Summarisation Report

fdqd43 – James Paterson

Background

Document and multi document summarisation is a relatively recent problem, created by the vast amount of digital data that is present online today, and the need to quickly understand it. The problem typically refers to taking in several documents on a similar subject, and then producing a shorter summary with all the important points of the original documents. The first types of summarisation attempted, and that include the TF and TF-IDF methodologies, are classified as statistical based algorithms **[1]** . These algorithms are noted as being good for general summaries, but often lack knowledge of the context and semantics of the content which limits their ability to produce coherent summaries. Improvements can be made to these methodologies through techniques such as stop word removal and refinement of the IDF measure. Apart from statistical algorithms, progress has also been made in other methods of summarisation, such as by considering the words of a document as a graph, or by looking for "cue words" such as "in conclusion" to further identify good sentences. But by far the most promising developments are that of machine learning. Using both supervised and unsupervised machine learning techniques allows quick feedback and refinement of an algorithm. However, these programs often need large amounts of data in the form of human written summaries that can be hard to come by. In this assignment I have implemented the TF and TF-IDF statistical algorithms with improvements to up the quality of the generated summaries.

Running the program

To run the program, you need to invoke "index.py" through the command line. It accepts these arguments:

```
python index.py -l 200 -i -p -f /files
```

`-l` Designates the maximum length of the summarisation to be generated. Default: **150**
`-i` Specifies that TF-IDF is to be applied. Does not require an argument. Default: **False**
`-f` Designates the directory for which the files are read from. Default: **/files**
`-p` Use this option to parse each article as its own text. Does not require an argument. Default: **False**

The program will automatically try to find files of the name "Doc X.txt" and run the summariser on the content in the directory specified.

Program Structure

index.py

This is the main file, and contains wrappers for the term frequency and inverse document frequency functions, as well as formatting the output and parsing the arguments.

common.py

The most important method is the **summarize** method, which as per the name, generates a summary of the text. The function takes in the original, uncleaned text of the articles, and applies the **weightFunction** to it. **weightFunction** is passed as a parameter to the function and is either the TF weighter or the TFIDF weighter. Both functions generate a list of 2-tuples where each tuple is of the form: `(sentence_index, sentence_weight)` in a list sorted such that the highest weighted sentence is at the start of the list.

The function then splits up the original article into sentences using the **splitSentences** function. The sentence splitting is done after removing newlines and the quote character, and then the text is split by the sequence "`. `". This was done to avoid the problem where acronyms such as U.S are split into multiple sentences, and where quotes of sentences muck up the splitter.

These sentences are then used to build the array *summarySentences*. We continue adding sentences to this array, until the length of the words in the summary is over the limit specified or until we run out of sentences. This calculation is done by the reduce statement in the while loop.

For each step in the loop, we first take the highest weighted sentence from the sentences and add it to *summarySentences* as a tuple `(sentence_index, sentence_text)`. We then delete this sentence from the document. Now, the weights and term frequencies are recomputed with the sentence removed for the next step. This is continued until adding the next highest weighted sentence would put us over the maximum summary length. To fill remaining space, we keep adding sentences from the document until we can't add any more without going over the limit.

The summary text is now constructed. This is done by sorting the sentences to be in the order that they originally appeared in the document. Each sentence is then appended to the last one with a reduce statement to build the summary, which is then returned.

Another important function in this file is **preprocess**. This function is used to turn raw data into a list of sentences that can be processed by the TF and TFIDF algorithms. The text is first split into sentences with **splitSentences**, and then every sentence is split into words with **splitWords**. This function turns a sentence into singular words. It achieves this by first stripping the sentence to avoid any white space on either side. The sentence is then converted to lowercase to avoid conflicts between capitalised and non-capitalised words. Punctuation is then removed from the string. The punctuation removed is in the array *CHARS_TO_STRIP*. Afterwards, the string is then split by the space character to get the array of words. Words that contain the apostrophe character are cropped such that only the first half of the string remains. For example, "it'll" is converted to "it". Any empty strings are then removed from the array. The data is then returned to the **preprocess** function. In it, the list of words is checked for stop words. Each word is compared against a list of stop words. If it's a stop word, it's removed from the list. These stop words are from NLTK and are stored in the array *STOPWORDS*. After this, the words are merged back into a single string and returned.

The last important function in this file is **calculateSentenceWeight**. The function takes in a pre-processed *sentence*, a summed list of term *frequency* and the list of *terms*. These are such that for some term `i`, `terms[i]` will return the actual term, and `frequency[i]` will return the frequency of that term in the document. First, the sentence is split up into words. Each word is then looked up in the terms list, and its frequency is added to the total sentence weight. The weight is then increased dependent on the content of the word. First, it's checked to see if it's numeric. If it is, it's weight is increased by one. Its weight is increased again by one if it's a four-digit number, as these are likely years. Each term is also compared against a list of months and abbreviations for months, stored in *DATE_STRING_LIST*. If it's one of these, then the weight is increased by one again. This is done to increase the chance we keep sentences with useful information like dates and figures. We then normalise the sentence weight by the number of words in the sentence to enable us to compare it to other sentences in the document.

The only other functions in this document are **loadArticle**, which loads an article given an id, and **numWords**, that counts the number of words in a string quickly.

tf.py

The code for the term frequency algorithm is contained in the tf.py file. The file is mainly a wrapper for the **tf** weighter. This first pre-processes the text using **preprocess**, and then obtains a list of terms and a frequency matrix from **calculateTermFrequency**. This is a standard implementation of the TF algorithm – First a list of features (words) is extracted from the document. A zeroed matrix is then created, where every row represents a document (sentence), and every column represents some word. A number `x` in the cell `(i, j)` means that `x` occurrences of the term `j` were found in the document `i`. This number is calculated for every cell by iterating over the words in each sentence, looking up `i` in the feature list, and incrementing the relevant cell. The resultant matrix and term list are then returned to **tf**. The **tf** procedure then uses this frequency matrix to calculate the weight of every sentence in the document with **calculateSentenceWeight**. This weight data is then returned in the format mentioned earlier, a sorted list of 2-tuples where each tuple is of the form: `(sentence_index, sentence_weight)`.

tfidf.py

The TFIDF procedure works from the previous term frequency algorithm, by first generating the frequency matrix using **calculateTermFrequency**. This frequency matrix is then modified using the method **transformFrequencyTdidf**. This procedure sums the number of documents each term is mentioned in, and

then uses this to calculate the idf score for every element. Each element is then multiplied by it's respective idf score, and the matrix is returned. The list of sentence weights is built in much the same way as before in the **tf** method. The IDF measure helps to include terms that are relatively infrequent but contain lots of semantic meaning, as they are not mentioned often in other similar documents.

Results

Below are some comparisons between the TF and TFIDF algorithms for a 150-word sample. The differences between the samples are highlighted. Full 150-word summaries for all 8 articles are contained in the file *summaries_each.csv*. Summaries for the whole article of 100, 250 and 500 words are in the file *summaries_together.csv*.

Term frequency	**Term frequency inverse document frequency**
It seems surreal to me, said Elon Musk, proprietor of SpaceX, and for once he was understating things. FALCON Heavy, the highly anticipated SpaceX rocket, blasted into orbit on Tuesday as Elon Musk rewrote the history books with the incredible feat. Here are all the facts you need to know about the Falcon Heavy and Tesla roadster launch. There was some confusion that Elon Musk was sending the Tesla to Mars. Falcon Heavy's successful launch opened a new chapter for SpaceX. For the meantime, SpaceX continues to render services through the Falcon 9 and Falcon Heavy. The the maiden launch of the Falcon Heavy rocket system later this month, will, among other things, send Musk's old Tesla Roadster to Mars. Nowhere did Tuesday's launch of Elon Musk's SpaceX Falcon Heavy rocket echo as powerfully as in Russia, says Bloomberg's Leonind Bershidsky.	It seems surreal to me, said Elon Musk, proprietor of SpaceX, and for once he was understating things. Conceivably, the Falcon Heavy could even transport people to the moon, at a fraction of the expected cost of an SLS launch. Here are all the facts you need to know about the Falcon Heavy and Tesla roadster launch. There was some confusion that Elon Musk was sending the Tesla to Mars. Falcon Heavy's successful launch opened a new chapter for SpaceX. For the meantime, SpaceX continues to render services through the Falcon 9 and Falcon Heavy. The the maiden launch of the Falcon Heavy rocket system later this month, will, among other things, send Musk's old Tesla Roadster to Mars. Nowhere did Tuesday's launch of Elon Musk's SpaceX Falcon Heavy rocket echo as powerfully as in Russia, says Bloomberg's Leonind Bershidsky.

The two algorithms produce very similar output – Most likely because they are both based on the TF algorithm. However, from reading the outputs for multiple articles, TFIDF tends to produce slightly more readable summaries, often including sentences with dates and years that are not in the TF output. This is likely to be because of the compounding between years and dates being relatively infrequent, and the extra weighting given to these types of data. However, the outputs only ever tend to differ by one or two sentences, and both tend to be readable.

I have also tested the data on different lengths of data. Shorter summaries of around 100 words are typically not readable for both pieces of data, and do not contain many pieces of relevant information. This is likely because the large amount of data simply cannot be compressed into this space, as there is not even space in 100 words for a single sentence from each document.

Summaries of longer lengths, around 250 words, are far more readable. Apart from repetition of information (for example "The maiden launch of the Falcon Heavy rocket system later this month, will, among other things, send Musk's old Tesla Roadster to Mars.", and "The rocket is carrying Elon Musk's red Tesla Roadster." communicate basically the same information), the summaries generally flow well. This issue was addressed by rerunning the classifier with each sentence removed, however there is clearly a need to add further processing, such as decreasing sentence weight for sentences like those already included in the summary. There is also an issue with sentences that do not carry much meaning but have lots of high frequency words in being included, for example "Here are all the facts you need to know about the Falcon Heavy and Tesla roadster launch.", which does not actually communicate much information.

In general, the longer the summary gets, the more readable it is. Summaries of 500 words are very readable and contain lots of figures and dates, especially when TF-IDF mode is enabled.

[1] Lloret, E. & Palomar, M. Artif Intell Rev (2012) 37: 1. https://doi-org.ezphost.dur.ac.uk/10.1007/s10462-011-9216-z