ARTICLE SUMMARIZER

(Natural Language Processing using Python)

INTRODUCTION

Automatic summarization is the process of shortening a text document with software, in order to create a summary with the major points of the original document. Technologies that can make a coherent summary take into account variables such as length, writing style and syntax.

Automatic data summarization is part of machine learning and data mining. The main idea of summarization is to find a subset of data which contains the "information" of the entire set. Such techniques are widely used in industry today. Search engines are an example; others include summarization of documents, image collections and videos. Document summarization tries to create a representative summary or abstract of the entire document, by finding the most informative sentences, while in image summarization the system finds the most representative and important (i.e. salient) images. For surveillance videos, one might want to extract the important events from the uneventful context.

There are two general approaches to automatic summarization: extraction and abstraction. Extractive methods work by selecting a subset of existing words, phrases, or sentences in the original text to form the summary. In contrast, abstractive methods build an internal semantic representation and then use natural language generation techniques to create a summary that is closer to what a human might express. Such a summary might include verbal innovations. Research to date has focused primarily on extractive methods, which are appropriate for image collection summarization and video summarization.

How the Summarizer Works

This article summarizer is an extractive model that works based off word frequency scoring. It is built utilizing the NLTK Python library. Extractive summarization techniques produce summaries by choosing a subset of the sentences in the original text.

The frequency scoring system we will be implementing is as such:

1) Read from source — Read the unabridged content from the source, a file in the case of this exercise.
2) Perform formatting and cleanup — Format and clean up our format so that it is free of extra white space or other issues.
3) Tokenize input — Take the input and break it up into its individual words.
4) Scoring — Score (count) the frequency of each word in the input and score sentences based on word score.
5) Selection — Choose the top N sentences based on their score.

Requirements:

1) Python 3.2+
2) NLTK Library
3) NLTK Corpus

Project Setup

Begin with installing the NLTK module with Pip.

```
$ pip install nltk
```

Reading Content:

We will be using functions called “open()” and “read” that are responsible for accepting a file path and returning the entire contents in memory, or giving error if the file cannot be found or does not have read permission.

```
content_fd=open('example1.txt','r')
content=content_fd.read()
```

Data Cleanup:

After reading our data to the variable “content”, we now have to clean it. About 80% of the time in any data analysis project is spent working on data sanitization, collection, and normalization.

The sanitize_input command converts the different white space and new line characters to strip away and uses String.translate to do so. The goal is to replace any extra whitespace characters besides the one space after ending punctuation

```
content = sanitize_input(content)

def sanitize_input(content):
    replace = {
        ord('\f') : ' ',
        ord('\t') : ' ',
        ord('\n') : ' ',
        ord('\r') : None
    }

    return content.translate(replace)
```

Tokenizing

Now that the content is clean, we have to tokenize it by breaking it up into sentences and words.

```
sentence_tokens, word_tokens = tokenize_content(content)

def tokenize_content(content):
    stop_words = set(stopwords.words('english') +
list(punctuation))
    words = word_tokenize(content.lower())

    return [
        sent_tokenize(content),
        [word for word in words if word not in stop_words]
    ]
```

Stop words are common words like and or or that we do not want to have an impact on the scoring of the sentences for our final summary. So the stop_words variable contains a set of the stop words for English contained in NLTK and a list of common punctuation which is also provided by the toolkit. The word_tokenize function converts the text into lower case and returns a set of all the unique words. The sent_tokenize function splits the sentences in the text and returns them as separate elements in a list. Finally, this function returns a list of separate sentences using list comprehension and a list of all the unique words which does not include the stop words. The tokenized sentences is passed to the sentence_tokens variable and the tokenized words is passed to the word_tokens variable.

Scoring

Now the program contains a list of unique sentences as well as a list of unique words, so we can score the frequency each word occurring in the passage and use that to grade the sentences.

```
sentence_ranks = score_tokens(word_tokens, sentence_tokens)

def score_tokens(filterd_words, sentence_tokens):
    word_freq = FreqDist(filterd_words)
    ranking = defaultdict(int)
```

```
    for i, sentence in enumerate(sentence_tokens):
        for word in word_tokenize(sentence.lower()):
            if word in word_freq:
                ranking[i] += word_freq[word]

    return ranking
```

The word_tokens and sentence_tokens variables are passed to the score_tokens function. The FreqDist function is a function that accepts a list of tokens, like our filtered word list, and returns a structure where each key is the word and each value is the number of times that word occured. A defaultdict is initialized and we iterate over the sentences and increase their score based on the frequency of that particular word in the word_freq variable. If a word is present in a sentence, the ranking of the sentence is incremented by the frequency value of that word in the word_freq variable. The value of ranking will then contain key values of the sentence's numeric position, and their score.

Selection

After scoring is complete, we have to construct our summary from the N highest scoring sentences where N is our desired length. The value of the length can be changed according to the user requirement.

```
length=5

summary = summarize(sentence_ranks, sentence_tokens, length)
def summarize(ranks, sentences, length):
    if int(length) > len(sentences):
        print("Error, more sentences requested than available.
Adjust the length.")
        exit()

    indexes = nlargest(length, ranks, key=ranks.get)
    final_sentences = [sentences[j] for j in sorted(indexes)]
    return ' '.join(final_sentences)
```

The sentence_ranks dictionary , as well as the list of unique sentences in the sentence_tokensvariable and the length of the summary requested are passed to the summarize function.

The function then checks the length requested, is not longer than the total number of sentences available. If it is, we error out.
The nlargest function takes the sentence ranking data and turns it into a list containing the N(length requested) largest ranks of the of the sentences in the sentence_tokens variable. Then this list of indexes is used in a list comprehesion to put each sentence from the tokenized list into the final summary. sorted() is used to sort the list of indexes so the sentences will appear in the order of how they appear in the original passage. Finally these values are joined together into a string and returned as a summary.