CE306/CE706 - Information Retrieval
Assignment 1: Indexing for Web Search

Drew Naylor (dnayloa – 1504692)

Cieran Almond (ca16873, 1604959)

Software Used:

- Python 3.7
- Unix (Vim)

Packages imported:

- Urllib.request – a library that helps in opening of URLs.
- Nltk – a library that helps with processing text data. It includes text processing libaries that we later use in our code such as classification, tokenization, stemming, tagging, parsing etc.
- Beautiful Soup – a library for pulling data out of HTML and XML files.
- Json – a library for writing processed data to a json format.
- Pandas – a library for data processing and digestion
- Sklearn – a library for machine learning and other data science tools

Implementation:

This program has been tested in a Linux and Windows based environment.

Please install the following modules before you run the program.

NLTK requires Python versions 2.7, 3.5, 3.6, or 3.7

For installation please run:

Install NLTK: run pip install --user -U nltk

Install Numpy (optional): run pip install --user -U numpy

Test installation: run python then type import nltk

Install beautifulsoup4

We configured a Git repository (found here: https://cseegit.essex.ac.uk/dnayloa/ce306-assignment-1) so that we could both contribute and commit code remotely since we were working as a pair.

Once that was configured, we analysed the requirements of the assignment. We came to an agreement that the order in which we do things should follow this general format:

HTML Parsing> Preprocessing> Stemming> Speech Tagging> Selecting Keywords

Once this was decided, we created a *main.py* file in our Git directory. This would be the main file we would run the code from.

We created a class named *Ass1*, and empty placeholder functions configured like so:

```
def grab_html():

        pass
```

An example of the first function we created within our assignment 1 class

This just allowed us to structure the order of tasks (functions) in which we thought was most logical.

We both decided to do some background research to see if we could find any libraries which may be useful to us in order to complete task 1 (grabbing/reading in HTML from the provided links).

It was at this point where we found *"BeautifulSoup and NLTK".* After some deliberation and comparison of other alternative libraries, we decided these would be the best libraries to use.

From this point the general development of the project followed the same format:

Anaylse Task > Create Empty function in the *Main.py*> Anaylse useful libaries that assist in the completion of that specific task> Implement code

For example, completing our *"Stemmer"* task was implemented like so:

1. Understand that stemming the words is a requirement needed to complete the task.
2. Create placeholder stemming function within *Main.py*

```
def stem_words():

        pass
```

3. Research libraries that would assist in completing the task. In this instance we imported *“PorterStemmer”* from NLTK.

```
from nltk import PorterStemmer
```

4. Complete stem words function

```
def stem_words(self, text_list):
    for word in text_list:
        text_list[text_list.index(word)] = self.stemmer.stem(word)
    return text_list
```

5. Test words were stemmed by printing the original text folder, then calling the stem words function and compare the output in the terminal.
6. Once we had tested the stem words function worked correctly as intended, we would write the output to a text document.

Implementation followed this process for each of the functions we created.

Code Description:

Note all our code is commented using best practise described on the website: https://realpython.com/documenting-python-code/

```
class Ass1:
```

This class is responsible for housiong all functions related to this assignment. Once the *main* within *Ass1* is ran, it generates a .txt file for each function in order to demonstrate completion of the task.

def grab_html(self, url):

fp = urllib.request.urlopen(url)

mybytes = fp.read()

html = mybytes.decode("utf8")

fp.close()

return html

This function takes the defined URL and opens and decodes it using UTF8.

```
def stem_words(self, text_list):
        for word in text_list:
            text_list[text_list.index(word)] = self.stemmer.stem(word)
        return text_list
```

This function uses a nltk library called snowball to stem all of the words.

```
def remove_stopwords(self, text_list):
        relevant_words = []
        for word in text_list:
            if word not in self.stop_words and word != "":
                relevant_words.append(word)
        return relevant_words
```

This takes the stemmed list and removes stop words defined in the nltk library stopwords.words("english")

```
    def remove_punctuation(self, text):
        for word in text:
            if word in self.punctuation:
                text = text.replace(word, " ")
        return text
```

This takes the processed text and removes punctuation defined in a list by iterating through a for loop, and removing each occurrence of the punctuation defined below.

self.punctuation = "–!\"#$%&()+-.,/:;<=>?@[\]^_`{|}~\n"*

```
def remove_single_words(self, text_list):
        for word in text_list:
            if len(word) > 1:
                text_list[text_list.index(word)] = text_list + " " + word
        return text_list
```

This takes the processed text again and removes words that are 1 character in length, as this was considered jargon that didn't provide any relevance to the text.

```
def writer(file_name, text):
    f = open(file_name, "w")
    if (isinstance(text, list)):
        for i in text:
            f.write("%s\n" % str(i))
    else:
        f.write(text)
    f.close()
```

This writes the strings of each function to a .txt folder once called in the main.

```
def __init__(self):
        self.stemmer = SnowballStemmer("english")
#Stemmer object
        self.stop_words = set(stopwords.words("english"))
#Stop words list from nltk
        self.punctuation = "–!\"#$%&()*+-.,/:;<=>?@[\]^_`{|}~\n"
```

Initialize function defines a list of punctuation that we will target to remove, stop words defined in English language from the library Snowball.

```
def main():

    vocab_list = []
#list of strings that are in the documents

    ass1 = Ass1()

    url_f = open('url.txt', 'r')
#list of urls to be checked

    for url in url_f:

        try:

            soup = BeautifulSoup(ass1.grab_html(url), "lxml")

            writer("html.txt", str(soup))


            #removing js under the script tag from html

            for tag in soup(["script"]):

                tag.extract()


            #removing css under the style tag from html

            for tag in soup(["style"]):

                 tag.extract()


            text = soup.get_text()
#remove all other html tags, leaving raw text

            writer("text.txt", text)


            sentence = sent_tokenize(text)
#splitting the sentences

            writer("sentences.txt", sentence)


            text = ass1.remove_punctuation(text)
#removing punctuation
```

```
            writer("removed_punct.txt", text)


            text = text.lower()
            text_list = text.split(" ")
            text_list = ass1.remove_stopwords(text_list)
#removing stopwords
            text = " "
            text = text.join(text_list)
            writer("removed_stopwords.txt", text)


            vocab_list.append(text)


            text_list = ass1.stem_words(text_list)
#stemming words
            text = " "
            text = text.join(text_list)
            writer("stemmed.txt", text)




            tokens = nltk.word_tokenize(text)
#seperating text to part of speech
            tokens = nltk.pos_tag(tokens)
#displaying words with part of speech tags
            writer("part_of_speech.txt", tokens)


        #handling http errors
        except(urllib.error.HTTPError):
            print(url[0:-1] + colored(" http error", "red"))
        except(UnicodeDecodeError):
            print(url[0:-1] + colored(" utf-8 error","green"))
```

```
    '''Performing tfidf on the list of document
vocabularies'''

    #TF-IDF

    tfv = TfidfVectorizer(use_idf=True)

    tfidf_vector = tfv.fit_transform(vocab_list)

    #convert to pandas dataframe for formatting

    tfidf_vector = pd.DataFrame(tfidf_vector.T.todense(),
index=tfv.get_feature_names(), columns=['url1', 'url2'])

    print(tfidf_vector)

    writer("tfidf.txt", tfidf_vector.to_string(header = True,
index=True))
```

This is the main of Class Ass1. It has calls to each function which performs different forms of data processing on the supplied URLs. Each function then writes it's output to a appropriately named .txt folder for analysis.

During the pipeline we output the formatted text to a list for the tfidf process.

The final part performs the tfidf calculations using Sklearns.tfidfVectorizer object. This is then converted to a pandas DataFrame for formatting and written to a text file.

Output from two supplied webpages (each stage):

- **HTML parser: text.txt**
- **Sentence splitter: sentences.txt**
- **Stopword: removed_stopwords.txt**
- **Removing Punctuation: removed_punctuation.txt**
- **Part-of-Speech Tagging: part_of_speech.txt**
- **Stemming: stemmed.txt**
- **TF*IDF: tfidf.txt**

Discussion:

A short list of features that could be implemented to better clean our text:

- Handling large documents and large collections of text documents that do not fit into memory. This could be improved using lighter-weight libraries, and reducing redundancy in our code.

- Extracting text from markup like HTML, PDF, or other structured document formats. At the moment BeautifulSoup is only designed to scrape XML type documents.

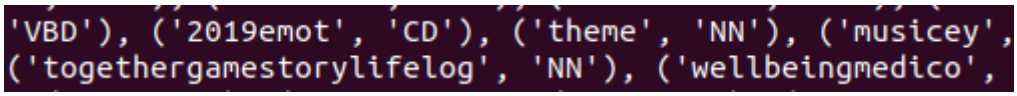
- Transliteration of characters from other languages into English. When we tested our program with other URLs, we also used pages that used different languages. When this was performed, the different languages were not processed since our program is only designed to perform text processing on English words.

- Decoding Unicode characters into a normalized form, such as UTF8.

- Handling of domain specific words, phrases, and acronyms.

- Handling or removing numbers, such as dates and amounts. This is another problem that we experienced first hand, for example “100” should be processed into English word format “One Hundred” in order to assign it better contextual relevance.

- Locating and correcting common typos and misspellings. We attempted to find a way to remove jargon words, though when we tried implementing this, we found that it also was removing words that could be considered relevant. Therefore a future improvement would be to remove jargon words. An imaged example of our “relevant jargon words” were the following:



As you can see, ‘togethergamesstorylifelog’ should actually be processed as separate words:

- Together
- Game
- Story
- Life
- Log

- Stemming : Can instead use Lemmatization. This would take into consideration the morphological analysis of the words. To do this however, it is necessary to have detailed dictionaries which would then have to be indexed to look through to link the form back to its lemma.

  This would also have to include dictionaries of every language, not just English, to provide an accurate analysis.

- Runtime of the program is quite long for processing only two documents, if we were to process more than two webpages, a longer runtime would have to be taken into consideration.

- The tf-idf suffers because our vocabulary is actually quite poor. We need to add more words to our stopwords to remove certain frequent words that are bespoke to this collection. For example a list of documents about MRI scanners may need Magnetic Resonant Imagining removes as this is relevant in all documents and therefore useless.

References:

- **https://pandas.pydata.org/**
- **https://scikit-learn.org/stable/**
- **https://www.nltk.org/**
- **https://www.crummy.com/software/BeautifulSoup/bs4/doc/**