NLP Report for Coursework 1

Taks1:

To get the input we use “PlaintextCorpusReader” from “nltk.corpus”, and we also need to download “nltk.download("punkt")” and “nltk.download("stopwords")” for tokenization and stopwords.

We first use "nltk.WordPunctTokenizer" to tokenize the raw input from the Simpsons (while testing I noticed this tokenizer would work the best as it separates words from their apostrophes and punctuation marks, but also based on whitespaces).

Lower all the characters after the tokenization, because we would certainly need case folding. For example for the word episode we have the variations:

episode - 1153
Episode - 188
episodes - 187
Episodes - 2

Thus we need to include all of them as one token.

Then from nltk.stem.wordnet we use WordNetLemmatizer to take care of the plurals of the words (Lemmatization), for cases seen as in the example above (episode-episodes).

I also tried stemming but I noticed that some tokens were affected by the stemming in a way not intended so I decided to not use it (‘krusty’ to ‘krusti’, ‘Marge’ into ‘marg’, ‘episode’ into ‘episod’, ‘character’ to ‘charact’).

Next get rid of the most common english stopwords "stopwords" from "nltk.corpus” and some manually entered ones, mostly punctuation marks(" ' ", ' , ' ,' " ', ' . ', …), so we are able to get rid of common tokens that would appear in most datasets.

The number of unique tokens found is 8246 and we need to select the top 100.

Finally we make a frequency distribution using "FreqDist" from nltk and select the "_.most_common(100)" to get the top 100 most frequent tokens.

Task2.1:

Task2:

Implementation:

For the Inverted Index I made:

- self.inverted_index_docfreq - a Frequency Distribution using FreqDist from nltk to hold the frequency of each term in all the documents
- self.inverted_index_postings - a nested dictionary to hold all the terms, and for each term the document ids with the number of appearances in each
- self.token_sequence - a list inside a dictionary which is arranged by document ids as keys and the text tokenized in the correct sequence in a list as the value of the dictionary. Used for the proximity search.
- self.documents - a dictionary to keep the documents raw data with the ids as keys

The way the Inverted Index class works is it first reads data from text files and sets an id based on the text file name. Then it also reads, saves and tokenizes the characters and locations names from the csv files.

Next you index all the documents, processing the documents one by one. This step is where you fill the Inverted Index (self.inverted_index_docfreq, self.inverted_index_postings and self.token_sequence).

The proces_document method in the index is the one responsible for pre-processing (case folding, stopwords removal, lemmatization), tokenization and multi-word expression tokenization for the characters and locations names.

A limitation of my implementation is that I do not use POS tagging and sentence tokenization, thus I am losing a lot of the context and meaning of words and phrases. For our task specifically, I do not find it as a very big inconvenience as we are doing a search engine, looking for specific terms, locations or characters.

Another problem is that I use a pretty strict punctuation-based tokenizer that separates whitespaces and punctuations, keeping the punctuation in separate tokens. This could prove to be a problem for context and sentence splitting and can have an impact on the quality of the text as a whole.

Decisions and features:

For reading the data I used PlaintextCorpusReader() from nltk.corpus and put the document ID and document text in a dictionary, with the key being the ID and the value the text.

After some testing and going through the raw input I noticed two problems:

1. The two different apostrophes: " ' " and " ’ "
2. The 3 types of hyphens: "-" "—" and "–" (10-year-old; and did not — do with her; April 20–26)

Those characters posed a few problems when tokenizing so I decided that I will replace all the " ’ " with the normal " ' " and "–" with “-” (because it kept dates as whole), directly in the raw text files but also in the stopword removal method. Regarding the "—", I removed it completely as I thought it brought no context or usefulness in my indexing.

I decided on using WordPunctTokenizer() from nltk to separate every word, in a single term first because my task was to be able to search through the tokens.

For example: “you’re” to 3 tokens (you, ‘, re), “Santa’s” to 3 tokens (Santa, ‘, s)

So after removing stop words I will be left with only non stopword tokens to look through. This has an impact on the speed of the search as there will be less tokens and on the recognition of characters and locations, as well as searching 2 terms in a proximity.

Basically the main reason for not choosing other tokenizers like TreebankWordTokenizer() or WhitespaceTokenizer was that I wanted to remove all the stop words that could prove a problem when indexing and searching single and multiword terms.

Because I wanted to efficiently search and identify the names in the simpsons_locations.csv and simpsons_characters.csv, I used MWETokenizer() (Multi-Word Expression), indexing multi-word terms based on the normalized_names from the csv files, so that I could go again through the tokens after the WordPunctTokenizer() and connect names into a single token; a multi-word term.

For example: “bart”, “simpson” would become “bart simpson”, “santa” “little” “helper” would become “santa little helper”, “flaming” “moe” would become “flaming moe”

One thing to note with the MWETokenizer() and stopwords removal is that even if there is a stopword in between a characters name, the stop word will be removed and it will be tokenized as a term (“ ‘Radio Bart’ The Simpsons ” will take “ bart simpson ”).

Here I have noticed a small inconsistency between simpsons_characters and simpsons_locations.

simpsons_characters.csv normalized_names are written a little differently than simpsons_locations.csv.

For example " 's " in “Bart’s Treehouse” and “Bart’s Friends”: in characters.csv, “Bart's Friends” is normalized as “barts friends”, whereas in locations.csv, “Bart's Treehouse” is normalized as “bart treehouse”, without the "s", which poses a problem when trying to index both with the same tokenization rules.

To fix this, make it consistent and searchable, because of the apostrophe tokenization, I had to remove all the “ ‘s “ from the normalized character names.

So for example, when using the demo function query you should write whatever you want to search without the “s“ if you don’t provide an apostrophe: “Santa’s Little Helper” or “santa little helper” instead of “santas little helper”, which makes the characters normalized names the same as the locations normalized names.

Regarding the pre-processing, I applied case folding first then stop-word removal. Because I used the WordPunctTokenizer() I was able to get rid of a large amount of them, making the process faster and more efficient.

The stop words I used were from the python library nltk.corpus and some manually entered ones. Apart from the usual “the”, “some”, “you”, “s”, “nt” …, I also added ‘.’, ‘,’, ‘;’ and some which I found, because of the text being downloaded from wikipedia, '←','→','•' which made sense removing, as they were not providing any information, and specifically in the characters and locations lists we get from our datasets, those are removed as well (‘Dr. Marvin Monroe’ to ‘dr marvin monroe’). This is a trade off though, as by using this amount of stopwords we are going to lose context in an amount of cases (‘13.9’ will be tokenized as ‘13’ and ‘9’; referencing in the text will lose context as well ‘[7][8]’ to ‘7’ and ‘8’; ‘11.1 million’ will become ‘11’ ‘1’ and ‘million’). But my decision of doing this was based on the fact that, doing a term search engine with characters and locations, it would help with the tokenization, greatly improving the Inverted Index performance, as most of the most frequent tokens without removing stop words were stop words (Example: “the” appears 6354 times).

An exception I did was when it came to “-”, for 2 reasons, one: because I thought that the hyphen in most cases kept the meaning of words (‘highest-rated’, ‘scratch-and-win’, ‘nineteenth-century’), and two: because it was used and kept for the normalized names in the locations and characters csv files (‘bob rv round-up). When removing the stopwords, if I find a hyphen then I take the word before it and the word after it and connect it all in a token (from the initial tokens ‘re’ ‘-’ ‘election’ I get ‘re-election’, same for ‘10-year-old’, etc.). Thus, the hyphen can be used in the demo function’s search query.

I wanted to keep a bit of more of the context and quality of tokens so I decided to only use lemmatization and not stemming, as stemming proves to sometimes stem too much (‘Marge’ into ‘marg’, ‘episode’ into ‘episod’) and could complicate the indexing.

It is true that with lemmatization you also lost a part of the context but I find it not as excessive as stemming. The only limitation with this would be if you were searching for an “un”-lemmatized term in the inverted index tokens, and the search would return not only your term, but also the lemmatized version.

It is also important to note that, because whenever a token is entered into the Inverted Index, it passes through the same tokenization methods, whether it is from the text files, the names of characters and locations or the input from the demo function query, any type of pre-processing would not pose a problem when searching (any input will be tokenized the same way; lemmatized etc.).

This makes it so that whatever is entered from the csv files or in the demo function query (with or without apostrophe, uppercase or lowercase, with or without stopwords), will have the highest probability of matching the tokens in the inverted index.

The size of the Inverted Index is significantly decreased because of the tokenizer I use (punctuation-based) and the stopwords removal. This increases performance, making the search engine faster, as there are a lot less tokens to look through.

Because I want the search to provide more results and include as many terms as possible, I apply lemmatization, which makes the tokens lose a bit of their quality in sentences.

What I can say about The Simpsons data set is that I found out how standard lacking, random and complicated can real world data prove to be and that we have to be careful when working with unannotated data, because various problems, like the inconsistencies apostrophes and hyphens stated above, can arise.

Task2.2:

In the demo function, after calling the Inverted Index class and entering the input query, I tokenize the query (both WordPunctTokenizer() + case folding, stopword removal, lemmatization and MWETokenizer()) so that it matches the tokens in the Inverted Index. Then I check if a character or location is in the search term/terms.

If only one term is entered or the terms are a character or location then I start looking at the Inverted Index Postings list and display all the info about the term.

If two terms are entered, even if they are or contain a character or location, I will do a proximity search:

For example for the queries:

1. “musical” - a single term query
2. “bart simpson” - it will do a character single term search but also a proximity search, but note that the proximity search is done for ‘bart’ and ‘simpsons’ separately, because ‘bart simpson’ will be tokenized as one because he is a character; so they are two different type of searches
3. “bart simpson blackboard” - it will do a proximity search for “bart simpson” and blackboard”
4. “Santa’s Little Helper” - will be normalized as “santa little helper”, marked as a character and the searched as a single term
5. “re-election” - normalized as “re-election” and searched as a single term
6. “stark raving” - will be recognised as two terms as they are not a location or a character and will only do a proximity search

For the proximity search I found 2 ways of doing it, but each one having their own limitations:

1. I could use n-grams, with n being the range (window) in which I want to search for a term in the proximity. After making the n-grams you just search in each engram if the term appears
   But this way I could not count how many times the two terms appear close to each other in a document, as the 2 terms could appear in 2 or more n-grams that are in the same place, making me count both, while there is only really one appearance of both.
2. I could use a list proximity search, where I look through all terms and when I find the first term, I look n steps to the left and n steps to the right and see if the second term is there, where n is how far I want to look to the left and right. But, in this case if the first term I’m looking for is the first in a text and the second one is the last, even if they are more than n steps apart from each other, they would be counted as proximal as when looking through a list of words, the item behind the first one is the last one in the list.

I decided on using the second method. The window size is how far to look to the left of a token and how far to look to the right of it (I set it at 3 -> it will look at 6 near terms).

This demo function was a necessity in testing the Inverted Index class. It proved to be very useful in supporting the task as it was a way for the user to interact with the Inverted Index and test the performance, possibilities and limitations of the search engine.

Fragments of the output of the program:

Task1:

The top 100 most frequent tokens we get by this process are [(token), (frequency in corpus)]:

```
[('episode', 1529), ('homer', 777), ('simpson', 575), ('bart', 483), ('show', 430), ('lisa', 381), ('reference', 328), ('marge', 302), ('season', 294), ('4', 277), ('1', 258), ('one', 257), ('2', 249), ('3', 247), ('5', 223), ('film', 207), ('scene', 192), ('6', 188), ('edit', 188), ('first', 168), ('song', 167), ('production', 165), ('guest', 158), ('fox', 155), ('written', 152), ('television', 147), ('writer', 147), ('jean', 141), ('plot', 140), ('family', 140), ('series', 136), ('7', 134), ('rating', 133), ('original', 129), ('get', 128), ('directed', 121), ('appearance', 121), ('make', 121), ('best', 120), ('also', 120), ('week', 120), ('said', 118), ('al', 117), ('character', 116), ('time', 115), ('krusty', 114), ('american', 109), ('reception', 109), ('gag', 106), ('named', 106), ('aired', 104), ('8', 103), ('parody', 103), ('cultural', 101), ('day', 101), ('reiss', 98), ('producer', 97), ('like', 96), ('list', 95), ('would', 94), ('originally', 94), ('12', 94), ('9', 93), ('two', 91), ('network', 90), ('feature', 89), ('star', 89), ('couch', 87), ('mike', 86), ('jump', 85), ('1992', 85), ('later', 83), ('burn', 83), ('matt groening', 82), ('13', 82), ('voice', 81), ('11', 79), ('tell', 79), ('book', 79), ('springfield', 78), ('next', 78), ('go', 78), ('received', 78), ('new', 78), ('14', 78), ('10', 76), ('called', 75), ('previous', 74), ('dvd', 73), ('home', 73), ('state', 72), ('air', 71), ('name', 71), ('play', 71), ('sequence', 71), ('guide', 70), ('jackson', 69), ('several', 69), ('end', 69), ('nielsen', 69)]
```

Task2:

```
--------------------------------------SINGLE TERM SEARCH-------------------------------------------------
Enter your query: musical
Tokenized for searching as: ['musical']
QUERY SEARCH: Appears as a term/token (musical) in:
11 episodes ['3.1', '3.8', '3.10', '3.13', '3.20', '3.22', '4.2', '4.12', '4.13', '4.20', '4.22']
1 times in episode: 3.1
1 times in episode: 3.8
1 times in episode: 3.10
1 times in episode: 3.13
2 times in episode: 3.20
1 times in episode: 3.22
16 times in episode: 4.2
2 times in episode: 4.12
1 times in episode: 4.13
1 times in episode: 4.20
3 times in episode: 4.22
30 times in total
------------------CHARACTER SEARCH + UNNORMALIZED QUERY SEARCH----------------------
Enter your query: Santa's Little Helper
Tokenized for searching as: ['santa little helper']
You entered a: Character
QUERY SEARCH: Appears as a term/token (santa little helper) in:
7 episodes ['3.2', '3.4', '3.11', '3.16', '3.19', '3.22', '4.5']
1 times in episode: 3.2
1 times in episode: 3.4
1 times in episode: 3.11
1 times in episode: 3.16
28 times in episode: 3.19
1 times in episode: 3.22
1 times in episode: 4.5
34 times in total
-------------------------------------------LOCATION SEARCH-------------------------------------------------
Enter your query: springfield elementary school
Tokenized for searching as: ['springfield elementary school']
```

```
You entered a: Location
QUERY SEARCH: Appears as a term/token (springfield elementary school) in:
7 episodes ['3.16', '3.18', '3.23', '4.4', '4.6', '4.7', '4.15']
1 times in episode: 3.16
1 times in episode: 3.18
1 times in episode: 3.23
1 times in episode: 4.4
1 times in episode: 4.6
1 times in episode: 4.7
1 times in episode: 4.15
7 times in total
---------------SEARCH WITH HYPHEN AND MULTIPLE WORDS AS ONE TERM-------------------
Enter your query: 10-year-old
Tokenized for searching as: ['10-year-old']
QUERY SEARCH: Appears as a term/token (10-year-old) in:
2 episodes ['3.20', '4.10']
1 times in episode: 3.20
1 times in episode: 4.10
2 times in total
----------------------------------------------PROXIMITY SEARCH---------------------------------------------
Enter your query: stark dad
Tokenized for searching as: ['stark', 'dad']
You entered a: Character
PROXIMITY: Appears as 2 terms (stark) and (dad) in:
6 episodes ['3.1', '3.2', '3.8', '3.19', '3.24', '4.6']
24 times in episode: 3.1
1 times in episode: 3.2
1 times in episode: 3.8
1 times in episode: 3.19
1 times in episode: 3.24
1 times in episode: 4.6
29 times in total
---------DEMO FUNCTION SEARCHING FOR CHARACTER BUT ALSO PROXIMITY------------
Enter your query: flaming moe
Tokenized for searching as: ['flaming moe']
You entered a: Location
QUERY SEARCH: Appears as a term/token (flaming moe) in:
3 episodes ['3.9', '3.10', '3.11']
1 times in episode: 3.9
19 times in episode: 3.10
1 times in episode: 3.11
21 times in total
PROXIMITY: Appears as 2 terms (flaming) and (moe) in:
1 episodes ['3.10']
3 times in episode: 3.10
3 times in total
-----------------------------TO NOTE: THOSE ARE 2 DIFFERENT SEARCHES----------------------------
---ONE FOR THE CHARACTER ‘FLAMING MOE’ AND ONE FOR ‘FLAMING’ AND ‘MOE’----
```

EXPLANATIONS: Enter your query: = input from user | Tokenized for searching as: = how it is searched in the inverted index | You entered a: Character/Location = if term(s) are a character or location | QUERY SEARCH: = single term search | PROXIMITY: = two terms search