CS6370: Natural Language Processing
Assignment 1

Release Date: 27th Feb 2024 Deadline: 11th March 2024

Name:	Roll No.:
Siddhagavali Shital Bhiku	ME20B166

General Instructions:

1. 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.
2. The programming questions for the Spell Check and WordNet parts need to be done in separate Python files.
3. 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.
4. Any submissions made after the deadline will not be graded.
5. Answer the theoretical questions concisely. All the codes should contain proper comments.
6. 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]

Simplistic Top-Down Approach to Sentence Segmentation:

A simplistic top-down approach to sentence segmentation for English texts could be to split the text into sentences based on punctuation marks that typically denote the end of a sentence, such as periods (.), exclamation points (!), and question marks (?).

Issues and Strategies:

1. **Abbreviations:** Texts often contain abbreviations that include periods, such as "Dr.", "Mr.", "U.S.A.", etc. Splitting sentences at every period would incorrectly segment these abbreviations.
   ***Strategy*:** Maintain a list of common abbreviations and check if the word before the period is in this list before splitting.
2. **Quotations**: Sentences may contain quoted text that includes sentence-ending punctuation.
   ***Strategy*:** Check if the punctuation mark is within quotation marks before splitting.
3. **Numerical Values**: Decimal numbers, dates, and other numerical values may contain periods.
   ***Strategy*:** Check if the characters surrounding the period are numerical before splitting.
4. **Ellipsis and Multiple Exclamations/Questions:** An ellipsis (…) or multiple exclamation/question marks (!!!/???) should not be treated as sentence boundaries.
   ***Strategy*:** Check for these patterns and treat them as a single unit.

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 NLTK Punkt Sentence Tokenizer uses a machine learning-based approach to identify sentence boundaries by learning from a large corpus of text. It adapts to diverse writing styles and contexts, providing better accuracy compared to a simplistic top-down rule-based approach, which relies on fixed punctuation rules and may struggle with contextual nuances.

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]

My approach performs better than Punkt Tokenizer:
Consider a scenario where the text contains unconventional usage of ellipses (...) to represent pauses or hesitations between sentences for example: "This is a sample document... it contains various examples... demonstrating different linguistic features."
Here my approach can handle the scenario much better by recognizing eclipses are sentence boundaries, which are based on simple punctuation rules.
My approach performs worse than Punkt Tokenizer:
Consider a scenario with a highly technical document that includes numerous abbreviations, acronyms, and specialized symbols, for example: "The IEEE 802.11 standard specifies WLAN communication using CSMA/CA, and its amendments (e.g., 802.11a, 802.11b) define various modulation schemes such as OFDM."
In this case, the Punkt Tokenizer, because Punkt is trained in the diverse text due to which it can capture complex patterns in the text, and handle the intricacies of technical language, abbreviations, and specialized symbols.

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 simplistic top-down approach for tokenization in English text could be splitting the text into sentences whenever a period (‘.’) is encountered, and then splitting each sentence into words whenever a space (’ '), new lines, tabs, hyphens, etc.. is encountered.

Failure Cases:

- **Hyphenated Words:** Words connected by hyphens (e.g., "well-known") may be incorrectly tokenized as separate words.
- **Abbreviations:** Abbreviations like "U.S.A." or "Dr." may be tokenized incorrectly.
- **Contractions:** The tokenizer would incorrectly split contractions. For example, “I’m happy” would be incorrectly split into “I’m”, “happy”
- **Punctuation:** The tokenizer would not separate punctuation from words. For example, “Hello, world!” would be incorrectly split into “Hello,”, “world!”.

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

NLTK's Penn Treebank tokenizer utilizes a bottom-up approach. It employs a set of rules and heuristics derived from the Penn Treebank corpus to recursively break down text into smaller units, such as words, punctuation, and symbols.

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]

My approach performs better than Penn Treebank Tokenizer:
In scenarios where sentences contain unconventional or non-standard linguistic constructs that the Penn Treebank Tokenizer struggles to handle, for example: "This is an example of unconventional linguistic constructs, e.g., sentences_with_underscores orWordsMergedTogether."
In this scenario, my rule-based approach could better adapt to non-standard word formations.

My approach performs worse than Penn Treenbank Tokenizer:
In situations where sentences involve complex linguistic structures, nested expressions, or irregular abbreviations that require a more sophisticated tokenizer. For example: "In a world filled with machine learning algorithms (e.g., deep neural networks), tokenization becomes a challenging task."
In this scenario, the Penn Treebank Tokenizer,is trained on a diverse range of texts or context, can better handle the complexities introduced by nested expressions and irregular abbreviations.

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 and lemmatization are both text normalization techniques, but they differ in their approaches. Stemming involves removing prefixes or suffixes from words to obtain their root form (stem), often resulting in non-linguistic but valid roots. Example: "running" → "run."
Lemmatization, on the other hand, transforms words into their base or dictionary form (lemma), considering linguistic rules. Example: "better" → "good."

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

```
from util import *

# Add your import statements here
from nltk.stem import PorterStemmer
from nltk.tokenize import word_tokenize

class InflectionReduction:

    def reduce(self, text):

        """
        Stemming/Lemmatization

        Parameters
        ----------
        arg1 : list
            A list of lists where each sub-list a sequence of
            tokens  representing a sentence
        Returns
        -------
```

```
        list
            A list of lists where each sub-list is a sequence of
            stemmed/lemmatized tokens representing a sentence
        """
        # Download the 'punkt' resource
        nltk.download('punkt')

        reducedText = []
        # Initialize the Porter Stemmer
        porter_stemmer = PorterStemmer()

        for sentence in text:

            # Perform stemming on each token
            stemmed_tokens =
                [porter_stemmer.stem(token) for token in sentence]
            # Add the stemmed tokens to the result
            reducedText.append(stemmed_tokens)

        return reducedText
```

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]

```
from util import *

# Add your import statements here
from nltk.corpus import stopwords

class StopwordRemoval:

    def fromList(self, text):
        """
        Remove stopwords from a list of tokenized sentences.

        Parameters
        ----------
        text : list
            A list of lists where each sub-list is a sequence of
            tokens representing a sentence.

        Returns
        -------
        list
            A list of lists where each sub-list is a sequence of
            tokens representing a sentence with stopwords removed.
        """

        nltk.download('stopwords')

        stopword_removed_text = []

        # Get the list of stopwords from NLTK
        stop_words = set(stopwords.words("english"))

        for sentence in text:
            # Remove stopwords from each sentence
```

```
            filtered_tokens =
  [token for token in sentence if token.lower() not in stop_words]

            # Add the filtered tokens to the result
            stopword_removed_text.append(filtered_tokens)

        return stopword_removed_text
```

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 list of stopwords specific to a corpus involves analyzing the term frequency and document frequency within the corpus. Words that are frequent across multiple documents and offer minimal discriminatory power may be considered as custom stopwords. This data-driven approach tailors the stopwords list to the characteristics of the specific corpus.

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]

```
import json
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from collections import Counter
nltk.download('punkt')
nltk.download('stopwords')

# Load the Cranfield dataset
with open('./cranfield/cran_docs.json', 'r') as file:
    cranfield_data = json.load(file)
    documents = [item["body"] for item in cranfield_data]

# Tokenize and aggregate word frequencies
```

```
word_counts = Counter()
for doc in documents:
    words = word_tokenize(doc.lower())  # Tokenize and lowercase
    words = [word for word in words if word.isalpha()]  # Keep
only alphabetic tokens
    word_counts.update(words)

# Identify the top N high-frequency words as potential stopwords
N = 500  # You might adjust this based on your analysis
top_n_words = [word for word, count in word_counts.most_common(N)]

# Compare with NLTK's stopwords
nltk_stopwords = set(stopwords.words('english'))
custom_stopwords = set(top_n_words)
common_stopwords = nltk_stopwords.intersection(custom_stopwords)
unique_to_cranfield = custom_stopwords.difference(nltk_stopwords)
unique_to_nltk = nltk_stopwords.difference(custom_stopwords)

print(f"Common stopwords: {sorted(common_stopwords)}")
print(f"Length of Common stopwords: {len(common_stopwords)}")
#print(f"Unique to Cranfield: {sorted(unique_to_cranfield)}")
#print(f"Unique to NLTK: {sorted(unique_to_nltk)}")
```

Output:

- **Common stopwords:** ['a', 'about', 'above', 'all', 'an', 'and', 'any', 'are', 'as', 'at', 'be', 'because', 'been', 'being', 'below', 'between', 'both', 'but', 'by', 'can', 'during', 'each', 'for', 'from', 'further', 'has', 'have', 'having', 'here', 'if', 'in', 'into', 'is', 'it', 'its', 'm', 'more', 'most', 'no', 'not', 'of', 'on', 'only', 'or', 'other', 'out', 'over', 'same', 'should', 'so', 'some', 'such', 'than', 'that', 'the', 'their', 'then', 'there', 'these', 'they', 'this', 'those', 'through', 'to', 'under', 'up', 'very', 'was', 'were', 'when', 'where', 'which', 'will', 'with']
- **Length of Common stopwords: 74**

For more results run: Please run **python customStopwordsRemoval.py**

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]

Estimating the similarity between each query in a set Q and every document in a corpus D, using TF-IDF vectors over a vocabulary V, involves calculating the dot product of these vectors. For each query-document pair, this computation includes V multiplications and (V-1) additions, where V is the vocabulary size.
Given Q queries and D documents, the total operations are:

- Multiplications: **(Q * D * V)**
- Additions: **(Q *D *(V-1))**

Total number of computations = Q * D * (2V - 1)
Thus, the total number of operations to estimate the similarity for all query-document pairs is a function of the number of queries, the number of documents, and the size of the vocabulary.

2. Suggest how the idea of the inverted index can help reduce the time complexity of the approach in (1). You can introduce additional variables as needed. [3 marks]

Using an inverted index, the final computation for estimating similarity between queries Q and documents D using TF-IDF vectors over a vocabulary V focuses on:

1. Q queries,
2. D_t, the average number of documents containing a given term,
3. T_q, the average number of unique terms in a query.

The total number of operations in this case will be = **(Q * D_t * T_q),** making the time complexity more manageable and dependent on the specific characteristics of the queries and the distribution of terms across documents, rather than the total number of documents or the size of the vocabulary.

Part 6: Spell Check [Theory + Implementation]

1. Construct a vocabulary V of all the types (unique tokens) from the Cranfield dataset. You may additionally filter out alpha-numeric types. Represent each type in V as a vector in a vector space spanned by all possible bigrams of the English alphabet (‘aa,’ ‘ab,’ ‘ac,’… ‘zz’). Given the typos - ‘boundery’, ‘transiant’, ‘aerplain’ - find the top 5 candidate corrections corresponding to each. [5 marks]

```
class SpellCheck:
    def __init__(self, json_file_path):
        self.vocabulary_vectors =
self.load_and_process_data(json_file_path)

    def load_and_process_data(self, json_file_path):
        # Load JSON data
        with open(json_file_path, 'r') as file:
            cranfield_data = json.load(file)

        # Concatenate all bodies to form the corpus
        corpus = ' '.join([doc['body'] for doc in cranfield_data])

        # Tokenize and clean the corpus
        tokens = word_tokenize(corpus.lower())
        tokens = [token for token in tokens if token.isalpha() and
token not in stopwords.words('english')]

        # Construct the vocabulary and represent tokens as vectors
        unique_tokens = set(tokens)
        vocabulary_vectors = {token: self.token_to_vector(token)
for token in unique_tokens}
        return vocabulary_vectors


    def token_to_vector(self, token):
        alphabet = 'abcdefghijklmnopqrstuvwxyz'
        bigram_space = [a + b for a in alphabet for b in alphabet]
        bigram_index = {bigram: idx for idx, bigram in
```

```
enumerate(bigram_space)}

        token_bigrams = [''.join(bg) for bg in bigrams(token)]
        vector = np.zeros(len(bigram_space))
        for bg in token_bigrams:
            if bg in bigram_index:
                vector[bigram_index[bg]] += 1
        return vector

    def find_corrections(self, typo):
        typo_vector = self.token_to_vector(typo)
        similarities = []
        for token, vector in self.vocabulary_vectors.items():
            sim = cosine_similarity([typo_vector], [vector])[0][0]
            similarities.append((token, sim))
        similarities.sort(key=lambda x: x[1], reverse=True)
        return [(token, sim) for token, sim in similarities[:5]]

spell_check = SpellCheck('./cranfield/cran_docs.json')
corrections_for_boundery =
spell_check.find_corrections('boundery')
corrections_for_transiant =
spell_check.find_corrections('transiant')
corrections_for_aerplain =
spell_check.find_corrections('aerplain')

print('Corrections for "boundery":', corrections_for_boundery)
print('Corrections for "transiant":', corrections_for_transiant)
print('Corrections for "aerplain":', corrections_for_aerplain)
```

- **Corrections for "boundery":** ['bounded', 'bound', 'unbounded', 'boundary', 'bounds']
- **Corrections for "transiant":** ['transient', 'transit', 'transcendant', 'radiant', 'transition']
- **Corrections for "aerplain":** ['plain', 'explain', 'airplane', 'explains', 'explaining']

2. Write a function in Python to compute the Edit Distance between two input strings. For each typo listed above, find the candidate among the top 5 closest to the typo using the Edit Distance function. Assume the cost of insertion, deletion, and substitution to be equal to 1. [4 marks]

```
## Edit distance for spell correction
def edit_distance(str1, str2):
    """
    Compute the edit distance between two input strings.

    Args:
    str1 (str): The first input string.
    str2 (str): The second input string.

    Returns:
    int: The edit distance between the two strings.
    """
   m, n = len(str1), len(str2)

    # Initialize a matrix to store the distances
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Base case initialization
    for i in range(m + 1):
        dp[i][0] = i
    for j in range(n + 1):
        dp[0][j] = j

    # Compute the distances
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            cost = 0 if str1[i - 1] == str2[j - 1] else 1
            dp[i][j] = min(dp[i - 1][j] + 1,         # Deletion
                           dp[i][j - 1] + 1,         # Insertion
                           dp[i - 1][j - 1] + cost) # Substitution

    return dp[m][n]
```

```
# Function to find the closest candidate for each typo
def find_closest_candidate(typo, candidates):
    min_distance = float('inf')
    closest_candidate = None
    for candidate in candidates:
        distance = edit_distance(typo, candidate)
        if distance < min_distance:
            min_distance = distance
            closest_candidate = candidate
    return closest_candidate, min_distance

candidates = {
    'boundery': [correction for correction, score in
corrections_for_boundery],
    'transiant': [correction for correction, score in
corrections_for_transiant],
    'aerplain': [correction for correction, score in
corrections_for_aerplain]
}

# Finding the closest candidate for each typo
closest_candidates = {typo: find_closest_candidate(typo,
candidates[typo]) for typo in candidates}

for typo, closest in closest_candidates.items():
    print(f'The closest candidate for "{typo}" is "{closest[0]}"
with disatance "{closest[1]}".')
```

Output:

The closest candidate for "boundery" is "boundary" with distance "1".
The closest candidate for "transiant" is "transient" with distance "1".
The closest candidate for "aerplain" is "explain" with distance "2".

3. Experiment with different costs of insertion, deletion, and substitution (note that all three need not be the same), and identify necessary conditions under which Edit Distance is a valid distance measure. [2 marks]

To maintain the Edit Distance as a valid measure under different costs, the following conditions should be observed:

- Insertion: Cost > 0
- Deletion: Cost > 0
- Substitution: Cost > 0

1. **All costs must be positive:** Negative costs could violate the non-negativity and triangle inequality properties.
2. **The cost of substitution should not be less than the cost of insertion or deletion:** If substitution were cheaper than the sum of an insertion and a deletion, it could violate the triangle inequality.

I have conducted experiments by adhering to the rules mentioned above, and the output remained unchanged. However, when deviating from these rules, such as increasing the cost of deletion and insertion compared to substitution, the outcomes obtained it is causing accuracy issues here I mean that achieved results may not be desirable and the algorithm can become less efficient.

Part 7: WordNet [Theory + Implementation]

From the NLTK library, use the WordNet interface for the following tasks:

1. Print the list of all synsets corresponding to the words ‘progress’ and ‘advance.’ [1 marks]

```
import nltk
from nltk.corpus import wordnet
nltk.download('wordnet')
def get_synsets(word):

    synsets = wordnet.synsets(word)
    return synsets

words = ['progress', 'advance']

for word in words:
    synsets = get_synsets(word)
    print(f"Synsets for '{word}':")

    for synset in synsets:
        print(f"  - {synset.name()} ")

    print("\n")
```

Synsets for 'progress':
- advancement.n.03

- progress.n.02
- progress.n.03
- progress.v.01
- advance.v.01
- build_up.v.02

Synsets for 'advance':
- progress.n.03
- improvement.n.01

```
 - overture.n.03
 - progress.n.02
 - advance.n.05
 - advance.n.06
 - advance.v.01
 - advance.v.02
 - boost.v.04
 - promote.v.01
 - advance.v.05
 - gain.v.05
 - progress.v.01
 - advance.v.08
 - promote.v.02
 - advance.v.10
 - advance.v.11
 - advance.v.12
 - advance.s.01
 - advance.s.02
```

2. Print the definitions corresponding to the synsets obtained in the previous question. [1 marks]

```
Definitions for 'progress':
 - advancement.n.03 - gradual improvement or growth or development
 - progress.n.02 - the act of moving forward (as toward a goal)
 - progress.n.03 - a movement forward
 - progress.v.01 - develop in a positive way
 - advance.v.01 - move forward, also in the metaphorical sense
 - build_up.v.02 - form or accumulate steadily


Definitions for 'advance':
 - progress.n.03 - a movement forward
 - improvement.n.01 - a change for the better; progress in development
 - overture.n.03 - a tentative suggestion designed to elicit the reactions of
others
 - progress.n.02 - the act of moving forward (as toward a goal)
 - advance.n.05 - an amount paid before it is earned
```

- advance.n.06 - increase in price or value
- advance.v.01 - move forward, also in the metaphorical sense
- advance.v.02 - bring forward for consideration or acceptance
- boost.v.04 - increase or raise
- promote.v.01 - contribute to the progress or growth of
- advance.v.05 - cause to move forward
- gain.v.05 - obtain advantages, such as points, etc.
- progress.v.01 - develop in a positive way
- advance.v.08 - develop further
- promote.v.02 - give a promotion to or assign to a higher position
- advance.v.10 - pay in advance
- advance.v.11 - move forward
- advance.v.12 - rise in rate or price
- advance.s.01 - being ahead of time or need
- advance.s.02 - situated ahead or going before

3. Estimate the path-based similarity between the words 'advance' and 'progress' using the similarities between their synsets. [2 marks]

```
## Path simalirity between words
from nltk.corpus import wordnet

# Load synsets for both words
advance_synsets = wordnet.synsets('advance')
progress_synsets = wordnet.synsets('progress')

# Function to calculate the path-based similarity between synsets
of two words
def calculate_similarity(synsets1, synsets2):
    max_similarity = 0
    for syn1 in synsets1:
        for syn2 in synsets2:
            similarity = syn1.path_similarity(syn2)
            if similarity is not None and similarity >
max_similarity:
                max_similarity = similarity
    return max_similarity
```

```
# Calculate and display the maximum path-based similarity between
'advance' and 'progress'
max_similarity = calculate_similarity(advance_synsets,
progress_synsets)
max_similarity
```

Output: 1.0

4. Considering that the number of synsets of the words ‘advance’ and ‘progress’ are ‘m’ and ‘n,’ respectively, what is the number of calls made to the inbuilt path-based similarity function while computing the similarity between the two words? [1 marks]

When calculating the similarity between two words, we compare each synset of the first word with each synset of the second word to find the maximum similarity score. If the first word has m synsets and the second word has n synsets, you will make **(m * n**) calls to the path-based similarity function.