CS6370: Natural Language Processing
Assignment 1 (Part-A)

Release Date: 20th Feb 2025 Deadline: 8th March 2025

Name:	Roll No.:
Anirudh Rao	**BE21B004**

General Instructions:

1. This assignment consists of two parts: A and B. Part B will be released later with a separate deadline.
2. The template for the code (in Python) is provided in a separate zip file. You are expected to fill in the template wherever instructed. Note that any Python library, such as nltk, stanfordcorenlp, spacy, etc, can be used.
3. The programming questions for the Spell Check and WordNet parts need to be done in separate Python files.
4. A folder named 'Roll_number.zip' that contains a zip of the code folder and your responses to the questions (a PDF of this document with the solutions written in the text boxes) must be uploaded on Moodle by the deadline.
5. You may discuss this assignment in a group, but the implementation must be completed individually and submitted separately.
6. Any submissions made after the deadline will not be graded.
7. Answer the theoretical questions concisely. All the codes should contain proper comments.
8. The institute's academic code of conduct will be strictly enforced.

The goal of this assignment is to build a search engine from scratch, which is an example of an Information Retrieval system. In the class, we have seen the various modules that serve as the building blocks for a search engine. We will be progressively building the same as the course progresses. This assignment requires you to build a basic text processing module that implements sentence segmentation, tokenization, stemming/lemmatization, spell check, and stopword removal. You will also explore some aspects of WordNet as a part of this assignment. The Cranfield dataset, which has been uploaded, will be used for this purpose.

Part 1: Sentence Segmentation [Theory + Implementation]

1. Suggest a simplistic top-down approach to sentence segmentation for English texts. Do you foresee issues with your proposed approach in specific situations? Provide supporting examples and possible strategies that can be adopted to handle these issues. [2 marks]

> - A simplistic top-down approach to sentence segmentation involves splitting text using punctuation markers like "`.`", "`!`", and "`?`".
> - Abbreviations like "Dr." can cause incorrect segmentation. To handle this, we can create a predefined list of common abbreviations to avoid incorrect splits.

2. Python NLTK is one of the most commonly used packages for Natural Language Processing. What does the Punkt Sentence Tokenizer in NLTK do differently from the simple top-down approach? [1 marks]

> The Punkt Sentence Tokenizer in NLTK differs from a simple top-down approach by using unsupervised machine learning to detect sentence boundaries, rather than relying on fixed rules like splitting by punctuation markers. It has been trained on a large corpus of text and has "learned" to identify sentence boundaries and distinguish between sentence-ending periods and periods in abbreviations.

3. Perform sentence segmentation on the documents in the Cranfield dataset using:
   a. The proposed top-down method and
   b. The pre-trained Punkt Tokenizer for English

   State a possible scenario where
   a. Your approach performs better than the Punkt Tokenizer
   b. Your approach performs worse than the Punkt Tokenizer [4 marks]

> The sentence segmentation was performed by implementing a Python code in the `sentenceSegmentation.py` file.

a. The naïve approach performs better than the Punkt Tokenizer when quotation marks are involved. For example, the sentence ‘"Is the cost of pie here 3.14?", she asked.’ is identified as a single sentence correctly by the proposed approach but incorrectly split by Punkt as ['"Is the cost of pie here 3.14?', '", she asked.'].
b. The Punkt Tokenizer works better in scenarios where the punctuation marks being used as delimiters in the naïve approach are present in non-standard forms, e.g. in ellipses (…). The sentence ‘He said, 'I don't know... Maybe later.' Then he left.’ is correctly split by Punkt as ["He said, 'I don't know... Maybe later.'", 'Then he left.'] but the top-down approach splits it incorrectly as ["He said, 'I don't know...", "Maybe later.' Then he left."].

Part 2: Tokenization [Theory + Implementation]

1. Suggest a simplistic top-down approach for tokenization in English text. Identify specific situations where your proposed approach may fail to produce expected results. [2 marks]

> A top-down approach to tokenization would be to split the text at whitespaces and use the split obtained as the tokens. This will not work as expected for contractions like "don't". This approach will consider this to be a single token but we would like it to be tokenized as "do" and "not".

2. Study about NLTK's Penn Treebank tokenizer. What type of knowledge does it use - Top-down or Bottom-up? [1 mark]

> The NLTK Penn Treebank Tokenizer employs a top-down approach for tokenization. This tokenizer utilizes a series of predefined regular expression rules to systematically split text into tokens.

3. Perform word tokenization of the sentence-segmented documents using
   a. The proposed top-down method and
   b. Penn Treebank Tokenizer

   State a possible scenario along with an example where:
   a. Your approach performs better than Penn Treebank Tokenizer
   b. Your approach performs worse than Penn Treebank Tokenizer

   [4 marks]

> a. The naïve approach outperforms the Penn Treebank Tokenizer in situations where we wish to preserve contractions. For example, the word "can't" is treated as a single token in the top-down approach by the Penn Treebank Tokenizer splits it as "ca" + "n't", which is undesirable.
> b. The naïve approach performs worse whenever punctuation marks are involved. For example, tokenizing "This book is good." gives ["This", "book", "is", "good", "."] from Penn Treebank, which is correct, but the top-down approach gives ["This", "book", "is", "good."].

Part 3: Stemming and Lemmatization [Theory + Implementation]

1. What is the difference between stemming and lemmatization? Give an example to illustrate your point. [1 marks]

> Stemming is a rule-based process that cuts off word suffixes without considering the actual meaning. It often produces non-real words. Lemmatization is dictionary-based and reduces words to their base (lemma) form, ensuring the result is a valid word.
>
> For example, consider the word "running". Stemming by removal of the verb suffix "-ing" gives "runn", a nonsense word. Lemmatization on the other hand reduces the word correctly to "run".

2. Using Porter's stemmer, perform stemming/lemmatization on the word-tokenized text from the Cranfield Dataset. [1 marks]

> NLTK's implementation of Porter's stemmer was used to perform stemming in the `inflectionReduction.py` file.

Part 4: Stopword Removal [Theory + Implementation]

1. Remove stopwords from the tokenized documents using a curated list, such as the list of stopwords from NLTK. [1 marks]

> NLTK's list of English stopwords was used to remove stopwords from the tokenized documents.

2. Can you suggest a bottom-up approach for creating a list of stopwords specific to the corpus of documents? [1 marks]

> A bottom-up approach for creating a corpus-specific stopword list involves analyzing the documents statistically by calculating word frequencies across the corpus and identify high-frequency words. These high-frequency words can be treated as stopwords. A threshold frequency can be set to determine which words are stopwords. For example, the mean μ and standard deviation σ of all word frequencies can be computed and the threshold can be taken as $\mu + \sigma$ for stopwords.

3. Implement the strategy proposed in the previous question and compare the stopwords obtained with those obtained from NLTK on the Cranfield dataset. [2 marks]

> Using the above approach yields 84 stopwords in the Cranfield dataset. These include scientific terms like 'effects', 'heat', 'body', 'experimental', 'boundary', along with common words like 'it', 'a', 'and', 'have', 'has', 'an', 'is', 'also', 'were'. The NLTK stopwords list has 198 words but these are not specific to the Cranfield dataset.

Part 5: Retrieval [Theory]

1. Given a set of queries Q and a corpus of documents D, what would be the number of computations involved in estimating the similarity of each query with every document? Assume you have access to the TF-IDF vectors of the queries and documents over the vocabulary V. [1 marks]

A single TF-IDF vector will be of length V, where V is the size of the vocabulary. If we are comparing similarity based on cosine similarity, we will have to compute the dot product of the query vector with a document vector, as well as compute the norms of the two vectors. The dot product computation will take $O(V)$ operations. The norm computation will also take $O(V)$ operations. The multiplication will take $O(1)$ operations. Comparing the computed cosine similarity to a threshold will also take $O(1)$ operations. To compare a single query against a single document, we take $O(V + V + 1 + 1) = O(V)$ operations.

Given a set of Q queries and D documents, the number of operations required would be $\boldsymbol{O(Q \cdot D \cdot V)}$.

2. Suggest how the idea of the inverted index can help reduce the time complexity of the approach in (2). You can introduce additional variables as needed. [3 marks]

An inverted index stores which documents contain which terms in the vocabulary. Instead of scanning all D documents, we only compare a query with relevant documents (those containing at least one query term).

Let us define

$\mathrm{V} = \{w_1, w_2, \dots, w_V\}$
$\mathrm{D} = \{d_1, d_2, \dots, d_D\}$
$\mathrm{Q} = \{q_1, q_2, \dots, q_Q\}$

An example of an inverted index would be

$\text{Index} = \{w_1 : \{d_3, d_7, d_{11}\}, w_2 : \{d_5\}, \dots, w_V : \{d_1, d_{D-2}\}\}$

Given a query with unique terms $\{t_1, \dots, t_{V'}\}$, we create a subset of D defined as

$$\mathrm{D}' = \bigcup_{i=1}^{n} \mathrm{Index}[t_i]$$

It can be expected that the size of D' is much smaller than that of the original set of documents D.

$$D' \ll D$$

We can then compute the cosine similarities on this smaller subset D'. Additionally, we can restrict the similarity computation using the TF-IDF entries corresponding to the V' unique terms in the query. It can be expected that this is smaller than the total number of words in the vocabulary.

$$V' \ll V$$

Thus, the required number of computations will be

$$O(Q \cdot D' \cdot V')$$

Because $D' \ll D$ and $V' \ll V$, we can say that

$$O(Q \cdot D' \cdot V') \ll O(Q \cdot D \cdot V)$$

Thus, inverted indexing helps reduce the computational complexity.

We have ignored the computations required to create the inverted index. It is assumed to have been done in a prior one-time pre-processing step.