A logo of a university

Description automatically generated

**CZ4052 Cloud Computing**

School of Computer Science and Engineering

AY23/24 Semester 2

**Cloud Computing Project 2024**

Chen Hongpo (U2022097E)

**Introduction**

TextRank is an algorithm based on PageRank, Google's web page ranking algorithm, for keyword extraction and text summarization. PageRank estimates importance of each website by creating a web graph, with the assumption that websites receiving more links from other websites are more important. TextRank utilizes such a graph, with words as the nodes instead. Combining this with NLP techniques such as keyword extraction, tokenization and TF-IDF, we create this algorithm for text summarization and generation. The rise of cloud computing has also enabled the much-needed scalability and parallelization for processing large amounts of text. This report will take a deep dive into the procedures and techniques applied in TextRank, and show a working example of it.

A diagram of a network

Description automatically generated

**Problem Statement**

With the proliferation of digital media, we are getting bombarded with an ever-increasing amount of information as publishers compete for our attention spans. We, as individuals, often lack the time to read entire articles or documents to determine their usefulness or our interest in them. Therefore, there is a growing demand for a system that can generate concise summaries of large amounts of text while retaining the key information.

The goal of this project and report is to answer to this demand and create a reliable program that utilizes the TextRank algorithm to summarise a text sample input given by the user. This input could be as large as a short novel, and we aim to leverage cloud computing technologies to efficiently process large amounts of data. Specifically, we focus on extractive summarization where we take the more important sentences in the text to form the summary. We also aim to test how different parameters in the algorithm will affect the results.

**Implementation**

Data and Preprocessing

We use sample text we found from the internet, taken from articles on common news sites such as the Straits Times.

We then preprocess the text for usage in TextRank. First, we tokenize the text into sentences and we iteratively process the sentences to change them to lowercase characters to ensure uniformity. We further tokenize the sentences into words to create a list of words that appear in the text, while removing stopwords which are commonly used words in the English language such as 'the', 'a', etc.

A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

TextRank Graph

We use the Term-Frequency Inverse Document Frequency (TF-IDF) word vectorizer to turn the list of words into a vector with words represented by fractions. TF-IDF is a statistical method that measures how important a term is within a document relative to a collection of documents .This matrix is transformed and multiplied by itself to get a similarity matrix, capturing the pairwise similarities between sentences. This matrix is then converted into a graph.

PageRank algorithm is applied to the graph to assign a score to each of the nodes depending on how many other nodes are pointing to it. With this, we have a list of sentences from the text sorted by importance based on the PageRank score.

When we ask for the summary, a custom fraction of sentences from the text is returned in order of importance.

A screen shot of a computer code

Description automatically generated

A screen shot of a computer program

Description automatically generated

Parameter Exploration

Other than the fraction of sentences returned, we can also tweak various PageRank parameters, namely alpha and max iterations., with their default values being 0.85 and 100 respectively. Alpha in this case is the probability of hopping in the graph to a new node (like in the surfer model).

Lower values of alpha make the algorithm more random, making the importance of the sentences more spread out. This could induce more variation in the summaries given. The maximum iterations determine the number of times the algorithm iterates to produce the importance scores. When it is too low, the values fail to converge, while too high of a iteration count results in unnecessary computations as the values have already converged.

Performance Evaluation

We use the evaluation metric ROUGE to determine the performance of our model. It stands for Recall-Oriented Understudy for Gisting Evaluation, and measures the similarities between 2 texts using word sequences. This is defined as ROUGE-N, which is the overlap of n-grams between the 2.

However, an issue we faced was the lack of reference summaries to compare to.

A screenshot of a computer

Description automatically generated

Integration with the Cloud

To enhance scalability with large amounts of text, we think a good SaaS model would be to make use of the cloud to leverage computing power we otherwise would not have access to on our local machine.

For this case, we decided to go with AWS Lambda, by setting up an API Endpoint as a trigger for the lambda function. It will be shown in the examples.

A screen shot of a computer program

Description automatically generated

**A screenshot of a computer

Description automatically generated**

**Illustrative examples**

This short story given to the algorithm

A screenshot of a computer screen

Description automatically generated

was summarised to A screenshot of a computer

Description automatically generated

-More examples in the jupyter notebook.

As for the Cloud part, an API endpoint was setup as a trigger for the Lambda which contained the TextRank code. This API endpoint had a route for authentication and a automatically deployed stage 'prod'.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

**Conclusion**

This TextRank algorithm implementation will be able to quickly summarise an article to save time for the reader, allowing them to decide whether it is worth a read. As we proceed into this digital era filled with information, this will be an important tool in saving us from wasted time.

Furthermore, integration with Cloud services such as AWS will enable this to be marketed as a SaaS, as the subscription model gains more and more popularity.

**Appendix**

* Code snippet for preprocessing, graphing, and pagerank
* # Tokenize sentences
* sentences = sent\_tokenize(text)
* # Preprocess sentences
* stop\_words = set(stopwords.words("english"))
* preprocessed\_sentences = []
* for sentence in sentences:
* words = word\_tokenize(sentence.lower())
* words = [word for word in words if word.isalnum() and word not in stop\_words]
* preprocessed\_sentences.append(" ".join(words))
* # Vectorize, create similarity matrix and graph
* vectorizer = TfidfVectorizer()
* tfidf\_matrix = vectorizer.fit\_transform(preprocessed\_sentences)
* similarity\_matrix = (tfidf\_matrix \* tfidf\_matrix.T).A
* graph = nx.from\_numpy\_array(similarity\_matrix)
* # Apply PageRank
* pagerank\_scores = nx.pagerank(graph, alpha=0.85, max\_iter=100)
* # Sort sentences by score
* sorted\_sentences = sorted(pagerank\_scores, key=pagerank\_scores.get, reverse=True)
* # Extract top N sentences as summary
* N = math.floor(len(sorted\_sentences)\*0.3) # Change this value as needed
* summary = " ".join([sentences[i] for i in sorted\_sentences[:N]])
* print("TextRank Summary:")
* print(summary)
* Code snippet for metrics and cloud integration
* import torchmetrics
* from torchmetrics.text.rouge import ROUGEScore
* metric = ROUGEScore()
* metric.update(summary, reference\_summary)
* fig\_, ax\_ = metric.plot()
* import requests
* # AWS API Endpoint
* api\_endpoint = "https://9ss0r8k9l6.execute-api.ap-southeast-2.amazonaws.com"
* def invoke\_api(input\_text):
* try:
* payload = {"text": input\_text}
* response = requests.post(api\_endpoint, json=payload)
* if response.status\_code == 200:
* result = response.json()
* summary = result.get("summary")
* print(f"Summary: {summary}")
* else:
* print(f"Error: {response.status\_code} - {response.text}")
* except Exception as e:
* print(f"Error: {e}")
* # Invoke on text
* invoke\_api(text)