentence, for instance, might have only been read as negative by the individual, and not actually spoken in a negative manner. This is partially an issue with written text, as it loses any nonverbal communication, such as body language, which is considered an important part of communication [17].

5.1.2.1 Outsourcing Data Annotation

It was realised too late into the project that there was not enough data, nor enough time to fix the issue. However, there are solutions available that could have prevented this issue. Should this project be restarted, a good idea would be to outsource the data annotation to other people. This would require ethics forms to be filled for every participant, but it would solve the issue of there not being enough time to annotate the data. It would also potentially solve the issue of bias in the annotations, if multiple people were given the same sentence the annotate. This does mean there would be a chance of conflict, if there are multiple different opinions on the sentiment of a single sentence, but these could be solved manually if needed, or automatically in the case of a single person having a different opinion than the rest. This would require more work to combine annotations from multiple sources, but it would replace the time spent doing the annotation without outsourcing.

5.1.3 AI Algorithms

For this project, the only AI algorithm used was Naive Bayes, as discussed in Section 3.4. As no other algorithms were tested, it's possible that Naive Bayes may not be the most useful algorithm for this project. Without other algorithms to compare it to there is no way to know. This issue, combined with the small training dataset, means that the Naive Bayes is inconsistent in its accuracy, as shown in Section 3.4.1.3.

Additionally, the feature set that was used may not have been ideal for the subject attempted. As discussed in Section 3.4.1.1, the feature set selected was a list of the top 3000 words used in the annotated data, paired with a boolean value to mark if the word was present in the sentence or not. This limits what can be learned from it, for two reasons. One, it removes the context for a word. This means that if a word that usually denotes a specific class, such as *Terrible* likely being negative, any form of negation, such as *not* or *never* loses any effect, as it is separated from the word itself. One potential way to combat this would be to pair up words that negate or exaggerate another when converting it into a feature, but this would require a lot of work to ensure edge cases were always covered.

The second issue with this method of creating features is that it loses the number of times a word is used, which may well be relevant to the sentiment analysis. The current method means that if a word, such as *Terrible*, is used only once, or many times, the sentiment extractor is not going to treat it any differently, as it will still only show "True" no matter how many times it's used. It would, however, still raise its position in the list that is ordered by frequency of words.

5.2 Future Work

If more time could be spent on the project, the following sections discuss what could be done to improve it.

5.2.1 Testing

It was planned that a set of tests for the whole project would be produced in order to ensure that the project behaved as expected. Unfortunately, due to time constraints, not many tests were actually written, and a portion of the project remain somewhat untested, beyond the usual manual testing done during production. One of the first things to do should more time be available would be the production of more Unit tests for the rest of the project.

5.2.2 More AI Algorithms

A good use of this additional time would be to implement more than just the one algorithm, and compare results. Should one algorithm prove better than all others, that one should be the one used for this project. If multiple have good results, a voting system could be implemented that ran the multiple algorithms, collected their results, and combined them to form one ”voted for” class for the data provided to them. This would require experimentation to check for accuracy and the best method for voting, but could potentially improve the consistency of the results, if not the accuracy.

5.2.3 Develop Search Tool

One of the major failing during development was that the proposed tool designed to search the parsed data for either a Member of Parliament by name, or a topic up for debate, was never developed. While the main aim of the project, to provide a way to train and use an AI algorithm to automatically extract sentiment from political speech, was completed, the use of this is less intuitive without a method to search the pre-existing data for sentiment. Therefore, it could be wise to use any additional time to complete this original project goal.

5.2.4 Work with Additional Data Sources

An interesting fact about the Hansard Report is the number of different forms it is offered in. The one used by the project is the Historical Hansard Dataset, as discussed in Section 1.1.1. However, one potential use for this project is to monitor the sentiment expressed by the politicians of today. Hansard is offered in a Daily format, where a report is released in the morning that documents the debates of the prior day. Modifying the Data Parser so that it may also parse these daily reports would give the project a lot more data to work with. Additionally, the Parliamentary website offers an Atom Feed for these reports, which could potentially be subscribed to by the project, so that when a Daily report is released, the parser could automatically download and parse this report, ready to be analysed by the rest of the system.

5.3 Project Conclusion

A great deal was learned from doing this project, especially from research done into Natural Language processing, and the development of a full project in Python. It is hoped that work can continue on this project after it has been submitted for the dissertation, and may eventually become a useful program for extracting sentiment from political speech.

Appendices

Appendix A

Testing Log

A log of the tests run by pytests. When the command `python3 -m pytest -v` is run from the command line, in the Python directory, this output is what is produced.

```
============================= test session starts ==============================
platform linux -- Python 3.5.2, pytest-2.8.7, py-1.4.31, pluggy-0.3.1 -- /usr/bin/python3
cachedir: .cache
rootdir: /home/adam/Documents/Hansard-Sentiment/Python, inifile:
collecting ... collected 25 items

tests/test_NLP.py::NLPSentenceSplitTestCase::test_sentence_split_contains_quote PASSED
tests/test_NLP.py::NLPSentenceSplitTestCase::test_sentence_split_hon PASSED
tests/test_NLP.py::NLPSentenceSplitTestCase::test_sentence_split_missing_space PASSED
tests/test_NLP.py::NLPSentenceSplitTestCase::test_sentence_split_normal PASSED
tests/test_NLP.py::NLPSentenceSplitTestCase::test_sentence_split_percent PASSED
tests/test_NLP.py::NLPSentenceSplitTestCase::test_sentence_split_single_sentence PASSED
tests/test_NLP.py::NLPNameExtractTestCase::test_empty_string PASSED
tests/test_NLP.py::NLPNameExtractTestCase::test_extract_name_dr PASSED
tests/test_NLP.py::NLPNameExtractTestCase::test_extract_name_mr PASSED
tests/test_NLP.py::NLPNameExtractTestCase::test_extract_name_mrs PASSED
tests/test_NLP.py::NLPNameExtractTestCase::test_extract_name_no_honorific PASSED
tests/test_NLP.py::NLPNameExtractTestCase::test_extract_name_only_name PASSED
tests/test_NLP.py::NLPNameExtractTestCase::test_extract_name_sir PASSED
tests/test_NLP.py::NLPNameExtractTestCase::test_not_string FAILED
tests/test_NLP.py::NLPNameMatchTestCase::test_empty_string FAILED
tests/test_NLP.py::NLPNameMatchTestCase::test_name_match_diff_format_1 PASSED
tests/test_NLP.py::NLPNameMatchTestCase::test_name_match_diff_format_2 FAILED
tests/test_NLP.py::NLPNameMatchTestCase::test_name_match_diff_format_3 PASSED
tests/test_NLP.py::NLPNameMatchTestCase::test_name_match_diff_names_1 PASSED
tests/test_NLP.py::NLPNameMatchTestCase::test_name_match_diff_names_2 PASSED
tests/test_NLP.py::NLPNameMatchTestCase::test_name_match_diff_names_3 PASSED
tests/test_NLP.py::NLPNameMatchTestCase::test_name_match_same PASSED
tests/test_NLP.py::NLPNameMatchTestCase::test_not_string FAILED
tests/test_Parser.py::SpeechParserTest::test_parser_find_files PASSED
tests/test_Parser.py::SpeechParserTest::test_parser_parse_files SKIPPED

```

```
================================== FAILURES ==================================
_____________________ NLPNameExtractTestCase.test_not_string _____________________

self = <test_NLP.NLPNameExtractTestCase testMethod=test_not_string>

    def test_not_string(self):
        test_string = 4
>       self.assertEqual(NLP.extract_name(test_string), "")

tests/test_NLP.py:111:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
NLP.py:40: in extract_name
    match = re.search(name_rex, text)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

pattern = re.compile('(\\bMr[ .]|\\bMrs[ .]|\\bSir[ .]|\\bDr[ .])[^!"#$%&\'()
    *+,/:;<=>?@[\\]^_`{|}~\\\\\\n]+', re.IGNORECASE)
string = 4, flags = 0

    def search(pattern, string, flags=0):
        """Scan through string looking for a match to the pattern, returning
        a match object, or None if no match was found."""
>       return _compile(pattern, flags).search(string)
E       TypeError: expected string or bytes-like object

/usr/lib/python3.5/re.py:173: TypeError
_____________________ NLPNameMatchTestCase.test_empty_string _____________________

self = <test_NLP.NLPNameMatchTestCase testMethod=test_empty_string>

    def test_empty_string(self):
        test_name_one = "Mr Thomas Ridgewell"
        test_name_two = ""
>       self.assertFalse(NLP.name_match(test_name_one, test_name_two))

tests/test_NLP.py:160:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

name_one = 'Mr Thomas Ridgewell', name_two = ''

    def name_match(name_one, name_two):
        # we assume the name has already been extracted using the above
    function. Maybe we should use that now anyway?
        if name_one == name_two:
            return True  # if they are the exact same then of course they match
        # if they are not the exact same, that doesn't quite mean they are
    different
        # could be a full name vs just surname (Mr John Smith vs Mr Smith)
        # so check if name two contains the surname from name one?
        # also, potentially make sure the first names are the same too. If we
    have John Smith and Matt Smith we need to
        # make sure their separate speech is categorised separately

        name_one_words = name_one.split()
        name_two_words = name_two.split()

```

```
85          # ideally, names will always be either 2 or three words, one honorific
        at the start, and a surname at the end
86          # with maybe a forename between the two
87          if len(name_one_words) == 2:
88              # two words means honorific and surname, right? right
89              hon_one = name_one_words[0]
90              forname_one = None
91              surname_one = name_one_words[1]
92          elif len(name_one_words) == 3:
93              hon_one = name_one_words[0]
94              forname_one = name_one_words[1]
95              surname_one = name_one_words[2]
96          else:
97              try:
98                  hon_one = name_one_words[0]
99                  surname_one = name_one_words[-1]
100                 forname_one = None
101             except IndexError:
102                 print("ERROR: Index out of bounds. Is the name blank?")
103                 print("Name causing error: {}".format(name_one))
104                 raise
105
106         if len(name_two_words) == 2:
107             # two words means honorific and surname, right? right
108             hon_two = name_two_words[0]
109             forname_two = None
110             surname_two = name_two_words[1]
111         elif len(name_two_words) == 3:
112             hon_two = name_two_words[0]
113             forname_two = name_two_words[1]
114             surname_two = name_two_words[2]
115         else:
116             try:
117 >               hon_two = name_two_words[0]
118 E               IndexError: list index out of range
119
120 NLP.py:92: IndexError
121 ---------------------------- Captured stdout call
        -----------------------------
122 ERROR: Index out of bounds.
123 Name causing error:
124 ____________ NLPNameMatchTestCase.test_name_match_diff_format_2
        ____________
125
126 self = <test_NLP.NLPNameMatchTestCase testMethod=test_name_match_diff_format_2>
127
128     def test_name_match_diff_format_2(self):
129         test_name_one = "Mr. Thomas Ridgewell"
130         test_name_two = "Mr. T. Ridgewell"
131 >       self.assertTrue(NLP.name_match(test_name_one, test_name_two))
132 E       AssertionError: False is not true
133
134 tests/test_NLP.py:135: AssertionError
135 ____________________ NLPNameMatchTestCase.test_not_string
        ____________________
136
137 self = <test_NLP.NLPNameMatchTestCase testMethod=test_not_string>
138
139     def test_not_string(self):
```

```
140         test_name_one = "Mr Thomas Ridgewell"
141         test_name_two = 4
142 >       self.assertFalse(NLP.name_match(test_name_one, test_name_two))
143
144 tests/test_NLP.py:165:
145 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
146
147 name_one = 'Mr Thomas Ridgewell', name_two = 4
148
149     def name_match(name_one, name_two):
150         # we assume the name has already been extracted using the above
    function. Maybe we should use that now anyway?
151         if name_one == name_two:
152             return True  # if they are the exact same then of course they match
153         # if they are not the exact same, that doesn't quite mean they are
    different
154         # could be a full name vs just surname (Mr John Smith vs Mr Smith)
155         # so check if name two contains the surname from name one?
156         # also, potentially make sure the first names are the same too. If we
    have John Smith and Matt Smith we need to
157         # make sure their separate speech is categorised separately
158
159         name_one_words = name_one.split()
160 >       name_two_words = name_two.split()
161 E       AttributeError: 'int' object has no attribute 'split'
162
163 NLP.py:58: AttributeError
164 ================ 4 failed, 20 passed, 1 skipped in 1.09 seconds
    ================
```

Appendixes/040518.log

Appendix B

Ethics Submission

The Ethics form submitted for this project

AU Status
Undergraduate or PG Taught

Your aber.ac.uk email address
adn2@aber.ac.uk

Full Name
Adam Neaves

Please enter the name of the person responsible for reviewing your assessment.
Prof. Reyer Zwiggelaar

Please enter the aber.ac.uk email address of the person responsible for reviewing your assessment
rrz@aber.ac.uk

Supervisor or Institute Director of Research Department
cs

Module code (Only enter if you have been asked to do so)
CS39440

Proposed Study Title
Hansard Sentiment Analysis

Proposed Start Date
29/01/2018

Proposed Completion Date
04/05/2018

Are you conducting a quantitative or qualitative research project?
Mixed Methods

Does your research require external ethical approval under the Health Research Authority?
No

Does your research involve animals?
No

Are you completing this form for your own research?
Yes

Does your research involve human participants?
No

Institute
IMPACS

Please provide a brief summary of your project (150 word max)
Training a sentiment analysis (Opinion Mining) AI model on the Hansard Dataset, a set of everything said in political debates for the past 200 years. The model will be trained to recognise sentiment expressed by politicians and will then be used to compare historically expressed opinions with those of modern politicians to see how things have changed.

Where appropriate, do you have consent for the publication, reproduction or use of any unpublished material?

Annotated Bibliography

[1] House of Commons Information Office, "Factsheet G17: The Official Report," *The Official Report*, no. August, 2010. [Online]. Available: https://www.parliament.uk/documents/commons-information-office/g17.pdf

A page published on the Parliament website that describes the history of the Hansard Report

[2] G. G. Chowdhury, "Natural language processing," *Annual Review of Information Science and Technology*, vol. 37, no. 1, pp. 51–89, 1 2003.

[3] B. Liu, "Sentiment Analysis and Subjectivity," *Handbook of Natural Language Processing*, no. 1, pp. 1–38, 2010.

A paper on the methods of sentiment analysis, and how to extract opinions from text.

[4] I. Augenstein, T. Rocktäschel, A. Vlachos, and K. Bontcheva, "Stance Detection with Bidirectional Conditional Encoding," jun 2016.

A paper on Stance Detection. It describes what stance detection is and what it can be useful for.

[5] O. Onyimadu, K. Nakata, T. Wilson, D. Macken, and K. Liu, "Towards sentiment analysis on parliamentary debates in Hansard," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8388 LNCS, 2014, pp. 48–50.

A paper detailing the progress made on trying to apply standard sentimen analysis techniques to the hansard dataset. It discusses the fact that these normal approaches are inadequate for the style of speech used in parlimentary debates.

[6] TheyWorkForYou: Hansard and Official Reports for the UK Parliament, Scottish Parliament, and Northern Ireland Assembly - done right. [Online]. Available: https://www.theyworkforyou.com/

A website that uses the Daily Hansard report to list the voting habits of MPs

[7] (2017) Fake News Challenge. [Online]. Available: http://www.fakenewschallenge.org/www.fakenewschallenge.org

A challenge to create an Ai system that can help detect ”fake news”. The first part of this challenge was to create a stance analysis system that could compare the stance between the article's title and contents.

[8] W. Ferreira and A. Vlachos, “Emergent: a novel data-set for stance classification,” 2016.

An article about a data set designed for stance classification, that the fake news challenge used as a part of the stance analysis section of the challenge.

[9] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python*, 2009.

A free book that serves as an introduction to the use of the Natural Language Toolkit. This toolkit is the one used for the project's NLP requirements.

[10] L. Richardson. Beautiful Soup Documentation. [Online]. Available: https://www.crummy.com/software/BeautifulSoup/bs4/doc/

Documentation for BeautifulSoup4, a HTML/XML parser for Python, which could be useful for dealing with the large badly formatted XML documents.

[11] Harrison. NLTK with Python 3 for Natural Language Processing - YouTube - YouTube. [Online]. Available: https://www.youtube.com/playlist?list=PLQVvvaa0QuDf2JswnfiGkliBInZnIC4HL

A Youtube Tutorial Playlist about the use of the Natural Language Toolkit for python. Harrison provides many tutorials on different aspects of using Python.

[12] T. M. Mitchell Tom Michael, 1951, *Machine learning*. McGraw Hill Education, 1997.

A book on Machine learning. It contains a sizeable chapter on Naive Bayes, including a definition of Bayes Theorem, the AI algorithm used in this project

[13] N. Wacholder, Y. Ravin, and M. Choi, “Disambiguation of Proper Names in Text,” *Proceedings of the 5th Applied Natural Language Processing Conference*, 1997.

A paper on the complexities of Name Disambiguation, describing what makes this such a challenging thing in computing.

[14] T. Hastie, “K-Fold Cross-Validation,” 2009.

Slides from a lecture on k-fold cross validation, describing the maths behind it and the purpose of using it.

[15] J. Brownlee. (2016) Do Not Use Random Guessing As Your Baseline Classifier. [Online]. Available: https://machinelearningmastery.com/dont-use-random-guessing-as-your-baseline-classifier/

A blog post on the benefits of using Zero Rule as a baseline rather than random guessing.

[16] J. Knupp. (2013) Improve Your Python: Understanding Unit Testing. [Online]. Available: https://jeffknupp.com/blog/2013/12/09/improve-your-python-understanding-unit-testing/

A blog post on the benefits of Unit Testing, with a description of what unit testing is and why it's a good idea.

[17] B. B. Whaley and W. Samter, *Explaining communication : contemporary theories and exemplars*. Lawrence Erlbaum Associates, 2007.

A book on communication, both verbal and non-verbal. It describes the importance of body language in communication and the expression of sentiment.