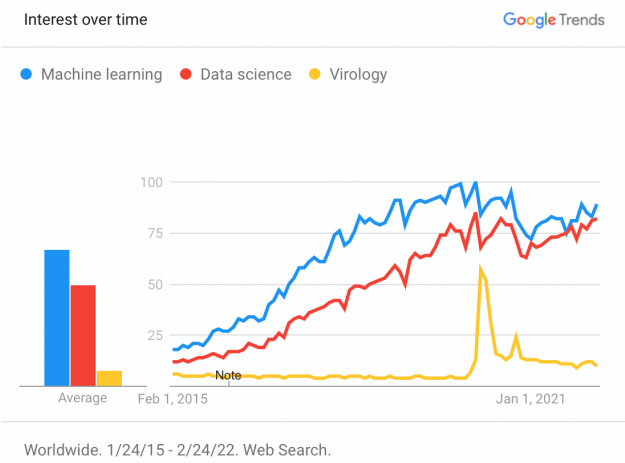
Identifying and summarising academic papers for efficient literature review.

BY: TAJALDEEP SINGH AHLUWALIA
JACOBS UNIVERSITY BREMEN
DATA SCIENCE FOR SOCIETY AND BUSINESS, 2021-23

Abstract

Aim of this paper is to analyse academic papers and summarising them in order to extract important information. The motivation is to be able to parse text to quickly gather insights about the content within the text.

There has been a positive trend in the number of papers published in many disciplines such as machine learning, data science, virology, etc[1]. The available literature to review is increasing and text summarisation can become a useful tool in helping further optimise the research process. The scope of this paper is to apply text summarisation for an input text from a set of sample articles while maintaining a certain level of human readability in the output.



In order to do data summary we currently have two major approaches: abstraction and extraction[2].

Data abstraction deals with the contextual meaning and hidden semantics in the document. The aim of this summary is to be able to be read by humans. This process includes complex sentences and paragraphs that are grammatically correct. In order to achieve this, deep learning algorithms and language models are used and produce an abstraction of the input text.

Extraction is a process of summarisation that uses weights and ranks the sentences. Words with high frequency are ranked higher in importance. One of the approach to extraction is identifying important sentences which have important words that are previously identified. The output is useful in identifying important phrases and words.

Some of the work that has been done in the field leverages commonly used extractive methods including: Term Frequency-Inverse Document Frequency Method[3], Cluster Based Method[4], Text Summarisation with Neural Network[5].

We will be using the extraction method in this paper to summarise the input text and produce the a-priori assumption for using this method is that the important and relevant information is often reiterated in documents. This allows such texts to have high frequency words that are correlated with their importance.

In order to do the analysis we show the use of rapid miner to quickly prototype a text mining algorithm that produces some early insights into the process and help us understand and fine tune the problem statement at this stage.

The next steps are to load the input file as a string and use NLTK package and python programming language in order to split and count the words in the document. The algorithm scores each sentence in the input by taking the total word counts and frequencies in each sentence. We then rank the sentences and produce an output of desired number of sentences.

The paper concludes the process and ends with further possible future use cases such as creating an interface that receives the pdf as an input and returns the summary to the user. However that is left as further steps and is not included in the scope of this paper.

Methodology

This paper examines and formalises how to extract patterns and uncover new knowledge using many of the approaches available, but on unstructured natural language rather than ordered data. This is the broad field of text and web mining.

Converting text into semi-structured data is the first stage in text mining. There's nothing preventing developers from using any of the analytics techniques to categorise, cluster, and predict unstructured text once it's been turned into semi-structured data. Patterns can be detected by converting unstructured text into a semi-structured data set, and even better, models can be taught to detect patterns in new and unread material.

The new frontier of data science is unstructured data (text, audio, photos, videos, and so on). In his book Predictive Analytics (Siegel, 2013), Eric Siegel makes an interesting analogy: if all the data in the world were equal to the water on Earth, textual data would be the ocean, accounting for the vast bulk of the volume. The requirement to handle natural human language drives text analytics, however unlike numeric or categorical data, natural language does not exist in an organised format with rows (of examples) and columns (of attributes). As a result, text mining falls under the category of unstructured data science.

Each of the main steps will now be analysed in detail, as well as some key terminology and concepts. However, before these procedures are presented, a few key concepts in text analytics must be defined.

This paper leverages the python libraries to convert PDF data downloaded from google scholar. The *textract* library is useful in order to convert the PDF into a text blob. The downside of this library is the missing formats and loss of paragraphing and structure of the document. The extracted text is a single blob and needs further preprocessing in oder to convert it into a structured format.

Using python, we extract the data for a sample of PDFs into a table format with each row representing the data. In order to do so, we loop the directory elements in python and extract the text from each file into a list while appending it. This allows us to get a single array of all the documents. The pandas method to convert the list into a dataframe allows easier handling of the data. The data is then exported into an excel format to be further processed and analysed using RapidMiner.

Converting text into semi-structured data is the first stage in text mining. There are many forms of techniques to categorise, cluster, and predict unstructured text once it's been turned into semi-structured data. In order to find patterns, the unstructured text must be converted into a semi-structured data set.

The paper then demonstrates the use of RapidMiner to process the semi-structured data. In order to understand the nature of data, the similarity algorithm is used to assess the data. Before the cosine similarity can be measured, there are few preprocessing steps that are performed on the data such as tokenisation, stemming, etc.

RapidMiner provides useful operators to create an efficient pipeline to create processed and structured data. The output of this process also helps in creating a cosine similarity matrix that helps to screen the different input research papers. Once the research papers are processed, the paper switches to a pure pythonic pipeline in order to achieve text summarisation.

Data Preparation

In order to generate dataset for this paper, a sample of research papers were downloaded from the google scholar related to the BCBS 239: The principles for effective risk data aggregation and risk reporting. The papers are downloaded in PDF format, therefore, they pose a primary challenge of data representation.

```
# import required module
import os
import pandas as pd
import textract
```

In order to extract the data the paper leverages the python programming language to transform the data. We leverage the existing modules for data processing and handling. This paper relies on pandas and *textract* modules to extract and process the PDF files.

```
# assign directory
directory = 'Literature'


# iterate over files in
# that directory
dir = []
for filename in os.scandir(directory):
    if filename.is_file():
        dir.append(filename.path)
```

After initialising the modules, we define the directory where the PDFs are saved. The OS native module in python is used to scan the directory and get the paths. The for loop is used to iterate over all the files in the directory. The empty list *dir* is then appended with the path of the files in the directory. After this process the dir object has a list of all the files in the selected directory.

```
# filter the files that
# have .pdf extentsion
dir = list(filter(lambda k: '.pdf' in k, dir))
```

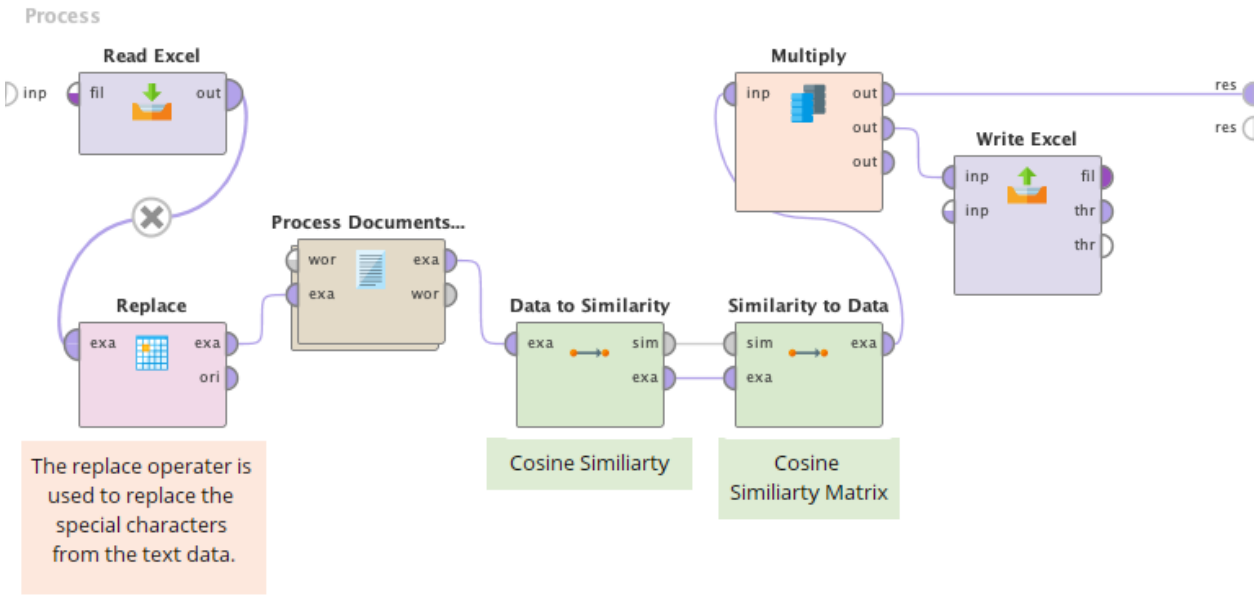
The text from the PDFs is extracted using the *textract* module. The loop appends the text from each PDF into a text list object using the 'pdfminer' method. The resulting list has all the text elements of the PDF. The next process is to simply create a pandas dataframe using the text and dir objects. The final dataframe is exported into an .xlsx format in order to preserve the text format. Using a CSV format is not a good practice in text mining as the CSV can have commas in the text that can result in irregular format.

```
# loop over the files and append
# them into a single list
text = []
for i in range(0, 30):
    text.append(textract.process(dir[i], method='pdfminer'))
```

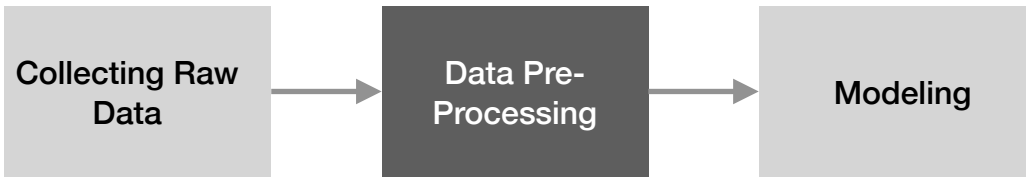
```
# create a single dataframe
# and export it into an excel file
a = pd.DataFrame(text, dir[:30])
a.to_excel("literature.xlsx")
```

Data Processing

In order to make use of the data, it needs to be preprocessed. RapidMiner is a great tool for quickly prototyping a data processing pipeline. The process will be highlighted in the graphics taken from the RapidMiner along with the description of the operators.



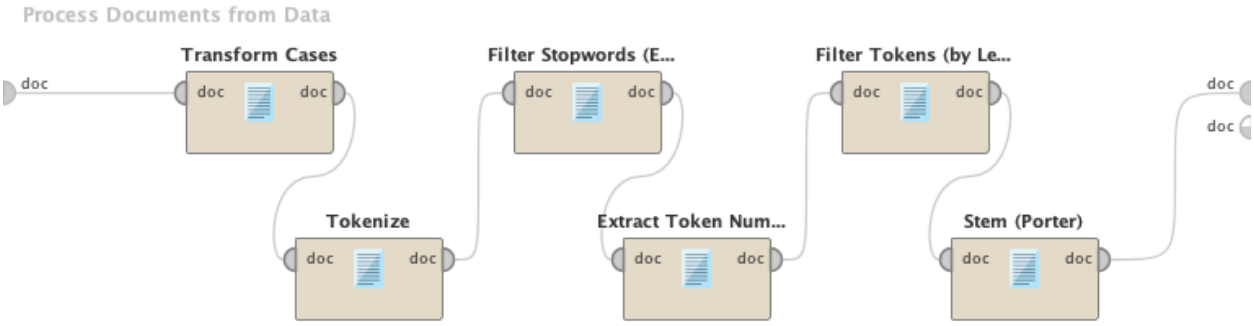
The high-level process for text mining is mentioned below. The raw data has already been collected. The data processing is the next step before any modeling can be done (eg. Classification, Clustering, etc.).



The overview of the RapidMiner process looks like the graphic below:
Data processing in Rapidminer is done after the data is loaded. The read excel operator loads the data to start this process. The next operator is the ***filter*** operator which is used to filter out special characters to clean the text data before the data processing can be done.

The ***process documents from data*** operator is used for word vector creation. **TF-IDF** scores are typically calculated in the preprocessing step of the three-step process described earlier for each word in the set of documents. The graphic below dives into the operator and reveals the preprocessing steps that are used in this paper.

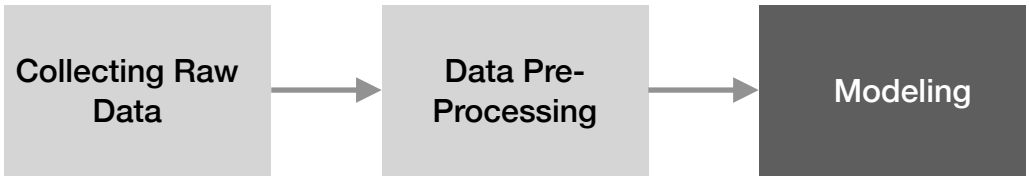
Before the data can be preprocessed, it is important to transform the data into lower case. This is done by using the ***Transfordiscretisingm Cases*** operator.



Each row is considered a separate document in the context of text mining. Words are also separated in the simplest case by a special character: a blank space. ***Tokenisation*** is the process of discretisdiscretisinging words within a document. Each sentence can be considered a separate document for the sake of this discussion.

The use of ordinary words like "a," "this," "and," and other comparable ones is worth noting. Clearly, a higher number of such phrases that do not truly communicate precise meaning would be expected in large documents. Before further analysis, most grammatical requirements such as articles, conjunctions, prepositions, and pronouns may need to be filtered. Stop words, which include most articles, conjunctions, pronouns, and prepositions, are examples of such terms. ***Stop word*** filtering is typically the next step following tokenisation. The token numbers are then extracted and filtered to omit small words with length less than 3-4 characters.

Words like "recognised," "recognisable," and "recognition" may appear in a variety of contexts, but they all suggest the same meaning. "Recognise" is the so-called root of all these highlighted words. The conversion of unstructured text to structured data can be simplified by reducing terms in a document to their basic stems, because only the presence of the root terms needs to be considered. This is referred to as ***stemming***. The Porter method is the most widely used stemming technique for text mining in English (Porter, 1980).



In order to understand the papers sampled for this project. Using Textual similarity can lead to some early insights into the nature of documents and the content inside them. There are many ways to find similarity between a set of documents. (e.g., word vectors from texts) Examples that have less distance are more similar. Some of the Possible methods are:

1. Euclidian distance = Δ Term Occurrences, square root of the squared sum.
2. Manhattan distance = Δ Term Occurrences summed up.

The general problem is the large difference if one text is much longer. This is the case with research papers as they are different in length. However, there are robust Methods against different text lengths:

1. Cosine Similarity. geometric angle between the vectors
2. divide by the size/length of the vector to normalise it

Cosine Similarity between vectors:

Cosine similarity is commonly used for comparing documents:

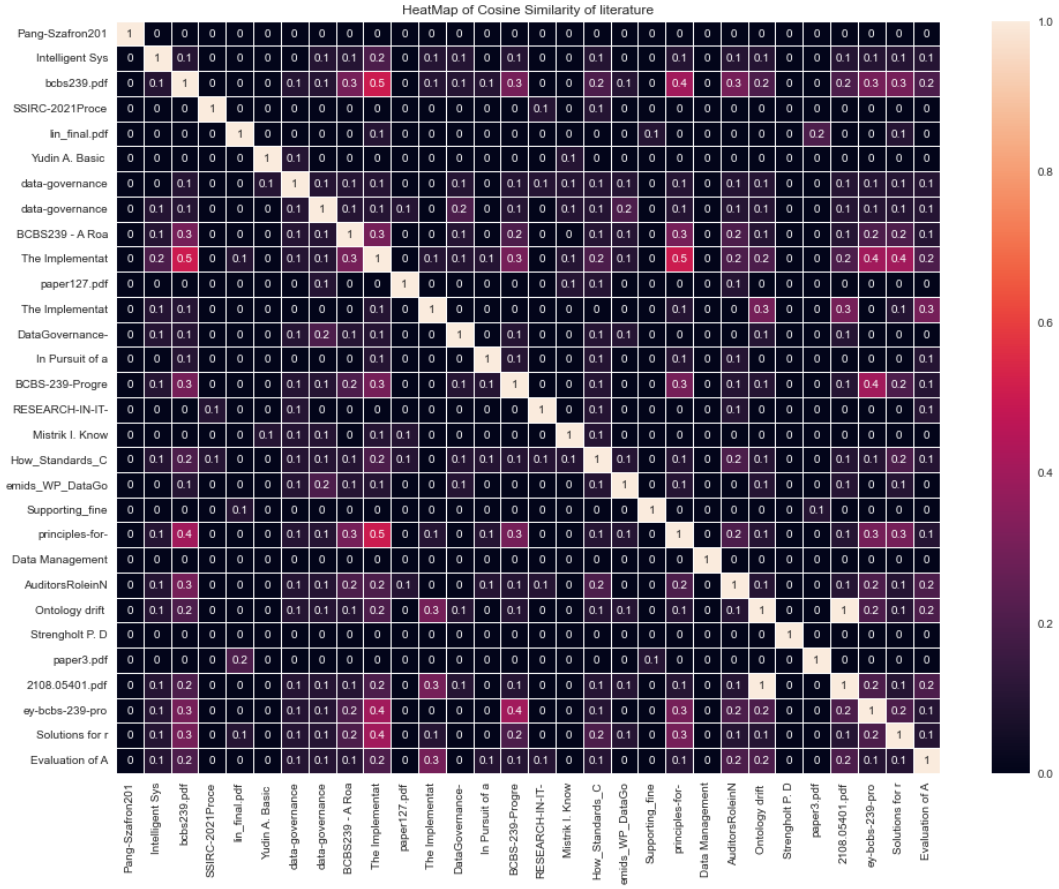
$$Sim(X,Y) = cos(X,Y)$$

Similarity = cosine of the angle between X and Y

If the vectors show exactly in the same direction (even different lengths) the angle is 0° and cos(X,Y) = 1
If the vectors are orthogonal (no common coordinates) the angle is 90° and cos(X,Y) = 0

The great thing about Cosines is that it can be calculated for any n dimensions. Which is particularly helpful in text data as the input vector is a highly dimensional space data. The ***data to similarity*** operator is used to calculate cosine similarity. However, the operator can also be used to calculate Jaccard similarity, Euclidian and Euclidean distances.

The output of this operator is then passed into ***similarity to data*** operator which yields a matrix with cosine score of each document with the others. However, this output format can be enriched by taking this matrix and saving it using the ***write excel*** operator. Python can be used to plot a heat map from the matrix. The output is easier to understand and also highlights the similar documents.



The code used to plot this heat map is fairly straight forward. *Seaborn* and *matplotlib* is used to plot the matrix.

In order to add extra context the labels are replaced by the actual name of the paper through passing the list through xticklabels and yticklabes arguments.

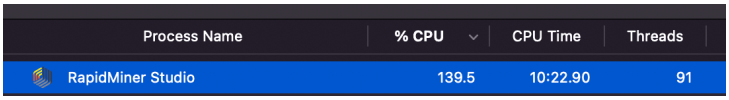
The result is a matrix that reveals the cosine similarity of documents.

```
# 1. Import Modules
import numpy as np
import seaborn as sns
import matplotlib.pylab as plt
plt.style.use("seaborn")

# 3. Plot the heatmap
plt.figure(figsize=(16,12))
heat_map = sns.heatmap(a.values, linewidth = .5
                       , annot = True
                       , xticklabels = label[:30]
                       , yticklabels = label[:30])
plt.title( "HeatMap of Cosine Similarity of literature" )
plt.show()
```

Limitations

As the paper discussed in the methodology section, RapidMiner still has limited capability to process PDF documents. The use of read excel is only made possible through the use of python library. Though there is a python operator that takes in python code and processes it, there is still no way to access directories and operating system commands in rapid miner. Due to this limitation, two parallel processes are required in order to create a pipeline for data analysis.



At the time of writing this paper, rapid miner turned out to be a very heavy application and required extensive use of RAM. The machine used for analysis is a MacBook Pro M1 Silicon with 8GB RAM.

The cosine similarity calculation alone along with the text preprocessing consumed more than 6GB of available RAM. Moreover, on a 16 core machine, rapid miner alone took 139.5% processing from a single core while running and the excess spilled to the other cores. This makes rapid miner a tool only to be used with much bigger machines. However, the quick and easy interface makes it much suitable for non-technical users.

The lack of ability to modify the output is also a limiting factor for RapidMiner. As in the case of cosine similarity matrix. There is no clear way to visualise it in a heat map. In order to overcome this barrier, we leveraged python libraries to visualise the matrix as a heat map. This allows the output to be more user friendly and intuitive.

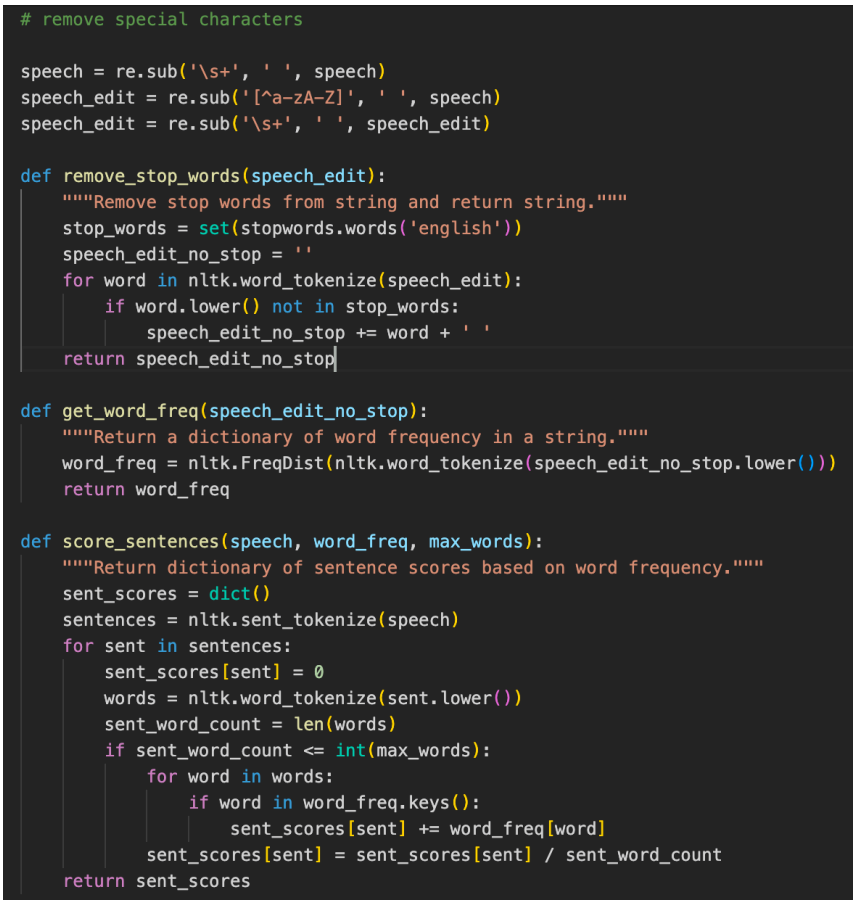
Conclusion for Rapidminer

The paper looked at the use of python programming language in conjunction with RapidMiner in order to carry out a data analysis pipeline. The following steps can be applied to any group of documents to find similarity between the documents and produce a human readable output that can be helpful in literature review and getting a general look at the available documents.

Text mining using python

We will generate 175 trillion gigabytes of digital data each year by 2025, according to International Data Corporation. However, the majority of this data—up to 95%—will be unstructured, meaning it will not be categorised into useable databases. Even now, the answer to a cancer cure may be sitting at our hands, but it is nearly impossible to obtain.

We must decrease the volume of data by extracting and repackaging key points into digestible summaries in order to make information easier to discover and absorb. There is no way to accomplish this manually because of the enormous volume of data. Natural language processing (NLP) fortunately assists computers in comprehending both words and context.



```
# remove special characters

speech = re.sub('\s+', ' ', speech)
speech_edit = re.sub('[^a-zA-Z]', ' ', speech)
speech_edit = re.sub('\s+', ' ', speech_edit)

def remove_stop_words(speech_edit):
    """Remove stop words from string and return string."""
    stop_words = set(stopwords.words('english'))
    speech_edit_no_stop = ''
    for word in nltk.word_tokenize(speech_edit):
        if word.lower() not in stop_words:
            speech_edit_no_stop += word + ' '
    return speech_edit_no_stop

def get_word_freq(speech_edit_no_stop):
    """Return a dictionary of word frequency in a string."""
    word_freq = nltk.FreqDist(nltk.word_tokenize(speech_edit_no_stop.lower()))
    return word_freq

def score_sentences(speech, word_freq, max_words):
    """Return dictionary of sentence scores based on word frequency."""
    sent_scores = dict()
    sentences = nltk.sent_tokenize(speech)
    for sent in sentences:
        sent_scores[sent] = 0
        words = nltk.word_tokenize(sent.lower())
        sent_word_count = len(words)
        if sent_word_count <= int(max_words):
            for word in words:
                if word in word_freq.keys():
                    sent_scores[sent] += word_freq[word]
            sent_scores[sent] = sent_scores[sent] / sent_word_count
    return sent_scores
```

NLP applications can, for example, summarise news feeds, evaluate legal contracts, research patents, assess financial markets, capture corporate knowledge, and provide study guides.

Text preprocessing

In order to use the raw text into any language models, the text has to be preprocessed. In order to have a better vector representation, a similar text preprocessing to RapidMiner is applied to the data in python.

The following code depicts a typical text preprocessing procedure in python. The process starts with removing stop words and special characters. The function then gets the word frequency and scores each sentence from the processed text.

```
while True:
    max_words = input("Enter max words per sentence for summary: ")
    num_sents = input("Enter number of sentences for summary: ")

    if max_words.isdigit() and num_sents.isdigit():
        break
    else:
        print("\nInput must be in whole numbers.\n")

speech_edit_no_stop = remove_stop_words(speech_edit)
word_freq = get_word_freq(speech_edit_no_stop)
sent_scores = score_sentences(speech, word_freq, max_words)

counts = Counter(sent_scores)
summary = counts.most_common(int(num_sents))
print("\nSUMMARY:")
for i in summary:
    print(i[0])
```

Summarisation

The application takes in the maximum number of words and sentences that are required as the desired output for the summary. Based on simple sentence score based on the word frequency, the program is able to summarise the entire text into a concise summary of desired words and sentences.

```
SUMMARY:
Experts in technology are not experts in the appropriate
public uses of technology.
WHICH DEMOCRACY?
Technology should not try to produce an appropriate
democracy; democracy should try to produce an appropriate
technology.
Will technology nourish or undermine democratic
institutions?
None of this denies the potential of the new technology for
efficient planning and information enhancement.
If democracy is to benefit from technology then, we must
start not with technology but with politics.
```

The algorithms works surprisingly well. It is able to use only the word frequency in order to score the important sentences. The summary is then displayed through a simply for loops that prints each sentence from descending order. The summary not only captures the title of the document, but it also captures the main points. However, if the program is run again with a sentence length of 10 words, several of the sentences are plainly too long.

This type of algorithm is good when there is a need for preserving the sentence structure and therefore allows the user to extract exact sentences. This approach is better when the context and the structure of the text has to be maintained.

Other summarisers:

However, there are different methods for text summarisation that can be used depending on the desired output. Some of these methods are presented in the code. We explore the following methods for text summarisation that can be used with python, The methods are part of the ***sumy*** package that can be found on https://pypi.org/project/sumy/:

1. **LexRankSummarizer**: a graphical based text summariser
2. **LsaSummarizer**: Latent semantic analysis is an unsupervised method of summarisation it combines term frequency techniques with singular value decomposition to summarise texts.
3. **LuhnSummarizer**: scores sentences based on frequency of the most important words.

The outcomes are logical, and users could utilise them to comprehend large texts and their contents in general. The algorithm that is chosen is a challenging decision because it produces a variety results that are subjective. This has made it much easier to describe the paper, but the engineer must also be familiar with the underlying statistics and mathematical implementation of each method in order to determine which one best suits your needs.

Conclusion

The paper used Rapidminer and Python in order to carry out the text analysis. The aim was to summarise text in order to extract important information for the document and this was achieved through the aforementioned steps.

Rapidminer was useful in carrying out text preprocessing and creating a similarity matrix based on cosine similarity between the different document. However, the software was not good at carrying out longer calculations with several documents. Moreover, the algorithms took a large amount of processing power.

Using python programming language on the other hand proved to be much faster and reliable. The PDF document reader was able to create a dataframe of all the documents with their titles. The TF-IDF and sentence ranking proved to be a fast method to summarise texts. The output is easily understandable and extracts the most important aspects of the document.

Code

THE CODE IS AVAILABLE ON GITHUB AT THE FOLLOWING LINK:
HTTPS://GITHUB.COM/TAJALAHLUWALIA/TEXT_SUMMARISER

References:

1. https://trends.google.com/trends/explore?date=2015-01-24%202022-02-24&q=%2Fm%2F01hyh_,%2Fm%2F0jt3_q3,%2Fm%2F080y1
2. Andhale, N., & Bewoor, L. A. (2016, August). An overview of text summarization techniques. In *2016 International Conference on Computing Communication Control and automation (ICCUBEA)* (pp. 1-7). IEEE.
3. García-Hernández, R. A., & Ledeneva, Y. (2009, February). Word sequence models for single text summarization. In 2009 Second International Conferences on Advances in Computer-Human Interactions (pp. 44-48). IEEE.
4. S. R. Patil and S. M. Mahajan, “A novel approach for research paper abstracts summarization using cluster based sentence extraction,” in Proceedings of the International Conference & Workshop on Emerging Trends in Technology - ICWET ’11, no. Icwet, p. 583, ACM Press, 2011.
5. K. Kaikhah, “Automatic text summarization with neural networks,” in 2nd International Conference on ’Intelligent Systems’., no. June, pp. 40– 44, IEEE, 2004.
6. Siegel, E. (2013). Predictive analytics: The power to predict who will click, buy, lie, or die. John Wiley & Sons.
7. Porter, M. F. (1980). An algorithm for suffix stripping. Program.
8. https://pypi.org/project/sumy/