

Implementing a Document Summarisation system on a Flask Framework

Krzysztof Dworczyk

40340711@live.napier.ac.uk

Edinburgh Napier University

Abstract

This paper will show how a summarisation system can be created using the NLTK library and how it can be implemented inside on a flask framework web application. A text pre-processing system is developed to allow for a seamless text upload. Further improvements are made to allow users to refine their searches with tags. This system will allow anyone trying to save time by condensing texts into smaller readable segments.

1 INTRODUCTION

As we gain access to more and more information via the internet like articles, new stories, and journals we may spend a lot of time reading this information. Recently however there have been a lot of research going into software's which can produce summarisation of single or multiple page documents. These are immensely useful in areas such as business where producing daily reports on company data and statistics is important to keep profits in the positive. Weather report are another area which has seen a rise in the use of natural language generation (NLG) to allow for monotonous tasks to be automated. However, this is not a new topic, there have been papers (Luhn, 1958) released as far back as 1950s titled "The

Automatic Creation of Literature Abstracts", which describe ways in which topics of research papers can be categories automatically to saves readers times in finding them. So, we can see that the interest in this area has been for a very long time and with good reason. The internet is absolutely filled to the brim with all kinds of information, some repeating similar stuff like news papers and others researching areas and producing reports on them. They all can benefit from being summarised.

To help in summarising text documents I have developed a web application which allows users to upload their document to be summarised by a certain percentage. The following paper is split up into following sections. Section 2 will detail the 2 different summarisation approach that can be applied to this project and which one we will be taking. Section 3 will introduce the technologies used to build the application such as NLTK for data pre-processing and sklearn to produces metrics on the words. I will go over what GloVe word embeddings are and how they improve performance in the web application. Section 4 will cover the analysis and design of the application to provides an overall goal and scope for the project as well as the vital steps needed for the project's completion. Section 5 will present the implemented web application with the user interfaces explained along with some important code snippets that allow the summarisation to happen. Finally, section 6 will conclude the research and explain what has been achieved as well as what further work can be done in the future.

2 EXTRACTIVE AND ABSTRACTIVE SUMMARISATION

Text summarisation can be divided into two different categories being Abstractive and Extractive Summarization. The former uses linguistic techniques and neural networks to comprehend the text and produces its own summary (Vipul Dalal, 2013) while the former creates sentence vectors using things like Bag of Words (**BOW**), term frequency-inverse

document frequency (**TF-IDF**) or word embeddings to apply importance to the extracted words to then be able to display on the most important ones.

2.1 EXTRACTIVE

This is the technique which will be applied in the project. It will rely on choosing one of the techniques to create vectors for the words to show relationships between them. A computer does not know if any words may be closer related or spoken together more often. To help with this we will be using pre-trained word embeddings which will save as a lot of overhead time which we would need to use for creating these embeddings.

2.2 ABSTRACTIVE

This method tries to create and abstract of the overall document. This results in a usually very short and concise summary if correctly implemented and trained. It will allow the system to create its own sentence structures to attempt at describing the texts it was given. This is much different from the extractive approach as in that one we will not be creating any new words or sentences but rather just ranking their importance to the overall document.

3 TECHNOLOGIES

In this section I will be going over the technologies used to pre-process the texts, represent them in vectors, develop a web application using python and implemented it on the flask framework.

3.1 LIBRARIES USED

Before we get into any of the bigger libraries let me quickly explain the smaller but still very important puzzle pieces to this project.

NumPy and **Pandas** both allows for very easy initialisation of arrays and manipulation of data inside them. We will be using both to help in creating similarity matrixes and getting the similarities between sentences. These will

basically be the main classes used throughout the project to transform the data.

Scikit-learn is a huge scientific library which provides many machine learning and deep learning algorithms. We will be using it to compute the similarity between pairs of sentences with its Cosine Similarity function.

3.2 NLTK & PRE-PROCESSING

The most important step in any data driven projects is the cleaning and pre-processing its data. It aims to reduce size, normalize, and remove any outliers from it (Bhaya, 2017). To help with this I will be using the NLTK library which provides a plethora of out-of-box tools which allows us to transform our texts any way we want.

Tokenization: Before any meaningful data cleaning can take place, we first need to tokenize the words/sentences (EMMA L. TONKIN, 2016). However, in this problem instance we will be using something called **sent_tokenize()** which is a NLTK tokenizer but for sentences. So instead of splitting it by words we do it by the sentence as we want to maintain the full context of them to later rank and display.

Sanitation: To sanitise the data we will be removing punctuations, numbers, and special characters. We then can lowercase all the words. Finally, we flatten the list by getting rid of any duplicates.

3.3 GLOBAL VECTORS FOR WORD REPRESENTATION

GloVe allows us to use already pre-trained word embeddings which significantly help our run times as this process is done beforehand. Word embeddings allows the computer to know which words are correlated and are close in the search space. (Jeffrey Pennington, 2016)

Word embedding usually need to be trained before being used but in our case, we are going to be using a pre-trained word vector called **Wikipedia 2014 + Gigaword 5** which can be found on the GloVe website. Its size is almost 1GB which provides us with 400K vocabulary

vectors. These will be used later in the web application implementation in the text summarisation python file.

3.4 FLASK FRAMEWORK

The Flask Framework came into existence only in 2010 but has shortly become one of the most used ones. It is a micro and lightweight web framework which allows for rapid development, prototyping and deployment of applications which is one of the factors of its popularity. It is a very well documented framework and even though its lightweight it provides enough complexity to develop anything as developer may need. It has built in security and has additional libraries which help with that.

The choice of this framework was simple, it provides a rapid way of prototyping my application as I constantly need to re-run it with new changes happening. It works on a route-based system and implements the POST and GET methods to retrieve data across pages in forms. Overall, it provides everything and more I need to develop my application.

4 ANALYSIS AND DESIGN

The following sections will lay out the analysis made to the project and finally show the design of final system implementation.

4.1 ANALYSIS

4.1.1 System Requirements Specification

4.1.1.1 Purpose

The purpose of the application will be to allow users, students, researchers, or anyone trying to save time to summarises their documents. The application will allow for a more streamlined way of summarising documents in hopes of help saving time.

4.1.1.2 Project Scope

The project will aim to implement the NLP techniques learned throughout the module to allow for summarisation of text documents. The summariser will be implemented on a

flask web application with options provided to tune the amount of summarization a user requires.

4.1.1.3 User Characteristics

The application did not require any technical backgrounds and will be created as to allow any level user to upload their text for summarisation.

4.1.2 Functional Requirements

In the section below I will explain in detail the functional parts of the application and how they will all interact with each other to allow for seamless text summarisation.

4.1.2.1 User Interface

Inputs

- Radio buttons for choosing the summarisation options, these will be automatic, manual and tag optimized
- Radio buttons for choosing input types lie from file, text box or example texts
- Range slider for a manual input of a percentage to summarise the texts by
- Text box for inputting the tags for further optimization of output texts
- File browser for locating text files on users' machine for uploading
- Text box for manual inputting of texts to the web application
- Submit button for begging the summarisation process

Outputs

- Multiline output boxes, these will contain the full text which is being used for processing and finally the summarised text

4.2 DESIGN

In this section I will discuss the design plans which were used to develop the final applications. These will help in understand the overall structure and flow of the application.

4.2.1 USE CASE Diagram

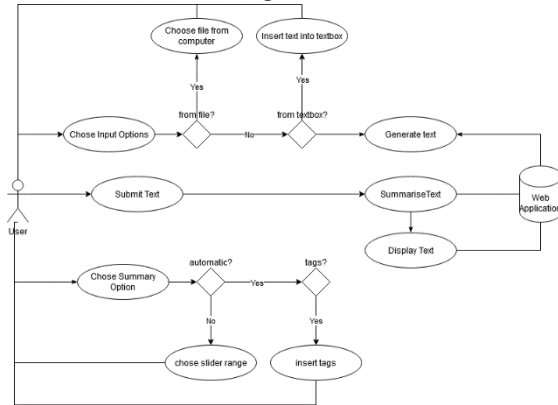


Figure 1 - use case diagram showing every possible action a user can take along with the choices which they lead to

The Figure 1 shows the use of the program from the user and web application perspective. Each action is represented by the ellipse shapes and the triangles signify a choice where the answer allows the user to use that part of the system.

The user has 3 initial options they use, the input options, summary options and finally the submit text which will be a button in the application. The input option allows the user to select a file from their pc using a file browser, insert the text straight into the application interface or they can choose to generate some random texts for showcasing. The summary options can either be left alone which will result in automatic summarisation which will aim for half of the text content or the user can choose a specific percentage they want the text to be summarised by. The last option for the summary is for tags optimization. This will be discussed in detail later, for now the user can chose to use them which will allow the user to insert them to the provided textbox.

Finally, the user can press the submit button which will apply all the settings and begin the summarization process. The web application will process it and display the output text for the user to see.

4.2.2 Class Diagram

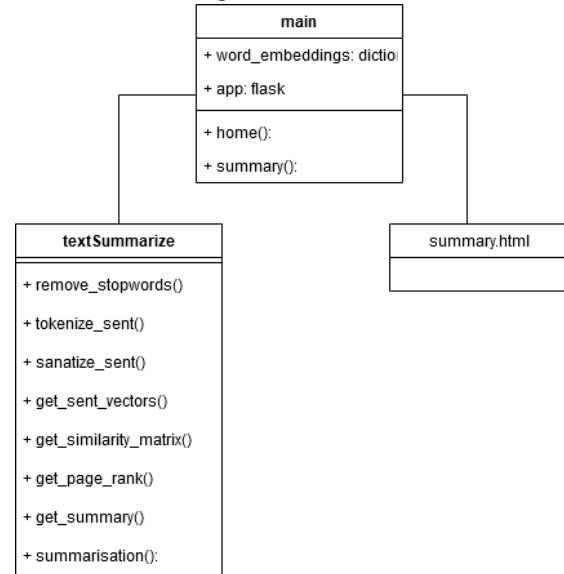


Figure 2 - class diagram showing the class structure with functions and variables

Figure 2 shows the initial design of the web application. It is not made of a lot of parts as most of the bulk of code is inside the **textSummarize** class, the rest are used to manipulate the HTML files around and feed it the correct data to display.

4.2.2.1 Main

The main class is where we start up our flask applications. Therefore, we will have a variable **app** in there which will be of type flask called **app**. We also include the word embeddings there as a dictionary, these will be unpacked on start-up as to only have to wait once for it. This way we do not have to wait every time we want to summaries a word. When the application is run the initial function **Home()** is run which display the html file which allows the user to begin choosing summary options and what type of input option they will use. Once the user has selected everything, they can submit the form which will be sent in as a POST method. This will trigger the **summary()** function which will begin to summarise the inputted text.

4.2.2.2 textSummarize

The textSummarize contains all the pre-processing and the sentence ranking algorithm. From the figure we can see there is a breakdown of pre-processing steps,

remove_stopwords() and **Tokenize_sent()** will tokenize out sentences which then we can removes stop words from. We further use **Sanatize_sent()** to remove punctuations, numbers and special characters and finally make it all lower case.

get_sent_vectors() will return sentence vectors for us.

get_similarity_matrix() will get us the similarities between sentences which we will then put inside the cosine similarity metric.

get_page_rank() we turn the similarity matrix *sim_mat* into a graph. The nodes on the graph will be the sentences and the edges will be representing the similarities scores between the sentences. We use the graph to apply our PageRank algorithm to receives the results.

get_summary() the final function will return us the final summary by extracting the N top sentences from the produces ranking. If the user decided to use tags, then we only add the sentence to the **final_sentence** if a word in it matches a tag. We then return the result.

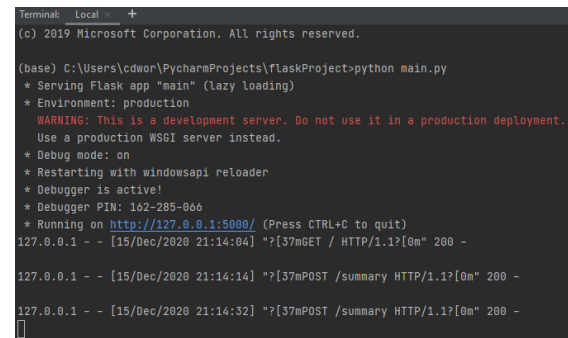
summarisation() this is the main function that is called to begin summarisation, it is where the main text is being input along with any of the user options. It will they begin to process the text by using the other functions.

4.2.2.3 Home.html

The summary.html file is made up of basic elements like labels, buttons, radio buttons, text boxes and range sliders. We can pass data to the page as flask allows us to give parameters to it when return templates to display. This allows for quick and dynamic changes to be made. We can go as far as write python inside the html file like loops or if statements which add great complexity and variety to what we can do. This was however kept simple as to allow more development towards the summariser.

5 WEB APP USER INTERFACE

5.1 STARTING FLASK/SERVER



```

Terminal: Local +
(c) 2019 Microsoft Corporation. All rights reserved.

(base) C:\Users\cdwon\PycharmProjects\FlaskProject>python main.py
* Serving Flask app "main" (Lazy loading)
* Environment: production
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
* Debug mode: on
* Restarting with windowsapi reloader
* Debugger is active!
* Debugger PIN: 162-285-866
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [15/Dec/2020 21:14:04] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [15/Dec/2020 21:14:14] "[37mPOST /summary HTTP/1.1" 200 -
127.0.0.1 - - [15/Dec/2020 21:14:32] "[37mPOST /summary HTTP/1.1" 200 -

```

Figure 3- server being started with some POST commands being sent from the summary page form when submitting a summary.

In Figure 3 we can see how a flask server is started using a simple command **python main.py** where the main is the file where the app is started. The console will then take some time to load the word embeddings and after some debugger warning, we get a local IP address where we can access the web application. The warnings are not relevant to us. At the end we can see 3 POST commands being sent when I summarised some of the files in the input folders. This would be hosted on a server with a public IP and with further development could allow users anywhere to summarise their own documents.

5.2 SUMMARIZATION OPTIONS

Figure 4 you can see the summarisation options which can be tweaked in the application. The first one is the automatic, this one will try and summarise the document with half of the necessary texts.

Summarization Options

☒ Automatic ☐ Manual ☐ Tag Optimized

Automatic Summarisation: 50%

Figure 4 - summarization option 1

To calculate our final summary length, we use the equation below in figure 5 which is found in the code.

```
summary_length = int((manual_range / 100) * num_sentences)
```

Figure 5 – code snippet of how we calculate how long the summary should be in lengths of sentences.

It takes the slider value **manual_range** and turns it into a decimal which we can then use to multiply the number of sentences we have. **summary_length** is then used to extract the top N amounts of sentences which will correspond to the percentage chosen. It may not always have a perfect number of sentences due to odd numbers. The system will always round down the number of sentences. So, if there are 9 then 50% would be 4.5 which will be interpreted as 4 sentence summarisations.

The next “manual” options in Figure 6 present a slider which allows the user to select a percentage the text will be summarised by. This is only a guide and will not be exact. It may only be different by a few percentages as it may not be possible to split the text up into equal parts as mentioned before.

Summarization Options

☐ Automatic ☒ Manual ☐ Tag Optimized

80%

Figure 6 - summarization option 2

The final options in Figure 7 are for tag optimization. This allows the user to insert tags that interests them for their specific document they are summarising. For example, if a page is being summarised about Text Files, then we may want to further optimise the summarisation by only including the parts which have our tags. This works on top of the summarisation, so the document still gets the

default 50% summarisation value but this time at the end we further extract sentences which contain our tags in them. The sentences with the most tags will appear first in the output, essentially further filtering the texts for the user.

Summarization Options

☐ Automatic ☐ Manual ☒ Tag Optimized

Word tags separated by comma:

Figure 7 - summarization option 3

The Figure 8 is the code snippet which will be run when a user enters their tags. If no tags are entered, then we simply extract the top N number of sentences which will be our final summary. If we however do have tags, then we loop through the sentences until we have found N number of sentences with the words which we have inserted into the tag box.

```
tagged_sentences = {}

for i in range(summary_length):
    final_sentences.append(ranked_sentences[i][1])_if no tags just use ranking

    if tags: _if tags then only select sentences with tags in them
        for sentence in final_sentences:
            for tag in separated_tags:
                if tag in sentence:
                    if sentence not in tagged_sentences.keys():
                        tagged_sentences[sentence] = 1
                    else:
                        tagged_sentences[sentence] += 1

return final_sentences, num_sentences, tagged_sentences
```

Figure 8 - code snippet showing the logic which is used to add tagged sentences into a list. It is mostly made up of **for** and **if** loops which will check every sentence for every tag.

5.3 INPUT OPTIONS

Next set of options presented in Figure 9 are for the user to select what type of input they will be using for the texts. The first “From File”, this lets the user to select a text document from their local workstation for summarising. This file will be uploaded to the flask storage container and processed by the text summariser.

Text Input Options

☒ From File ☐ Input Text ☐ Example Text

Text File Path: No file selected.

Figure 9 - text input options 1 is from file, opens a file browser

The second option in Figure 10 is for raw text input that can be typed or pasted into the text box. This has an unlimited word range which allows the user to paste multiple documents for multiple document summarisations.

Text Input Options

☐ From File ☒ Input Text ☐ Example Text

input raw text here!

Figure 10 - text input option 2 allows for users to input text directly

The last options are a quick use option “Example Text”. This will simply choose a random text in the already existing flask file container and process it using the user selected options.

Text Input Options

☐ From File ☐ Input Text ☒ Example Text

Figure 11 - text input option 3 allows for a random already existing file to be selected to process for a quick use of the options

5.4 UPLOADING THE TEXT

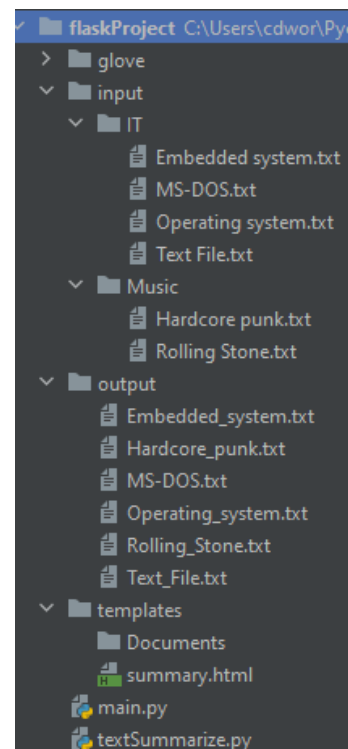


Figure 12 - Hierarchy view showing how the file storage works in flask

Figure 12 is showing the view in the IDE project hierarchy, we can see where the **glove** folder is with the word embeddings at the very top. Next are the input and output folders, if this were deployed on a real's server then the input files would be on the local machines and the output files are stored on the sever for displaying. When we chose files from the pc with the file browser they are first stored on the server for manipulation. When we click the generate button, we are choosing a random text from the output folder of already saved texts.

5.5 PROCESSING THE TEXT

In this section I will detail how the main code blocks work which pre process the text and get it ready for the algorithm.

```
def get_sent_vectors(clean_sentences, word_embeddings):
    sentence_vectors = []
    for i in clean_sentences:
        if len(i) != 0:
            v = sum([word_embeddings.get(w, np.zeros((100,))) for w in i.split()]) / (len(i.split()) + 0.001)
        else:
            v = np.zeros((100,))
        sentence_vectors.append(v)
    return sentence_vectors
```

Figure 13 - code snippet showing how we create the sentence vectors; we use the previously loaded word embeddings for it

In the Figure 13 above we are seeing a function which lets us get vectors for our sentences, we just need to pass it the cleaned sentences and the word embedding we have download from GloVe. These embeddings have already been loaded and we are just passing the references to them. This is where we are saving a lot of time by not loading it in every time, we need to summarise a piece of text. It would take upwards of 20 to 30 additional seconds before any output was produced.

Once we have created the vectors for out sentence, we return them, these will then be used in a matrix to get our similarities between sentences.

```
# similarity matrix
sim_mat = np.zeros([len(sentences), len(sentences)])

for i in range(len(sentences)):
    for j in range(len(sentences)):
        if i != j:
            sim_mat[i][j] = \
                cosine_similarity(sentence_vectors[i].reshape(1, 100), sentence_vectors[j].reshape(1, 100))[0, 0]

return sim_mat
```

Figure 14 - code snippet which returns a similarity matrix

In Figure 14 we are creating a similarity matrix for our sentences. We first initialise a NumPy array of all zeros of sentence length by sentence length. It then uses the scikit learn cosine similarity function to create our matrix which we then return to the main function.

```
def get_page_rank(sim_mat):

    nx_graph = nx.from_numpy_array(sim_mat)
    scores = nx.pagerank(nx_graph)

    return scores
```

Figure 15 - code snippet showing how the final scores of the sentence similarities are calculated

The final step in figure 15 in our processing before we can rank and extract the sentences is to create the ranking for the texts. We do this by turning the similarity matrix into a graph, the points are the sentences, and the edges are similarities scores between sentences.

5.6 FULL VIEW OF SUMMARIZATION

To get the final summary we call the **get_summary()** function. We pass it the sentence scores, sentences and if options have been selected, we also pass the manual summary range value or the tags the user has entered.

```
ranked_sentences = sorted(((scores[i], s) for i, s in enumerate(sentences)), reverse=True)
```

Figure 16 - code snippet for ranking the sentences by the created scores

The Figure 16 shows the single line of code that sorts out sentecness and scores of them into a single maked list we canll **ranked_list**. We can the use this to extract out summary.

Summarization Options

☐ Automatic ☒ Manual ☐ Tag Optimized

24%

Text Input Options

☒ From File ☐ Input Text ☐ Example Text

Text File Path: No file selected.

Full Text:

A text file (sometimes spelled textfile; an old alternative name is flatfile) is a kind of computer file that is structured as a sequence of lines of electronic text. A text file exists: placing one or more special characters, known as an end-of-file marker, as padding after the last line in a text file. On modern operating systems such as Microsoft Windows end-of-line delimiters, which are done in a few different ways depending on operating system. Some operating systems with record-oriented file systems may not use new level of description, there are two kinds of computer files: text files and binary files. Because of their simplicity, text files are commonly used for storage of information. The text file, it is often easier to recover and continue processing the remaining contents. A disadvantage of text files is that they usually have a low entropy, meaning that the file may contain no data at all, which is a case of zero-byte file. The ASCII character set is the most common compatible subset of character sets for English-language text files; a must be used. In many systems, this is chosen based on the default locale setting on the computer it is read on. Prior to UTF-8, this was traditionally single-byte encodings; very small, many are only usable to represent text in a limited subset of human languages. Unicode is an attempt to create a common standard for representing all known languages has the advantage of being backwards-compatible with ASCII; that is, every ASCII text file is also a UTF-8 text file with identical meaning. UTF-8 also has the advantage that encoding when it definitely isn't UTF-8. On most operating systems the name text file refers to file format that allows only plain text content with very little formatting (e.g., encoding, MS-DOS and Microsoft Windows use a common text file format, with each line of text separated by a two-character combination: carriage return (CR) and line feed (LF) operating systems; a file is regarded as a text file if the suffix of the name of the file (the "filename extension") is .txt. However, many other suffixes are used for text files with Microsoft Windows text files use ".ANSI", ".OEM", ".Unicode" or ".UTF-8" encoding. What Microsoft Windows terminology calls ".ANSI" encodings are usually single-byte (SBCS) byte character sets. ANSI encodings were traditionally used as default system locales within Microsoft Windows, before the transition to Unicode. By contrast, OEM encodings applications. "Unicode"-encoded Microsoft Windows text files contain text in UTF-16 Unicode Transformation Format. Such files normally begin with Byte Order Mark (BOM) encoded files with BOM[2] to differentiate UTF-8 encoding from other 8-bit encodings.[3] On Unix-like operating systems text files format is precisely described: POSIX defines a printable file as a text file whose characters are printable or space or backspace according to regional rules. This excludes most control characters indicated that the type of the file was "TEXT".[7] Lines of Macintosh text files are terminated with CR characters.[8] Being certified Unix, macOS uses POSIX format for text file "public.utf16-plain-text" for utf-16-encoded text and "com.apple.traditional-mac-plain-text" for classic Mac OS text files.[7] When opened by a text editor, human-readable editor, or as visible escape characters that can be edited as plain text. Though there may be plain text in a text file, control characters within the file (especially the end-of-file

Summarised Text:

Number Of Sentences: 12 out of 52 (23.076823076823077%)

[For example, source code for computer programs is usually kept in text files that have file name suffixes indicating the programming language in which the source is written.] On Unix-like operating systems text files format is precisely described: POSIX defines a text file as a file that contains characters organized into zero or more lines.[6] A computer file that is structured as a sequence of lines of electronic text. In operating systems such as CP/M and MS-DOS, where the operating system does not keep track of multiple character encodings available for Unicode, the most common is UTF-8, which has the disadvantage of being backwards-compatible with ASCII; that is, every ASCII file is unambiguously a printable file as a text file whose characters are printable or space or backspace according to regional rules. This excludes most control characters indicated that the type of the file was "TEXT".[7] Lines of Macintosh text files are terminated with CR characters.[8] Being certified Unix, macOS uses POSIX format for text file "public.utf16-plain-text" for utf-16-encoded text and "com.apple.traditional-mac-plain-text" for classic Mac OS text files.[7] When opened by a text editor, human-readable editor, or as visible escape characters that can be edited as plain text. Though there may be plain text in a text file, control characters within the file (especially the end-of-file

Figure 17 - snippet of the whole screen with a full summarisation being processed on a Wikipedia page on Text Documents.

The Figure 17 is an example of a summarisation from a text file uploaded to the application. It was a Wikipedia page about the Text File extension, I used a manual summarisation value of 24% which returned around 23% sentences back which is 12 of the 52.

5.7 TAG OPTIMIZED SUMMARISATION

Summarization Options

☐ Automatic ☒ Manual ☐ Tag Optimized

Word tags separated by comma:

Text Input Options

☒ From File ☐ Input Text ☐ Example Text

Text File Path: No file selected.

Full Text:

A text file (sometimes spelled textfile; an old alternative name is flatfile) is a kind of computer file that is structured as a sequence of lines of electronic text. A text placing one or more special characters, known as an end-of-file marker, as padding after the last line in a text file. On modern operating systems such as Microsoft end-of-line delimiters, which are done in a few different ways depending on operating system. Some operating systems with record-oriented file systems may not use new level of description, there are two kinds of computer files: text files and binary files. Because of their simplicity, text files are commonly used for storage of information text file, it is often easier to recover and continue processing the remaining contents. A disadvantage of text files is that they usually have a low entropy, meaning may contain no data at all, which is a case of zero-byte file. The ASCII character set is the most common compatible subset of character sets for English-language must be used. In many systems, this is chosen based on the default locale setting on the computer it is read on. Prior to UTF-8, this was traditionally single-byte encoding when it definitely isn't UTF-8. On most operating systems the name text file refers to file format that allows only plain text content with very little formatting. MS-DOS and Microsoft Windows use a common text file format, with each line of text separated by a two-character combination: carriage return (CR) and line feed (LF) operating systems; a file is regarded as a text file if the suffix of the name of the file (the "filename extension") is .txt. However, many other suffixes are used for text files with Microsoft Windows text files use ".ANSI", ".OEM", ".Unicode" or ".UTF-8" encoding. What Microsoft Windows terminology calls ".ANSI" encodings are usually single-byte character sets. ANSI encodings were traditionally used as default system locales within Microsoft Windows, before the transition to Unicode. By contrast, OEM encodings applications. "Unicode"-encoded Microsoft Windows text files contain text in UTF-16 Unicode Transformation Format. Such files normally begin with Byte Order Mark (BOM) encoded files with BOM[2] to differentiate UTF-8 encoding from other 8-bit encodings.[3] On Unix-like operating systems text files format is precisely described: POSIX defines a printable file as a text file whose characters are printable or space or backspace according to regional rules. This excludes most control characters indicated that the type of the file was "TEXT".[7] Lines of Macintosh text files are terminated with CR characters.[8] Being certified Unix, macOS uses POSIX format for text file "public.utf16-plain-text" for utf-16-encoded text and "com.apple.traditional-mac-plain-text" for classic Mac OS text files.[7] When opened by a text editor, human-readable editor, or as visible escape characters that can be edited as plain text. Though there may be plain text in a text file, control characters within the file (especially the

Summarised Text:

Number Of Sentences: 5 out of 52 (9.615384615384617%)

[On modern operating systems such as Microsoft Windows and Unix-like systems, text files do not contain any special EOF character, because file systems on these encoded text, "public.utf16-external-plain-text" and "public.utf16-plain-text" for utf-16-encoded text and "com.apple.traditional-mac-plain-text" for classic Mac OS text files.[7] Lines of Macintosh text files are terminated with CR characters.[8] Being certified Unix, macOS uses POSIX format for text file "public.utf16-plain-text" for utf-16-encoded text and "com.apple.traditional-mac-plain-text" for classic Mac OS text files.[7] When opened by a text editor, human-readable editor, or as visible escape characters that can be edited as plain text. Though there may be plain text in a text file, control characters within the file (especially the

Figure 18 - snippet of the screen again but with tag optimization as the summary options being used

To further optimize the summarisation the user can include tags in the search. In Figure 18 I have used the same document about Text Files but instead applied the Tag Optimization option, this should have returned around 50% of the text back however, only around 10% were returned or 5 out of the 52 possible. This was due to the included tags "Microsoft, macOS" which were included in the options. This resulted in the text being summarised in the normal way, but the resulting sentences were further queries for the tags mentioned above. This lets the user really refine their summarised and what they are interested in as there is no limit on the number of tags allowed.

6 CONCLUSION

The web application demonstrates how a text summarization algorithm can be implemented successfully. We used a flask framework which allowed for rapid development of the web side of the project and allowed me to focus on the text summarisation. We can select different summarisation options to alter the final output of the text like how much we want

to compress the text by in percent or we can further optimize by refining with tags. We can input text in different forms like from local machine or by pasting text straight into the web allowing for multiple texts. Overall, the paper managed to detail the development of a prototype of a text summarisation web application. A final goal of commercialising such application may even be considered but this was only a proof of concept and it manged that which I am pleased with.

There is further work that could be done in the future. Some UI changes could have been made to improve the user experiences like allowing to upload multiple files at a time and merging them to do an overall summarisation of the texts. These would have to come from the same domains. Different word embedding could have ben tested to see if they improve the results of output summarizations. The biggest one is the switch to abstractive summarisation methods; these would require lots f work to develop but would allow for more authentic and true summarisations where the computer understand the texts and produces its own summarised version.

Trends in Engineering and Technology,
(pp. 109-110).

7 REFERENCES

- Bhaya, W. S. (2017). Review of Data Preprocessing Techniques in Data Mining. *Journal of Engineering and Applied Sciences*.
- EMMA L. TONKIN, G. J. (2016). *WORKING WITH TEXT - Tools, Techniquesand Approachesfor Text Mining*. Chandos Publishing.
- Jeffrey Pennington, R. S. (2016). *GloVe: Global Vectors for Word Representation*.
- Luhn, H. P. (1958). The Automatic Creation of Literature Abstracts. *IBM Journal*, 159.165.
- Vipul Dalal, D. M. (2013). A Survey of Extractive and Abstractive Automatic Text Summarization Techniques. *International Conference on Emerging*