Automated Document Summarisation using Natural Language Processing

Technical Specification

Student Name: Niall Egan

Student Number: 19378906

Student Name: John McCormack

Student Number: 19517396

Supervisor: Ray Walshe

Date Completed: 2nd March 2021

Abstract:

This document will provide a breakdown of the technical aspects of our project. It will include a general overview of the project as well as an explanation of some key terms. Following this, we will look at the system architecture for the application and breakdown of key systems e.g.: summarisation system.

We will also take a look at the design of the application at a high level, specifically focusing on how data is passed around in the application. Finally, we will look at some the problems that arose during the development, the solutions for these problems and what we would do if we were to continue developing the summarization app in the future.

Table of Contents:

1. Introduction:

1.1 Overview:

For our CA326 project, we created an application to summarise documentation provided in pdf format. Our application makes use of Natural Language Processing techniques to do the summarisation. The application takes input from the user using a front-end user interface. Here they will specify the pdf document they want summarised. It is also where the final, summarised text will appear.

Once a pdf has been selected, the application will extract the text from the document and process it into a format usable by the model weighting scheme. Our application uses a TFIDF model combined with the distance between high frequency words to provide a rating for each sentence in the pdf.

The highest rated sentences for each paragraph are grouped together. The final summarised text is formatted and displayed on the interface. From here the Application will allow users to save the summarised text as a new pdf and provide a new document for summarisation.

1.2 Glossary:

- **Natural Language Processing:**
  Natural Language Processing (NLP) is one of the many technical fields under the umbrella of Artificial Intelligence. The goal of NLP is to create systems that allow computers to understand and interpret language in a similar way humans do.

- **Corpus:**

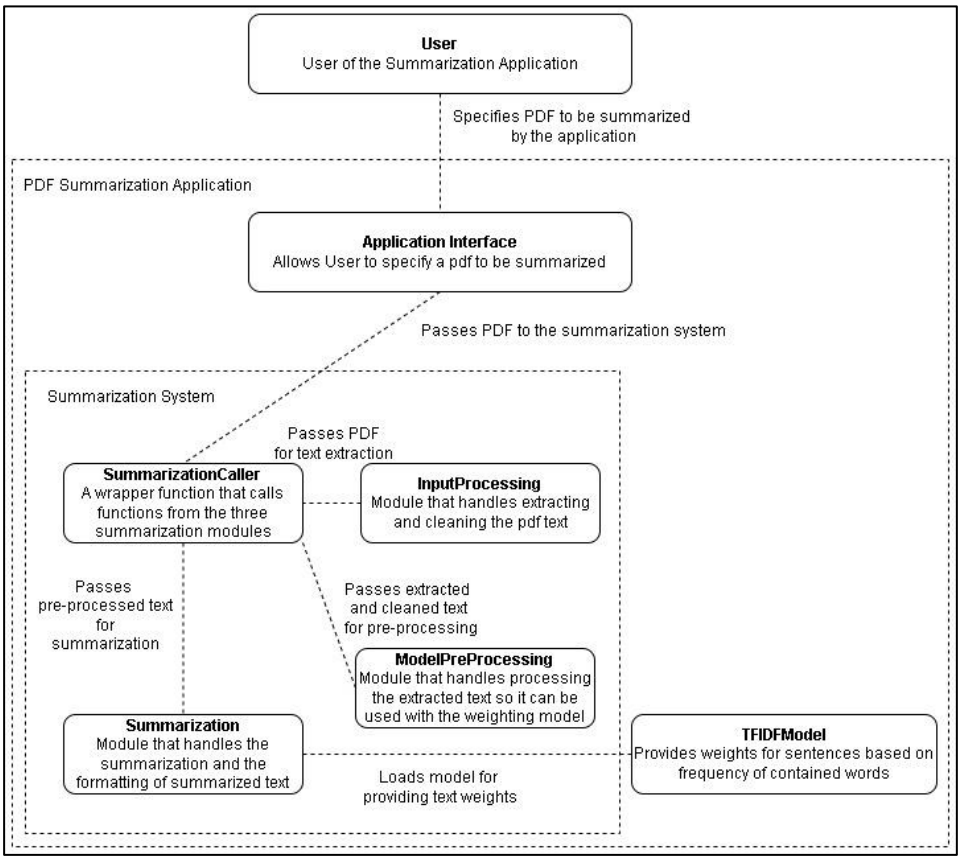
  In Natural Language Processing, the term corpus refers to a collection of text objects, referred to as documents. In programming terms, a corpus can be a list of containing strings.

- **Bag-Of-Words:**
  The bag-of-words model is a method of representing a text object, like a sentence, in a numerical format. A bag-of-words is a vector containing the number of times specific words appear in a piece of text. The bag-of-words model requires an associated dictionary to indict which words to be represented.

2. System Architecture:

2.1 System Architecture Diagram:

This is our system architecture diagram of the finalised application:



2.2 Changes to the System Architecture:

We made a number of changes to the initial system architecture specified in the Functional Specification over the course of developing the application. The two most significant changes are:

- Training and using our own weighting scheme model. Instead of using external models we decided to implement and train our own singular model using the Gensim Library.

- The scrapping of the Scripting Interface. Data from the User can now only be read in through the Application interface.

3. Implementation:

3.1 User Interface:

3.1.1 Required Libraries:

Creating the user interface required one external Python library, PySimpleGUI. This library provides pre-existing assets and simple syntax for building User Interfaces.

3.1.2 Interface:

This is how our User Interface is implemented:

```
def main():
    # User Interface testing
    sg.theme('Dark')
    layout = [
        [sg.Titlebar('Our App')],
        [sg.Text('Pick an Option')],
        [sg.Button('Summarize'), sg.Button('Train'), sg.Cancel('Close App')],
    ]

    window = sg.Window('PDF Summarizer', layout, size=(500, 500))
    while True:
        event, values = window.read()
        if event == 'Close App' or event == sg.WINDOW_CLOSED:
            break
        if event == 'Summarize':
            window.close()
            summarizeWindow()
        if event == 'Train':
            sg.popup('Under construction')

    window.close(); del window
```

```
def summarizeWindow():
    # User Interface testing
    sg.theme('Dark')
    layout = [
        [sg.Titlebar('Summarization Application')],
        [sg.Text('Filename')],
        [sg.InputText(), sg.FileBrowse()],
        [sg.Submit('Summarize'), sg.Cancel()],
        [],
        [sg.Text('Summarized data')],
        [sg.Multiline(key='-OUTPUT-', size=(400, 10))],
        [sg.Text('Save summarized data as PDF')],
        [sg.InputText(key='pdfname'), sg.FolderBrowse('Choose Folder'), sg.Button('Save as PDF')]
    ]

    window = sg.Window('PDF Summarizer', layout, size=(600, 600))
    while True:
        event, values = window.read()
        if event == 'Cancel':
            window.close()
            main()
        if event == 'Summarize':
            path = values['Browse']
            summary = output_format_gui(summarize_wrapper(path))
            window['-OUTPUT-'].update(summary)
            #sg.popup_scrolled(data)
        if event == 'Save as PDF':
            pdfname = values['pdfname']
            canvas = Canvas(pdfname)
            text_object = canvas.beginText()
            text_object.setTextOrigin(10, 730)
            text_object.setFont("Times-Roman", 12)
```

Our interface is made up of two windows, the main menu and the summarisation window. They both use PySimpleGUI's layout structure. The elements of the UI are stored in a list. Their position in the list indicates their

vertical position on the UI. Elements are in a sub list then they are on the same line.

The 'While True' loops control the logic for the custom buttons on the UI. Every loop, the current event is checked to see if it matches a previously defined event. The events are attached to buttons. If a matching event is detected, then it runs the appropriate code.

3.2 PDF Text Extraction:

3.2.1 Required Libraries:

To implement a system to extract text from pdf documents we had to use one external Python library, 'pdfminer.six'. This allowed us to extract text along with important information like font size.

3.2.2 Text Extraction:

This is our text extraction system:

```
def extract_text_with_font_sizes(data):

    path = 'testPDFs/' + data

    extracted_data = []

    for page_layout in extract_pages(path):
        for element in page_layout:
            if isinstance(element, LTTextContainer):
                for text_line in element:
                    for character in text_line:
                        if isinstance(character, LTChar):
                            font_size = round(character.size, 0)
                extracted_data.append((font_size, (element.get_text())))

    return extracted_data
```

It leverages pdfminer.six's system for iterating through a pdf's structure. It starts by iterating through a page's layout to find a text container. Once it finds one, it cycles through each text line and character to find the font size of the line. The font size and text line are then stored as a tuple in a list. It extracts font size so our application can maintain paragraph structure in the final summarised product.

The data extracted from here is then passed on for cleaning.

3.2.3 Extracted Text Cleaning:

This is our system for cleaning extracted text:

```
# Removes new line characters and PDF formatting artifacts from the text
def clean_data(data):
    cleaned = []

    # Removes new line characters
    for pair in data:
        heading = pair[0].replace("\n", "")
        body = []
        for sentence in pair[1]:
            sentence = sentence.replace("\n", "")
            sentence = re.sub('(\\u202f)|(\\x0c)|(·)', '', sentence)
            if sentence != ' ':
                body.append(sentence)

        cleaned.append((heading, body))

    return cleaned
```

It iterates through the list of paragraph lists and replaces any instances of a newline character or pdf formatting artifacts and replaces them with an empty string, effectively removing them. Removing the formatting artefacts required the use of a regular expression. This process is performed on both the heading and all sentences in the body of the paragraph.

3.3 Model Pre-Processing:

3.3.1 Required Libraries:

Model pre-processing required one external Python library, Natural Language Toolkit (NLTK). It also required the built-in class 'defaultdict'.

3.3.2 Tokenization and Removal of Stop words:

This is how the application tokenizes sentences:

```
# Splits the document in the Corpus into tokens
def tokenize(corpus):
    complete_token_list = []

    for paragraph in corpus:
        paragraph_tokens = []
        for document in paragraph[1]:
            tokens = document.split(' ')
            tokens = remove_stopwords(tokens)

            paragraph_tokens.append(tokens)

        complete_token_list.append((paragraph[0], paragraph_tokens))

    return complete_token_list
```

It iterates through each line of text in a paragraph, splitting them into tokens and adds them to a list. Each list of paragraph tokens is then appended to a final token list as a tuple with the paragraph heading.

Another element of pre-processing was removing stop words. Stop words, such as 'a' or 'in', don't provide any meaning to text so there is no purpose in considering them. This is the system we used:

```
# Removes stop words from the token lists
def remove_stopwords(tokened_words):

    removed = []
    for word in tokened_words:
        if word not in stopwords.words('english') and word != '':
            removed.append(word)

    return removed
```

It iterates through the tokens for all paragraphs. If the token appears in the list of stop words from the NLTK library, it is removed from the list.

3.3.3 Calculating Token Frequency:

This is how the application calculates how often tokens appear in the extracted text:

```
# Creates a frequency dictionary for the all tokens
def token_frequency(tokens):
    token_frequency_dict = defaultdict(int)

    for token_list in tokens:
        for sentence in token_list[1]:
            for token in sentence:
                token_frequency_dict[token] += 1

    return token_frequency_dict
```

The function iterates through the list of tokens. Every time a token appears its associated value in the dictionary increases by one. Since it's using the default dictionary class, tokens don't have to be added to the dictionary before they can be counted. If a token doesn't have an entry in the dictionary when used as a key, one will automatically be created for it.

We track token frequency for a number of reasons. It is used for identifying words that appear less than once, so they can be removed. It's used in the process of training the model.

3.4 Summarisation System:

3.4.1 Required Libraries:

The summarisation system required one external Python library, the Gensim Corpora library.

3.4.2 Converting to Bag-Of-Words format:

This is the function we used to convert the processed text into the Bag-of-Words format:

```
# Converts a document into bag-of-words format using corpora dictionary
def create_corpus_bow(texts, dictionary):
    bow_list = []

    for text in texts:
        text_bow = []
        for doc in text[1]:
            bow = dictionary.doc2bow(doc)
            text_bow.append(bow)
        bow_list.append(text_bow)

    return bow_list
```

The function iterates through the list of tokens for each sentence in a paragraph. The corpora dictionary function dc2bow() is used to convert a

sentence's token list to a Bag-Of-Words. Each BOW is added to a list for the paragraph which is then added to the final list.

3.4.3 Transforming text using the Model:

This is the system we used to get the weights for each sentence:

```
# Transforms a bag-of-words representation to correct vector space using provided model
def corpus_transform(corpus_bow, model):
    full_transform = []
    for paragraph in corpus_bow:
        transformed_paragraph = []
        for document in paragraph:
            transformed_doc = model[document]
            transformed_paragraph.append(transformed_doc)
        full_transform.append(transformed_paragraph)

    return full_transform
```

This iterates through the list of bag-of-words representations for each paragraph. The 'model[document]' notation converts or transforms the frequency value of each word in bow to a float based on the model. The transformed sentences are then added to a paragraph list. That list is then appended to a final list.

3.4.4 Calculating distance between High Value Words:

The function we used to calculate the distance between important words looks like this:

```
# Calculates the distance between the top two high frequency words
def get_word_distance(sentence, high_value_words):

    # Flag to indicate when to count words
    distance_state = 0
    distances = []
    count = 0
    for word in sentence:
        if word in high_value_words and distance_state == 1:
            # print('Toggled state, saved distance')
            distance_state = 0
            distances.append(count)
            count = 0
        elif word in high_value_words:
            distance_state = 1
            # print('Toggled state')

        if distance_state == 1:
            count += 1

    # Gets the average distance of all word distances
    avg_distance = sum(distances) / len(distances)

    return avg_distance
```

The function takes a sentence and a list of the two highest frequency words in the frequency dictionary. It iterates through the sentence. When it encounters a high value word, it sets the 'distance_state' flag to 1. When its value is 1, it will count each word it iterates over. When it encounters another high frequency word, the flag is set to 0 and the counting stops. The final count is added to a list.

Once all the distances have been found, the average distance is calculated and returned.

3.4.5 Calculating the final weights and final summarisation:

This is the system we used to calculate the final weight:

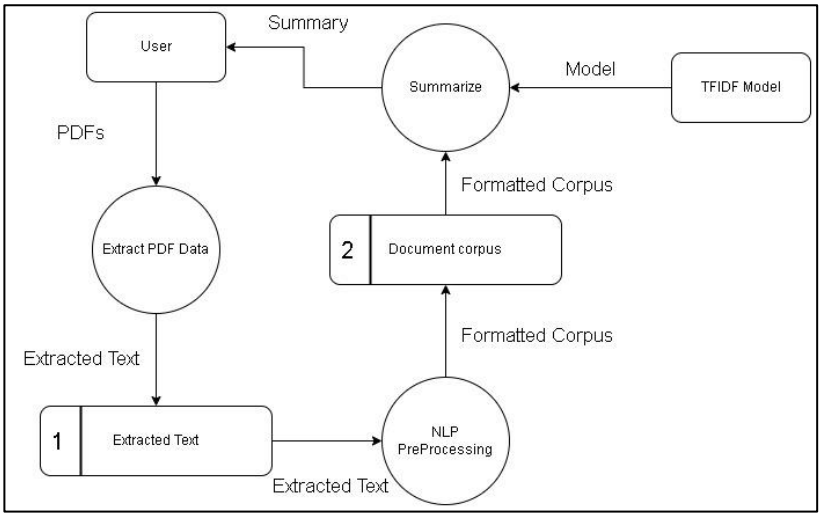
```
def combined_summarizer(values_list):
    def sort_helper(element):
        return element[1] + element[2]

    summarized_list = []

    for x in values_list:
        summarized_list.append((x[0], sorted(x[1], key=sort_helper, reverse=True)))

    return summarized_list
```

The function iterates through a list where each element is a tuple containing a paragraph name and a list containing sentences from the paragraph along with its model weight and average word distance. It sorts the list of sentences based on the sum of their model weight and high frequency word distance.

4. High-Level Design:

4.1 Data Flow Diagram:

This is our finalized Data Flow Diagram for the Summarisation application:



4.2 Changes to Data Flow:

The flow of data in our application hasn't changed significantly since the functional specification. The main change is to how completed summaries are handled. They no longer have a persistent data store. Once a summary is completed, it is pushed to the interface where the user can save it if they choose.

5. Problems and Resolution:

While working to develop the summariser application, we encountered a number of different problems and challenges that hindered development. Finding solutions for these problems was vital so that we could deliver the application on time. Some of these problems include:

- Finding an existing weighting scheme model to suit our needs. When we started developing the application, we initially intended to use an existing model so we could focus more on developing the application itself. However, after looking for a model, we realised we weren't going to find one that exactly suited our needs.

  Our solution to this problem train our model. While this process did take up time, it was beneficial as it allowed use to control exactly the type of documents our model would be trained on.

- Underestimating how time-consuming working with multiple Python libraries for the first time would be. When we started the project, neither of us had experience with Natural Language Processing and the related Python libraries. This lack of knowledge wasn't factored in during the initial planning stage and timetable. We ended up using a lot of time on just learning how to use the new libraries.

  Our solution to this problem was to unfortunately scale back on our idea. Initially, we intended to allow users to automate summarising pdfs with our application using scripts. This idea had to be cut in the end as we didn't have enough time to develop or test it.

6. Future Work:

There are a number of different avenues to improve the application that could be attempted in the future. These include:

- Experimenting with new weighting scheme models. When developing the application, we primarily worked with two models, LSI and TFIDF, eventually settling on the latter. We believe it would be worth testing other models to see how they impact the summarisation quality.

- Improving text extraction. This was probably the most difficult part of developing the application. Due to how PDF's handle formatting, it is difficult to maintain the paragraph structure. In the future, it would be worth it to try developing a new extraction approach and some other Python libraries.

- Finding a more reliable method of testing. While developing the application, we tested it by comparing the output with other summarising applications on the internet. However, it was not the most reliable. We had to perform judgement by eye as to if the output had improved and which was better. Another issue was that for long documents, like technical standards documents, the online summarisers would normally crash before completion, so it was difficult to test with those. A more reliable form of testing would definitely help with future developments.