CE306
Information Retrieval
Assignment 1

Alex Ebbage
1504283

Contents

1 Description of Implementation

1.1 Usage

Using Python 3.6, open the CMD and enter the command:

python 1504283.py <URL>

Where <URL> is replaced with the URL you wish to process. If the URL is valid, it'll begin the processing pipeline, outputting the results of each stage. Once the program has completed, a file will be produced showing the results of each stage for the pipeline, as was shown by the console. At the end will be the keywords and phrases found for the URL listed in alphabetical order.

1.2 Input

My Python script takes a URL as a command line argument. If there is more or less than 1 argument given then the program will throw an error and exit. It'll then attempt to process the given URL. Initially I used an array which held all the different URLs and just allowed the program to iterate through them, processing each individually, but I decided to add the command line functionality so a user can select their own URL to process.

1.3 HTML Retrieval

Once a valid URL has been given, the program uses it as an argument in a method which uses urllib, a module that allows data to be collected from a URL, to collect all the source code from the URL. This data is then parsed using BeautifulSoup, which comes equipped with several parsers. After the "soup" is collected, it is returned and then passed to the next stage.

1.4 HTML to Plaintext

Using the soup from the previous stage, I wanted to separate different components of the web pages data. First, meta-data is collected from the meta-tags by looking at their attribute names, first for "keywords", then for "description" if the first is unavailable. This information is simply stored into an array for processing later.

Next, in order to get useful text later on, tags which hold useless text need to be omitted. To do this, the decompose method from BeautifulSoup was used to omit all the script tags. Even though the script contents aren't seen on the website it'll get picked up by the get_text method which BeautifulSoup utilizes. Typically, JavaScript is what you'd find between the script tags, but it is unlikely that any useful data could be gained from it, hence the removal of it.

Following this, the title is found using the same method as for the meta-tags and its contents are stored in the same manner; a tuple holding a tag type and the contents. In the same block of code, the heading tags are all found and then iterated through, using the extract method to remove the data from the html code at the same time; preventing duplication of data later on. The headings are also stored in tuples indicating that the data is from header tags.

Finally, the rest of the web text needs to be collected. I tried several methods for this but found that doing it by removing the key information then getting the rest using the get_text method produced the best result. The text gained from the method had a lot of excessive whitespace due to the format of the HTML source code, so I used regex's substitute method to remove large gaps, cleaning the code up

for presentation at the end of the stage. Typically, I wouldn't do this as I wouldn't show my whole process so it'd never have to be presented. This is all returned in an array, ready for the next stage.

1.4 Tokenizing the Plaintext and Normalization

In this stage, the plaintext attained in the previous stage is passed as an argument. Then, a for-loop for each pair of data within the array is started. At the start of each loop the tag and data is taken for the pair, converting the data to lower case to normalise it. Additionally, some characters are replaced within the data to produce better results later on in the pipeline. The first change is the remove all the apostrophes; an effective rule which is used by TREC collections. The others changes which I found to improve results where to remove "+", ":" and "/".

Next, depending on the tag from the previous stage; indicated by the first item in the data pair, I perform one of 3 algorithms to end up with the same tuples as before but with tokenized data. For any pairs with the "meta" tag, I replaced any newline tokens with a space, then, split the lines by a comma. This gave me individual sentences or phrases, which I then iterated through to tokenize. The tokenized data along with the "meta" tag were placed as a tuple in a new array. The reason I chose to split the data is that keywords and description information is supposed to be written as keywords or key phrases which are separated by commas. So, by doing it this way, I can simply add the phrases/keywords to the index array with minimal processing later on.

For the title and header tags I simply used NLTKs word tokenize method on the data, as it should already be in sentences. This was appended to the new array like the previous data. Finally, the contents was split into sentences using NLTKs sentence tokenize method, then each sentence was looped through and converted into tokens in the same manner as the title and header tags. The reason I split the contents into sentences first is because when POS tagging is done, is uses the information gained from the neighbouring tags to determine what value it receives, so by trying to get sentences broken down as precisely as possible, I hoped it would produce a better result. Otherwise, ambiguity would be harder to avoid and the tagger may assign the wrong values to tokens. The array containing all the new tokenized tuples in then returned for the next stage.

1.5 POS Tagging Tokens

The data from the previous stage is passed as an argument, then, a for-loop which iterates through each pair of the data is created. The loop stores the tag and data from each pair, then NLTKs POS tagging function is used on the tokenized sentences. This method returns a POS tagged sentence which converts each token into a tuple holding the text value and the POS tag. Each pairs HTML tag and list of POS tagged tokens was appended to a new array, which is returned at the end of the stage.

1.6 Select Keywords and Phrases

One of the more important and time consuming stages of my processing pipeline. It takes the tagged data as an argument for processing, then, initializes a series of variables for assisting with the selection process. Firstly, the tag weightings are held within a dictionary, so the tag can be given as a key to receive a weighting value. The values assigned were played around with the produce what I thought were better results. Next, a list of English stop-words produced by NLTK which will be used to ignore unimportant words when counting word frequency later on, such as determiners. The WordNet lemmatizer is also initialized so it can be used to lemmatize words within noun phrases later in this stage. Finally, a string of POS grammar is created which contains all the combinations of POS tags that produce noun phrases.

The data pairs are iterated through like in previous stages, but this time, as well as the data and tag information being stored, the weighted value is stored by using the tag for the given pair and the dictionary I made earlier. First, the tag is checked if it is a “meta” or “title” tag, as I believe these should automatically be assigned as keywords or key phrases, though initially I was going to use them as heavily weighted terms. If the length of the data array is 1, then it is appended to the keywords array, otherwise, the words in the data are lemmatized using NLTKs lemmatizer on each word. Then, the phrase is combined with spaces in between each word and finally is appended to the key phrases array.

Next, for the data pair, a sentence chunker is created using the regex grammar made at the start of the stage. The data for each pair is passed through the chunker and all resulting sub-trees with the noun phrase tag are iterated through. Trees with 2 to 4 leaves are lemmatized, then, combined to produce a noun phrase string. If the phrase is 3 words, the weighted value is multiplied by 3; the reason for this is that phrases produce more useful information that singular words, however, I found that 3-word phrases were more useful with the given URLs than the 2 and 4-word phrases. The phrases then got added to the phrase frequency dictionary along with their weighted values.

Following this, each sentence had its punctuation tokens removed; this was to save time ignoring unnecessary tokens. Then for each of these words, if the word wasn’t in the stop words list and had a noun, adjective or verb POS tag, then it gets lemmatized and added to the word frequency dictionary, along with its weighted value. I found this produced a better result than when I used the NLTK stemmers, but will cover this later in the report.

Now, for where the frequency dictionaries and stats come in. First, I looked at the keywords. I iterated through the word frequency dictionary items, ignoring the values below 1 and finding the average value of the remaining items by getting the sum value and dividing it by the number of items. Values which occurred very frequently are often not useful indexes and so I devised a value which ignored the high frequency words. This was done by dividing the number of the words with a value greater than 1 by the average gained earlier. Then the word frequency dictionary was sorted in reverse order and iterated through, where any words with a value between the average and the cap were added to the keywords array. Also, any words with fewer than 4 letters were ignored, as they were typically not useful.

Next, a similar method was used for the key phrases, except the top 6 most common phrases were added to the key phrase array instead. This was an arbitrary value which would likely be changed if more websites were looked at and processed. For both the previous methods, if a word or phrase was already in one of the key arrays; if it was in the title or meta tags, then it would be ignored, as I didn’t want duplicate values in the final array. These arrays were then returned for the next stage.

1.7 Lemmatization

The method takes a word and its POS tag as an argument. It then uses WordNet’s lemmatizer and its tags to produce a lemma from the word. The POS tags first letter is taken and then a WordNet tag is selected based on the input. The lemmatizer is used in conjunction with the WordNet tag to produce a lemma which is then returned.

1.8 Finalization

This final stage takes the output from the previous stage and an empty array is created to store the final values. The keyword and key phrase arrays are looped through and appended to the key values array. The values are then sorted into alphabetical order and returned.

1.9 Output

There are 2 forms of output for my program. The first being through simple print statements and the second by writing the results of each stage to a text file. Both are handled through a single method which takes a title, the data to be outputted, an integer indicating the output algorithm to use and a reference to the file to output to, as arguments. The file is opened at the start of the program, using the title gained from the URL title tag as the filename, then is closed after the final output is done.

Within the method, there are several if-statements which are selected through the integer passed as an argument. At the start the title is written between 2 lines, producing a heading to each section. Then, one of a variety of code blocks is selected to convert the data to a format which can be written to a file, as it isn't as simple as printing data, unfortunately.

1.10 Main Routine and Comments

The main routine contains the command line input and then the segmented stages of the entire processing pipeline. At each stage the call to the output method is made using the data returned from the given stage for every stage. Keeping the code in stages like this makes it easy to make changes to a given stage without worrying about changing all subsequent methods.

Every method has a comment block about it explaining what it does, what arguments it takes as input and what is returned as an output. Some additional in-line comments have been made for more complex parts of code, but otherwise the use of self-explanative variable and method names should make it easy to follow. Methods are structured in the order that the processing pipeline follows and all relevant information is kept together.

2 Tools Used

2.1 URLLib

This module makes it easier to open URLs and retrieve the raw source code from a webpage in an ASCII text format. It provides some additional utilities but nothing required for this assignment. I used it to request the data from a URL and then to read the data received.

2.2 BeautifulSoup

This module allows data to be retrieved from HTML and XML files. It provides utilities to navigate and modify the parse tree. I used several of its functions, but mainly the find and find_all functions, which return a series of information relating to the given information. The HTML parser is provides saves a huge amount of time getting all the content from an HTML file as well, without it, regex would have to be used to extract all the information.

2.3 NLTK

2.3.1 Sentence and Work Tokenizer

The sentence tokenizer splits a string up into smaller strings using particular punctuation as splitting points. This could be done using regex but the utility NLTK provides make it cleaner. The word tokenizer splits a string up using white space and certain punctuation, which again, can be done using regex, but the utility provided makes the task far simpler.

2.3.2 POS Tagger

The POS tagger allows a list of tokens to be converted into a list of tuples containing the tokens and its assigned POS tag. A POS tag is assigned using statistical data and positioning with a series of tokens to determine the correct assignment for its meaning. There are other taggers which NTLK provides but I find POS to be by far the best. It also doesn't require training, which can take a long time, and so it far more efficient.

2.3.3 WordNet Lemmatizer

This provides a lemmatizer and a vocabulary which can be used effectively if the POS tag is known for a given word. It removes the inflectional ending to leave words in their common form. It provides a very powerful tool, which allows for morphological analysis through simple commands.

2.3.4 Regex Parser

I used this along with a grammar to find noun phrases within POS tagged sentences. It, like regex, is a very powerful utility. The chunker produces a POS tree from the tagged sentences and the grammar. It is very efficient but requires the sentences to be tagged and processed correctly to produce the proper results.

3 Discussion of Solution

3.1 Implemented Functionality

3.1.1 Stemming vs. Lemmatization

Both methods aim to reduce inflectional forms and occasionally, derivationally related forms of a word to a base form. Stemming is however crude, using a heuristic process to remove endings from words with the aim of achieving the correct result, but tends to remove endings it shouldn't or miss's endings it should. Lemmatization takes context into account, using POS tags, a vocabulary and the morphological analysis of words, typically aiming to remove inflectional endings only. This is why I went with lemmatization over stemming. Even though it is a more expensive operation as it has to look up vocabulary.

3.1.2 POS Tagging Method

There are several methods which can be used to POS tag a sequence of sentences, most requiring a tagger to be trained using a corpora. This process is very slow and often requires a huge amount of information to produce effective results, although, NLTK does provide sizeable corpora and tag banks. I chose to use NLTKs POS tagger because it produces very accurate results using positional information by looking at its neighbouring tokens and statistical probability to determine the correct POS tag. It's not perfect but produces results very quickly and the quality of the results is often better than the other taggers NTLK provides support for in my experience.

3.1.3 Tokenizing Method

I chose NLTKs tokenizing tools over doing the regex myself because the results would have been the same in the end, so by using what's already available I had more time to focus on other parts of the processing pipeline. NLTKs tokenizers provide clean methods to perform the tasks as well.

3.1.4 Noun Phrase Selection

I initially attempted to do n-grams up to 5-grams on my data. However, this process both took a very long time and didn't produce good results. I believe this was because it didn't take into account what its neighbouring tokens actually meant. I then tried to POS grammar method known as chunking, which produced some really fantastic results and was much more efficient. I believe it could be made even better but that would require changes to the pre-processing; tokenizing and html parsing.

3.2 Potential Improvements and Extensions

3.2.1 HTML to Plaintext Parsing Improvements

I think I have a lot of room for improvement in this area. I spent a long time trying to get the output to be better and believe I made some good headroom in achieving that, however, I think a better way to go about it is to extract parts bit by bit, instead of doing it for a few items (meta, title and headers) then making the rest standard text.

Issues I had initially were due to the unusual HTML formatting of your site, for instance, the li tags close in one go, rather than at the end of the list item, so when you collect all data from a set of li tags, you get all the child li elements, resulting in a lot of duplicated data. Perhaps if I create an html tree and work my way up the nodes from the leaves I could produce some impressive parsed plaintext.

Another issue is that when you use BeautifulSoups' get text methods, it takes all the whitespace with it, producing some really bad results. For instance, some unrelated blocks of texts get merged and some related blocks of texts get separated. This results in issues for sentence splitting and the effect snowballs down the pipeline, getting worse as it gets further along. This might be fixed by removing all the structure done with whitespace within the html file, then, using the elements parse tree to produce text which is nested correctly and separating certain information.

Finally, I think I should get more information from tags to store, such as formatting like bold, italic and underline. This could be attained through looking at the CSS or in-line formatting. Information such as class names may also provide information in some cases. On the flip-side, some tags can be ignored for similar reason, like the contents of a navigation bar don't need to be looked at as they are very similar for all websites I.e. Home, products, contact us etc.

3.2.2 Lemmatization, Stemming and Morphological Analysis

I think I can improve these things a bit. Other than improvements which would be gained through better parsing and tokenization, I believe a combination of stemming and lemmatization may produce good results. I'd probably use the Porter Stemmer, although would like to do a comparison of a selection of stemming algorithms to find the best one.

3.2.3 Keyword and Key Phrase Selection Method

I think some better statistical selection methods are available than what I came up with, so I'd look into those and find how to select better results. The weightings could probably be manipulated to produce more desirable words and phrases, probably by gaining more information from earlier in the processing pipeline. Some sort of cross comparison between the word frequency and the word phrases could probably be done to produce a ranking for each phrase using the words it's comprised of.

3.2.4 Potential Extensions

I would like to perform the processing on a series of webpages, perhaps a set of Wikipedia pages, then, learn how to use the indexed terms in comparison with a given query to select the best documents. Developing a better understanding for how the whole model works.

4 Outputs

4.1 Output for Udo Kruschwutz Webpage

URL Link: http://csee.essex.ac.uk/staff/udo/index.html

Results Link:

The Udo_Output.txt

4.2 Output for BCS Karen Spärck Jones Award Webpage

URL Link: http://irsg.bcs.org/ksjaward.php

Results Link:

KSJ Award_Output.txt