Department Of Aerospace Engineering
Indian Institute Of Technology Madras

CS6370-Natural Language Processing
Prof. Sutanu Chakraborti

Assignment - 1

Sanket Pramod Bhure (*AE20B108*)

March 11, 2024

Contents

List of Figures

1 Sentence Segmentation

1.1 Top-down approach to sentence segmentation

1.1 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.
Ans:
In English, **end marks**, including the **period**, **question mark**, and **exclamation mark**, signal the completion of a sentence. The simplest top-down approach to sentence segmentation involves separating text based on these end marks, such as {".", "!", "?"}.
Above approach is straightforward, but has many inherent **limitations**. The end-mark separation technique, primarily relying on punctuation like periods, encounters issues with correct sentence segmentation. For instance, the dot symbol used in abbreviations may trigger the segmentation process, resulting in fragmented sentences and rendering the extracted parts meaningless.
For example, the sentence **"I am currently pursuing my B.Tech in Aerospace Engineering."** is separated as:

- **Sentence 1:** I am currently pursuing my B.
- **Sentence 2:** Tech in Aerospace Engineering.

As sentences may get arbitrarily split near the abbreviations, the context of a long sentence may get lost entirely. English texts commonly feature abbreviations, short forms, and numbers with decimals, all using periods, leading the top-down approach to inaccurately separate them into distinct sentences.
To address limitations, we can expand the top-down approach with rules like detecting a space followed by a **capital** letter after a sentence's end. However, adding more rules reveals counter-examples where the defined rules fail. This is where a **bottom-up** approach might become useful where the system learns to segment sentences by itself, learning from relevant **data** in a particular domain.

1.2 Punkt Sentence Tokenizer in NLTK

1.2 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?
Ans:
In contrast to the top-down end-mark separation technique, the Punkt Sentence Tokenizer in NLTK adopts a **bottom-up** approach. It uses an **unsupervised** algorithm to construct a model for **abbreviations**, **collocations**, and sentence-starting words, employing this model to identify sentence boundaries. The tokenizer can be pre-trained or dynamically trained on a given dataset, providing flexibility for sentence segmentation. Punkt Sentence Tokenizer achieves this by two fold process:

- **Type-based classifier:** It initially distinguishes abbreviation(or initials/ordinal numbers) from normal words.
- **Token-based classifier:** It is more of refining stage (for example, abreviations which occurs at end of sentences are correctly identified).

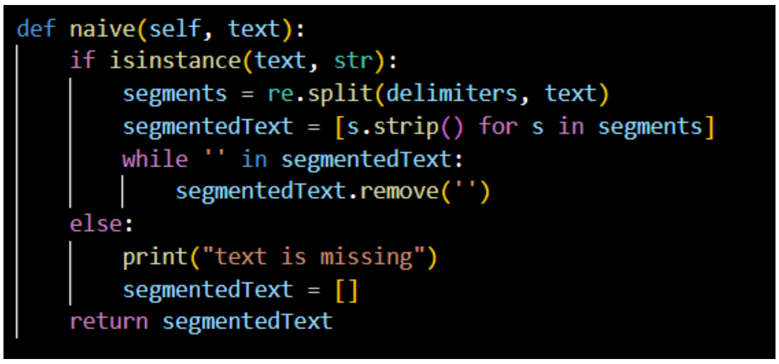
Sentence: **"Hi, my name is Sanky, I'm taking the NLP course offered by Prof. Sutanu Chakraborti."**

Punkt Sentence Tokenizer tokenizer detects that period in “Prof. Sutanu Chakraborti” do not mark sentence boundary. It tokenizes the sentence into a single sentence.

1.3 Sentence Segmentation on Cranifield Dataset

1.3 Perform sentence segmentation on the documents in the Cranfield dataset using:
Ans:

- **The proposed top-down method:** The naive function takes list as a input and performs the following steps:
  - It checks if the input is a list using the isinstance function.
  - If the input is a list, it splits the text into segments using delimiters(mentioned in our approach). The segments are stored in the segments variable.
  - Each segment is stripped of leading and trailing whitespace using the strip function, and the resulting segments are stored in the segmentedText list.
  - It then removes any empty segments from the segmentedText list using a while loop and the remove function.
  - If the text input is not a string, it prints a message stating that the text is missing, and assigns an empty list to segmentedText.
  - Finally, it returns the segmentedText list.



```python
def naive(self, text):
    if isinstance(text, str):
        segments = re.split(delimiters, text)
        segmentedText = [s.strip() for s in segments]
        while '' in segmentedText:
            segmentedText.remove('')
    else:
        print("text is missing")
        segmentedText = []
    return segmentedText
```

Figure 1: Sentence Segmentation using naive approach

- **The pre-trained Punkt Tokenizer for English:** The punkt function takes a text input and performs the following steps:
  - It checks if the text input is a string using the isinstance function.
  - If the text is a string, it initializes a PunktSentenceTokenizer tokenizer object with the text.
  - It then uses the tokenize method of the tokenizer to segment the text into sentences. The resulting segments are stored in the segmentedText variable.
  - The segmentedText is returned as the output of the function.
  - If the text input is not a string, it prints a message stating that the text is missing.Finally, it assigns an empty list to segmentedText and returns it.

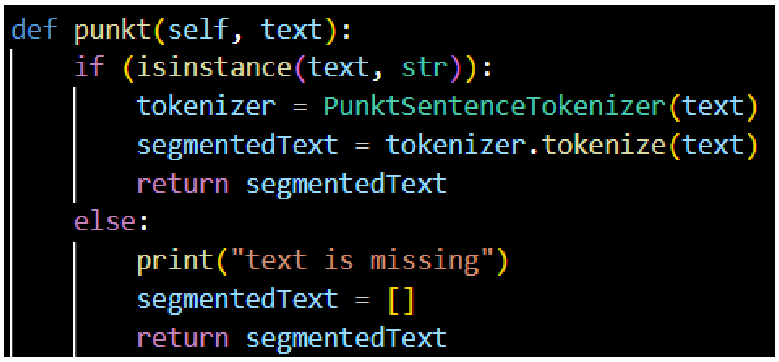

```python
def punkt(self, text):
    if (isinstance(text, str)):
        tokenizer = PunktSentenceTokenizer(text)
        segmentedText = tokenizer.tokenize(text)
        return segmentedText
    else:
        print("text is missing")
        segmentedText = []
        return segmentedText
```

Figure 2: Sentence Segmentation using Punkt Sentence Tokenizer

1.3 State a possible scenario where:
<u>Ans:</u>

- **Your approach performs better than the Punkt Tokenizer:**

  There are times when our top down approach performs better than Punkt Sentence Tokenizer, as the Texts cannot always be perfect there might be some spacing error like for example text: **'I know you.We studied in same college'**

  - **Top-down approach:** ['I know you.', 'We studied in same college']
  - **Punkt:** ['I know you.We studied in same college']

  where punkt failed to catch the spacing error

- **Your approach performs worse than the Punkt Tokenizer:**

  Here as well there are many cases where second method outperforms the first in case of all abbreviations for example text: **'I scored 12.75 marks in mid-sem.'**

  - **Top-down approach:** ['I scored 12.', '75 marks in mid-sem.']
  - **Punkt:** ['I scored 12.75 marks in mid-sem.']

  where it wrongly split by our top-down approach.

2 Tokenization

2.1 Tokenization in English text

2.1 Suggest a simplistic top-down approach for tokenization in English text. Identify specific situations where your proposed approach may fail to produce expected results
<u>Ans:</u>
The most straightforward and apparent **top-down** approach to word tokenization is the use of spaces and symbols like **hyphens**(-), **commas**(,), **forward-slashes**(/), etc. to locate word endings. . However this is naive approach and For example, let the sentence to be tokenized be **"I can't do this"**.

- **Tokenize:** ["I", "can", "t", "do", "this"]

The naive approach tokenizes "can't" into 'can' and 't', leading to the loss of information about negation (i.e., "cannot"). It may struggle with more complex compound words or expressions, leading to inaccurate tokenization and potential loss of **semantic** context.

2.2 NLTK's Penn Treebank Tokenizer

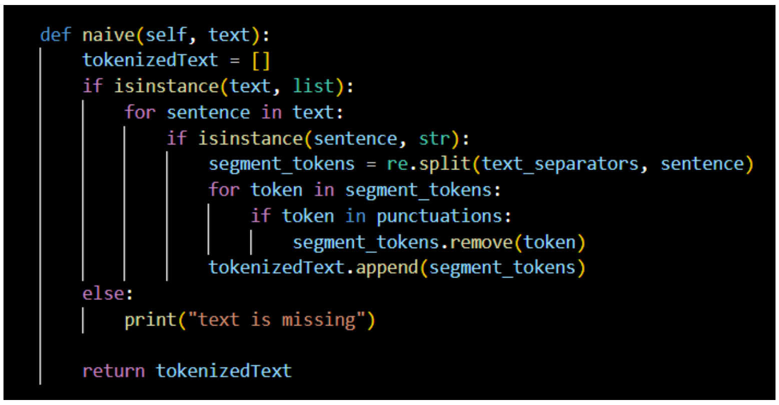
2.2 Study about NLTK's Penn Treebank tokenizer. What type of knowledge does it use - Top-down or Bottom-up?
<u>Ans:</u>
The Penn Treebank tokenizer is a **top-down** approach for word tokenization which uses **regular expressions** as defined in Penn Treebank to tokenize text. It uses knowledge based on expressions which occur regularly in English texts. This tokenizer assumes that text has already been segmented into sentences.
The Penn Treebank tokenizer performs the following four operations:

- Spliting of standard contractions such as "couldn't" to "could and n't", "I'll" to "I and 'll".
- Treating most punctuation characters as separate tokens.
- Spliting commas and single quotes when followed by whitespace.
- Separating periods which appear at the end of the line.

2.3 Word Tokenizer

2.3 Perform word tokenization of the sentence-segmented documents using:
<u>Ans:</u>

- **The proposed top-down method:** The naive function takes a text input and performs the following steps:
  - It takes a list of strings, where each string represents a single sentence.
  - Then, we check for an input.
  - The method then splits each sentence into tokens using regular expressions (**text_separators** = [' ,-/]) and removes **punctuations** (a list defined in util.py) tokens.
  - The resulting tokenized text is returned as a list of lists, where each sub-list represents a sequence of tokens
  - Additionally, the code includes handling cases where the input is not provided.

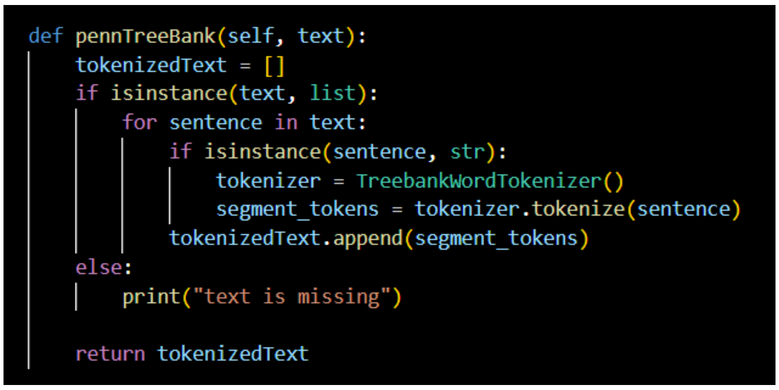


```python
def naive(self, text):
    tokenizedText = []
    if isinstance(text, list):
        for sentence in text:
            if isinstance(sentence, str):
                segment_tokens = re.split(text_separators, sentence)
                for token in segment_tokens:
                    if token in punctuations:
                        segment_tokens.remove(token)
                tokenizedText.append(segment_tokens)
    else:
        print("text is missing")

    return tokenizedText
```

Figure 3: Word tokenization using naive approach

- **Penn Treebank Tokenizer:** The punkt function takes a text input and performs the following steps:

- Here, we perform tokenization using the Penn Tree Bank tokenizer of the nltk.tokenize library
- Firstly, we check for an input.
- Then for every sentence, a tokenizer of nltk package is used to tokenize the sentence.



```
def pennTreeBank(self, text):
    tokenizedText = []
    if isinstance(text, list):
        for sentence in text:
            if isinstance(sentence, str):
                tokenizer = TreebankWordTokenizer()
                segment_tokens = tokenizer.tokenize(sentence)
            tokenizedText.append(segment_tokens)
    else:
        print("text is missing")

    return tokenizedText
```

Figure 4: Word tokenization using Penn Treebank Tokenizer

2.3 State a possible scenario where:

Ans:

- **Your approach performs better than Penn Treebank Tokenizer:**

  Our naive approach outperforms the Penn Treebank method, especially in **handling hyphens (-)**. The Penn Treebank ignores hyphens altogether. For example, let the sentence to be tokenized be **”The well-known actor starred in a record-breaking, award-winning performance.”**. The tokenized words produced by the Penn Treebank approach and our naive approach are as follows:

  - **Naive approach:** [’The’, ’well’, ’known’, ’actor’, ’starred’, ’in’, ’a’, ’record-breaking’, ’award-winning’, ’performance’]
  - **Penn Treebank Tokenizer:** [’The’, ’well-known’, ’actor’, ’starred’, ’in’, ’a’, ’record’, ’breaking’, ’award’, ’winning’, ’performance’]

  where Penn Treebank Tokenizer ignores the hyphens and failed to tokenize it independently.

- **Your approach performs worse than Penn Treebank Tokenizer:**

  The Penn Treebank Tokenizer is better at handling **standard contractions**. For example, let the sentence to be tokenized be: **”I can’t believe we’re already halfway through the year.”**

  - **Naive approach:** [’I’, ’can’, ’t’, ’believe’, ’we’, ’re’, ’already’, ’halfway’, ’through’, ’the’, ’year’]
  - **Penn Treebank Tokenizer:** [’I’, ’ca’, ’n’t’, ’believe’, ’we’, ’re’, ’already’, ’halfway’, ’through’, ’the’, ’year’]

  The naive approach tokenizes ”can’t” into ’can’ and ’t’, leading to the loss of information about **negation** (i.e., ”cannot”).

3 Stemming and Lemmatization

3.1 Stemming and Lemmatization

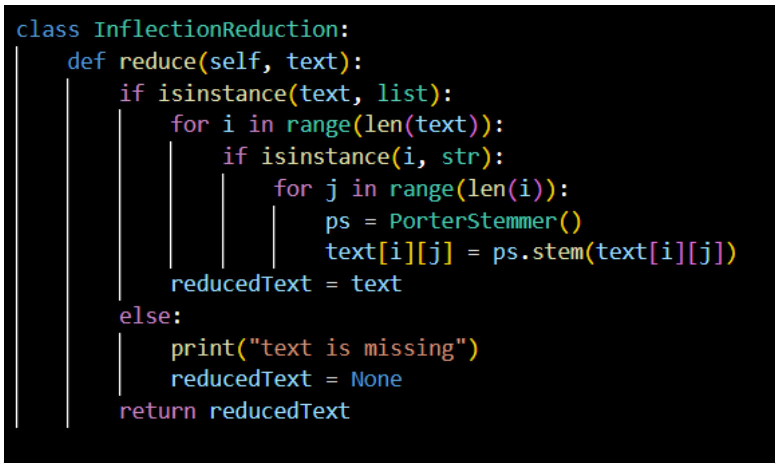
3.1 What is the difference between stemming and lemmatization? Give an example to illustrate your point ?
Ans:
Both stemming and lemmatization are text normalization techniques, used for reducing inflectional forms and derivationally related words to common base word form. Stemming chops off affixes from word to get the base word/stem. Stem may not be a meaningful word. In contrast, Lemmatization reduces words using full morphological analysis to identify the lemma, a meaningful word.

- **Stemming:** E.g., cunning $\rightarrow$ cun.
- **lemmatisation:** E.g., cunning $\rightarrow$ cunning.

Stemming is used when the dataset is large but may give incorrect meanings and spellings. Lemmatization can overcome this but it can be computationally expensive as it needs to scan the entire corpus.

3.2 Porter's Stemmer Algorithm

3.2 Using Porter's stemmer, perform stemming/lemmatization on the word-tokenized text from the Cranfield Dataset.
Ans:
In this process, we conduct lemmatization on tokenized words utilizing the NLTK library. Initially, we verify the presence of input. Subsequently, for each token within a sentence of a given text, we perform lemmatization on the word, aiming to reduce it to its base or root form.



```
class InflectionReduction:
    def reduce(self, text):
        if isinstance(text, list):
            for i in range(len(text)):
                if isinstance(i, str):
                    for j in range(len(i)):
                        ps = PorterStemmer()
                        text[i][j] = ps.stem(text[i][j])
            reducedText = text
        else:
            print("text is missing")
            reducedText = None
        return reducedText
```

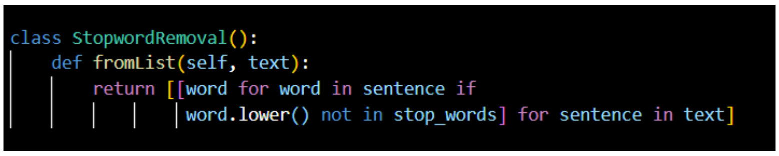
Figure 5: Stemming/lemmatization using Porter's Stemmer Algorithm

4 Stopword Removal

4.1 Stopword Removal using curated list

4.1 Remove stopwords from the tokenized documents using a curated list, such as the list of stopwords from NLTK.

Ans:
We remove the stopwords from the lemmatized set of words using the nltk library. Only those words not in stop words are considered. Stop words are taken from the ntlk library of python.



```
class StopwordRemoval():
    def fromList(self, text):
        return [[word for word in sentence if
                word.lower() not in stop_words] for sentence in text]
```

Figure 6: Stopword Removal using curated list

4.2 Bottom-up Approach

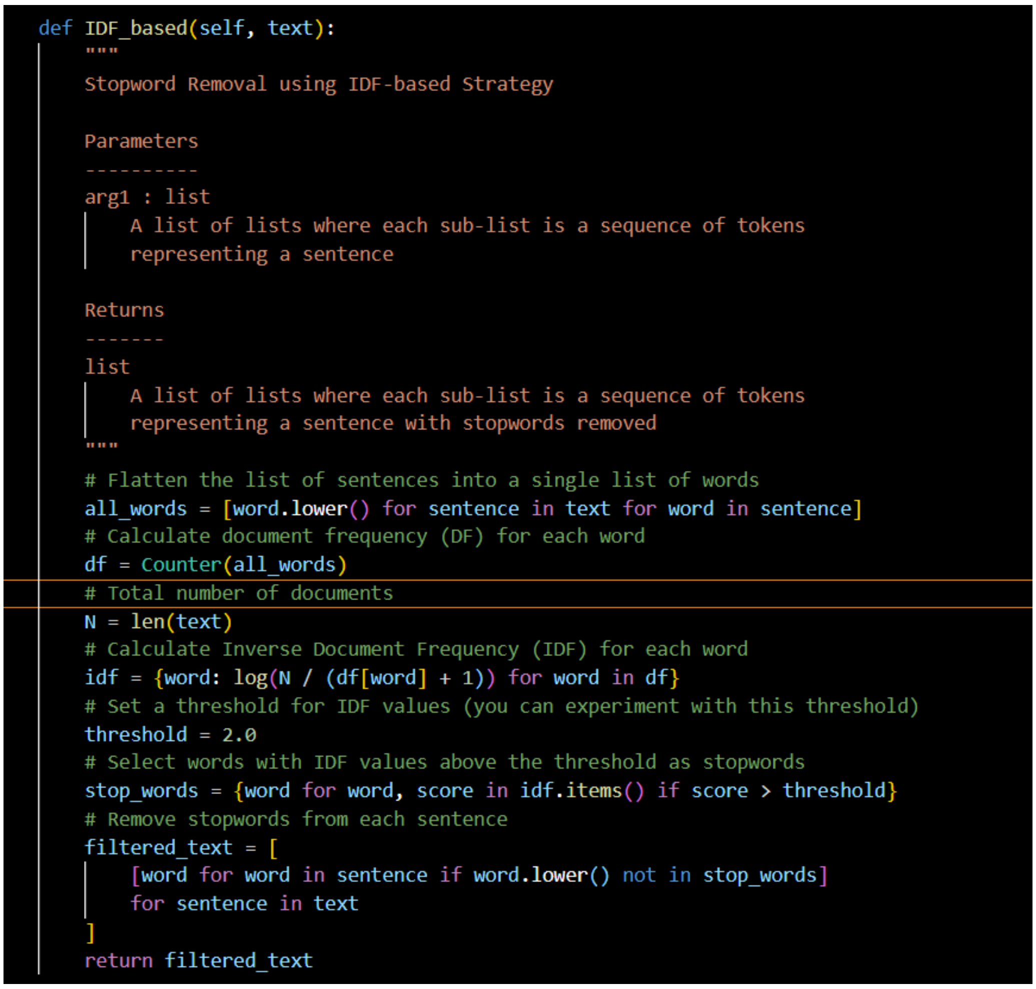
4.2 Can you suggest a bottom-up approach for creating a list of stopwords specific to the corpus of documents?
Ans:
The most intuitive bottom-up approach for creating a stopword removal involves sorting all words in the corpus by their **collection frequency**. The idea is to select the most frequent words as stop words.
A more involved way of doing this is with the help of **Inverse document frequency (IDF)**. IDF gives us measure of whether a term is common or rare in given document corpus. We can sort the IDF values and eliminate those words which fall below the threshold value, which we can experiment on. IDF is calculated as follows:

$$IDF(x) = \log\left(\frac{N}{n+1}\right)$$

Where N is the total number of documents, and n is the number of documents where term x appears. Sorting IDF values helps identify and remove frequently occurring words with low IDF.

4.3 Stopword Removal using bottom-up

4.3 Implement the strategy proposed in the previous question and compare the stopwords obtained with those obtained from NLTK on the Cranfield dataset.
Ans:
NLTK provides a predefined list of common stopwords, including frequently occurring words like "the," "and," and "is." These stopwords are commonly removed in text analysis tasks to focus on meaningful content.
After applying the **IDF-based** strategy and NLTK stopwords removal, a comparison is made. Words identified as stopwords by the IDF-based strategy but not present in the NLTK stopwords list are considered extra stopwords. Example extra stopwords include **'results,'** **'theory,'** **'medium,'** **'boundary,'** **'body,'** etc. Below is the implementation of the code of above strategy.



```python
def IDF_based(self, text):
    """
    Stopword Removal using IDF-based Strategy

    Parameters
    ----------
    arg1 : 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
    """
    # Flatten the list of sentences into a single list of words
    all_words = [word.lower() for sentence in text for word in sentence]
    # Calculate document frequency (DF) for each word
    df = Counter(all_words)
    # Total number of documents
    N = len(text)
    # Calculate Inverse Document Frequency (IDF) for each word
    idf = {word: log(N / (df[word] + 1)) for word in df}
    # Set a threshold for IDF values (you can experiment with this threshold)
    threshold = 2.0
    # Select words with IDF values above the threshold as stopwords
    stop_words = {word for word, score in idf.items() if score > threshold}
    # Remove stopwords from each sentence
    filtered_text = [
        [word for word in sentence if word.lower() not in stop_words]
        for sentence in text
    ]
    return filtered_text
```

Figure 7: Stopword Removal using IDF based strategy

5 Retrieval

5.1 Without Inverted Index

5.1 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.

Ans:

Given m **queries** in set Q and n **documents** in corpus D, each represented by TF-IDF vectors over vocabulary V, the number of computations involved in estimating the similarity of each query with every document is:

$$\text{Total number of Computations} = m \times n \times O(V)$$

Here, we need to calculate the **cosine similarity** for each query-document pair using the TF-IDF vectors (Dimension V):

$$\text{Cosine Similarity}(Q_i, D_j) = \frac{Q_i \cdot D_j}{\|Q_i\| \cdot \|D_j\|}$$

where Q_i represents the TF-IDF vector for query i and D_j represents the TF-IDF vector for document j and $O(V)$ represents the complexity of computing the dot product between two

vectors of length V, which is typically linear in the size of the vocabulary.
So, the total number of computations is the product of the number of queries (m), the number of documents (n), and the complexity of computing the cosine similarity between two TF-IDF vectors of length V, which is O(V).

5.2 With Inverted Index

5.2 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.
Ans:
In an inverted index, instead of storing the TF-IDF vectors for each document, we store a mapping of terms to the documents that contain those terms, along with the corresponding TF-IDF values. Let's introduce some additional variables: $|V|$ = Size of the vocabulary $|D|$ = Number of documents $|Q|$ = Number of queries $\overline{l_d}$ = Average number of terms in a document $\overline{l_q}$ = Average number of terms in a query To compute the similarity between a query and a document using an inverted index, we only need to consider the terms present in both the query and the document. This can be done efficiently by iterating over the terms in the query and looking up the corresponding documents in the inverted index. The time complexity of this approach would be:

$$\text{Time Complexity} = |Q| \times \overline{l_q} \times (\overline{l_d} + \log(|D|))$$

For each query ($|Q|$): For each term in the query ($\overline{l_q}$): Look up the corresponding documents in the inverted index ($\log(|D|)$ time complexity, assuming a balanced data structure like a tree or hash table). We can compute the similarity between the query and the retrieved documents ($\overline{l_d}$ time complexity) In contrast, the time complexity of the original approach without using an inverted index was:

$$\text{Time Complexity} = |Q| \times |D| \times |V|$$

By using an inverted index, we have effectively reduced the time complexity from being proportional to the product of the number of queries, documents, and vocabulary size, to being proportional to the product of the number of queries, average query length, and the sum of the average document length and the logarithm of the number of documents.

$$\text{Time Complexity Inverted Index} = |Q| \times \overline{l_q} \times (\overline{l_d} + \log(|D|)) \ll |Q| \times |D| \times |V|$$

This reduction in time complexity can be significant, especially when dealing with large document collections and a large vocabulary size.

6 Spell Check

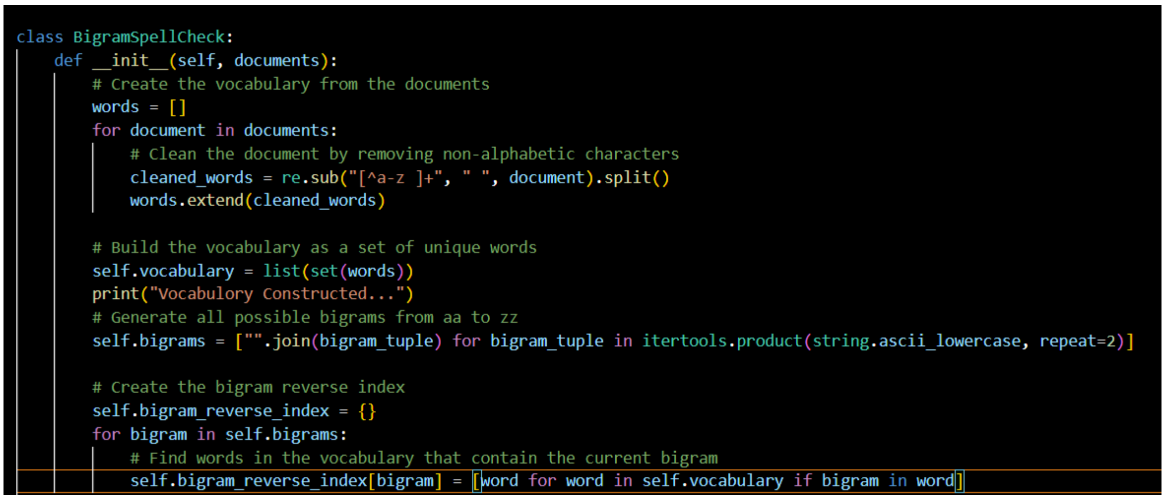
6.1 Bigrams

6.1 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.
Ans:
In order to find the first 5 candidate corrections:

- we initially constructs a **vocabulary** by iterating through each document in the provided dataset. This process involves cleaning the documents by removing **non-alphabetic** characters and collecting the resulting words.

- The vocabulary is then formed as a set of **unique words** from this cleaned word list.
- Following this, the script generates all possible **bigrams** from **'aa' to 'zz'** using itertools.product and the English alphabet, creating an exhaustive list of bigrams.
- Subsequently, a **reverse index** is established, wherein keys represent bigrams, and values are lists of words from the vocabulary that contain the respective bigram.

This reverse index acts as an efficient tool for swiftly retrieving words associated with a specific bigram.

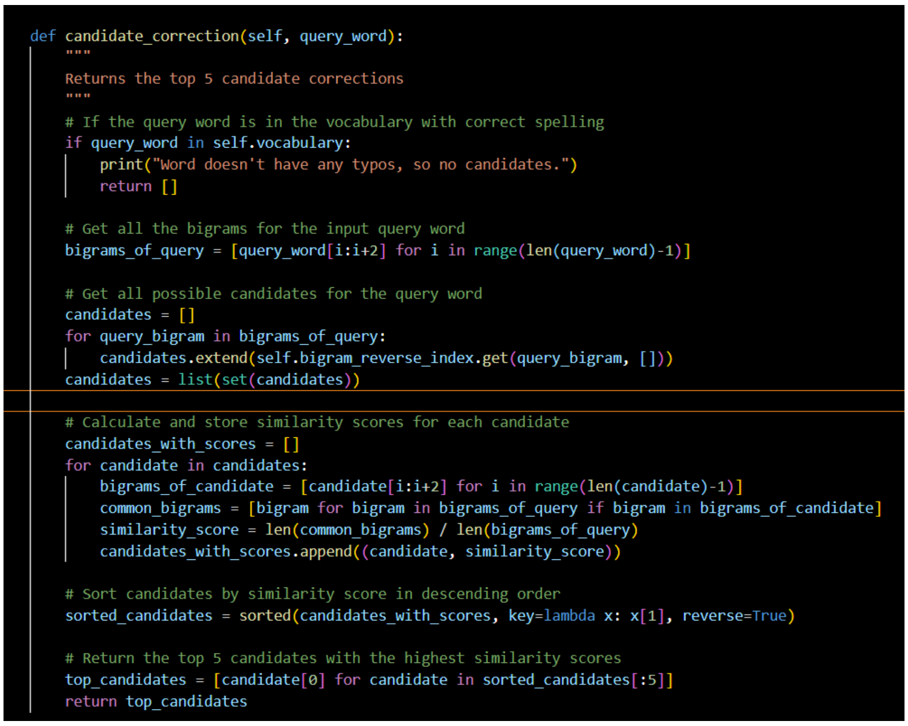


```python
class BigramSpellCheck:
    def __init__(self, documents):
        # Create the vocabulary from the documents
        words = []
        for document in documents:
            # Clean the document by removing non-alphabetic characters
            cleaned_words = re.sub("[^a-z ]+", " ", document).split()
            words.extend(cleaned_words)

        # Build the vocabulary as a set of unique words
        self.vocabulary = list(set(words))
        print("Vocabulory Constructed...")
        # Generate all possible bigrams from aa to zz
        self.bigrams = ["".join(bigram_tuple) for bigram_tuple in itertools.product(string.ascii_lowercase, repeat=2)]

        # Create the bigram reverse index
        self.bigram_reverse_index = {}
        for bigram in self.bigrams:
            # Find words in the vocabulary that contain the current bigram
            self.bigram_reverse_index[bigram] = [word for word in self.vocabulary if bigram in word]
```

Figure 8: Creating Reverse index implementation of bigrams from vocabulary.

Once the reverse index has been established, the **candidate_correction** function proceeds to find the top 5 candidate corrections for a given query word. Here's a step-by-step approach:

- We begin by checking if the query word is already in the constructed vocabulary. If so, print a message stating that the word has no typos, and **no candidates** are needed.
- Then, we obtain all the **bigrams** for the input query word by iterating through its characters. This helps in identifying potential candidates with **similar bigrams**.
- Retrieve all possible candidates for the query word based on its bigrams. Using the prebuilt **bigram reverse index** to efficiently obtain a list of words containing the query bigram.
- For each candidate, compute a **similarity score** by determining the proportion of common bigrams between the query word and the candidate. This score is calculated as the ratio of **common bigrams** to the **total number of bigrams** in the query word.
- Maintain a list (candidates_with_scores) to store each candidate along with its calculated similarity score.
- Sort the list of candidates with scores in descending order based on the similarity scores. Extract the top 5 candidates from the sorted list, considering those with the **highest similarity scores** as the most likely corrections.



```python
def candidate_correction(self, query_word):
    """
    Returns the top 5 candidate corrections
    """
    # If the query word is in the vocabulary with correct spelling
    if query_word in self.vocabulary:
        print("Word doesn't have any typos, so no candidates.")
        return []

    # Get all the bigrams for the input query word
    bigrams_of_query = [query_word[i:i+2] for i in range(len(query_word)-1)]

    # Get all possible candidates for the query word
    candidates = []
    for query_bigram in bigrams_of_query:
        candidates.extend(self.bigram_reverse_index.get(query_bigram, []))
    candidates = list(set(candidates))

    # Calculate and store similarity scores for each candidate
    candidates_with_scores = []
    for candidate in candidates:
        bigrams_of_candidate = [candidate[i:i+2] for i in range(len(candidate)-1)]
        common_bigrams = [bigram for bigram in bigrams_of_query if bigram in bigrams_of_candidate]
        similarity_score = len(common_bigrams) / len(bigrams_of_query)
        candidates_with_scores.append((candidate, similarity_score))

    # Sort candidates by similarity score in descending order
    sorted_candidates = sorted(candidates_with_scores, key=lambda x: x[1], reverse=True)

    # Return the top 5 candidates with the highest similarity scores
    top_candidates = [candidate[0] for candidate in sorted_candidates[:5]]
    return top_candidates
```

Figure 9: Extraction of top 5 candidate corrections code implementation

- **Top 5 candidate corrections for ’boundery’:** [bounded’, ’unbounded’, ’boundary’, ’underlying’, ’underneath’]
- **Top 5 candidate corrections for ’transiant’:** [’transient’, ’transparent’, ’transition’, ’transit’, ’transcendant’]
- **Top 5 candidate corrections for ’aerplain’:** [’interplanetary’, ’explained’, ’explaining’, ’powerplants’, ’airplanes’]

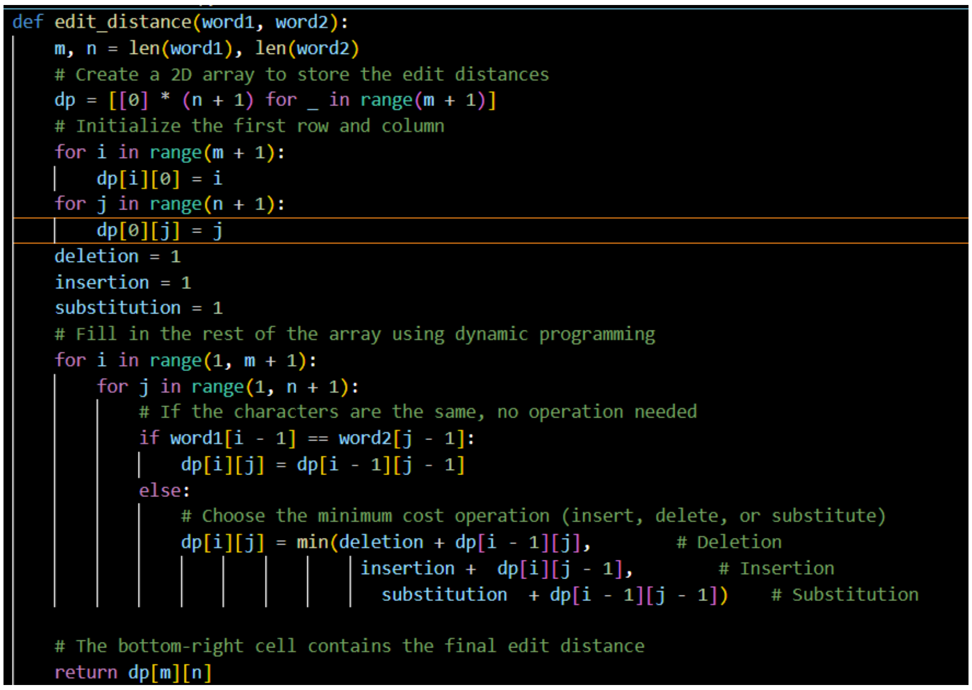
6.2 Top Five Candidate Corrections

6.2 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.

Ans:

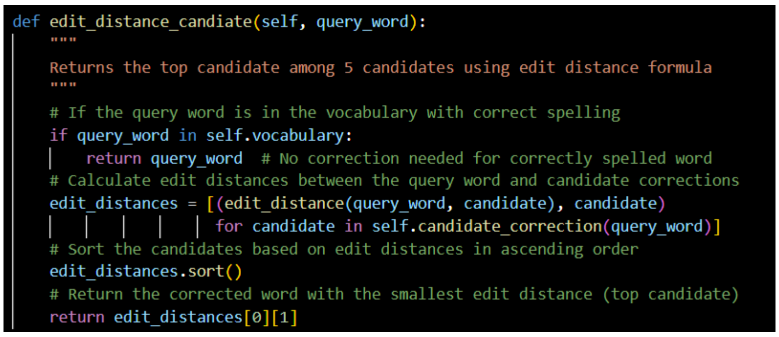
In order to find candidate among the top 5 closest to the typo using the Edit Distance function, we created **edit_distance** function in separate **Levenshtein.py** file. It has been implemented using similar **Dynamic-Programming** approach mentioned in class and each cost of operation is assigned to 1.

Now, using edit_distance function we use **BigramSpellCheck** class, present in **spell_check.py** file to calculate the edit distance between incorrect query and list of top 5 candidate corrections obtained by above method. Then, we extract the candidate with the **minimum** edit distance.



```python
def edit_distance(word1, word2):
    m, n = len(word1), len(word2)
    # Create a 2D array to store the edit distances
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    # Initialize the first row and column
    for i in range(m + 1):
        dp[i][0] = i
    for j in range(n + 1):
        dp[0][j] = j
    deletion = 1
    insertion = 1
    substitution = 1
    # Fill in the rest of the array using dynamic programming
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            # If the characters are the same, no operation needed
            if word1[i - 1] == word2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            else:
                # Choose the minimum cost operation (insert, delete, or substitute)
                dp[i][j] = min(deletion + dp[i - 1][j],        # Deletion
                               insertion +  dp[i][j - 1],        # Insertion
                                 substitution  + dp[i - 1][j - 1])    # Substitution

    # The bottom-right cell contains the final edit distance
    return dp[m][n]
```

Figure 10: Edit Distance code implementation.



```python
def edit_distance_candiate(self, query_word):
    """
    Returns the top candidate among 5 candidates using edit distance formula
    """
    # If the query word is in the vocabulary with correct spelling
    if query_word in self.vocabulary:
        return query_word  # No correction needed for correctly spelled word
    # Calculate edit distances between the query word and candidate corrections
    edit_distances = [(edit_distance(query_word, candidate), candidate)
                      for candidate in self.candidate_correction(query_word)]
    # Sort the candidates based on edit distances in ascending order
    edit_distances.sort()
    # Return the corrected word with the smallest edit distance (top candidate)
    return edit_distances[0][1]
```

Figure 11: Calculation of edit distance on top 5 correction candidates.

- **Closest candidate to ’boundery’ using Edit Distance:** ’boundary’
- **Closest candidate to ’transiant’ using Edit Distance:** ’transient’
- **Closest candidate to ’aerplain’ using Edit Distance:** ’airplanes’

6.3 Valid distance measure

6.3 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.

Ans:

The basic operations include **insertion**, **deletion**, and **substitution**, each with its own associated cost. It is essential to note that Edit Distance may not always qualify as a valid distance measure, as the assignment of costs is subjective and depends on the **specific context** or application.

If we experiment with different costs for these operations, and it’s crucial to recognize that the outcome can vary significantly. For instance, assigning equal costs to all operations is a common approach, but one can also assign different costs based on the nature of the application. However, when **negative costs** are introduced, especially when considering **insertion** or **substitution**, the resulting Edit Distance can become **negative**. This implies that, under certain conditions, the dissimilarity measure may not adhere to the traditional definition of a distance metric, where non-negativity is a fundamental criterion.

7 WordNet

7.1 List of all synsets

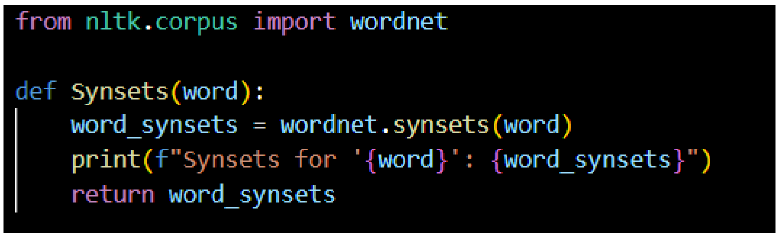
7.1 Print the list of all synsets corresponding to the words 'progress' and 'advance.'
Ans:
Synsets for 'progress':

- '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’



```
from nltk.corpus import wordnet

def Synsets(word):
    word_synsets = wordnet.synsets(word)
    print(f"Synsets for '{word}': {word_synsets}")
    return word_synsets
```

Figure 12: List of Synsets code implementation

7.2 List of all definitions

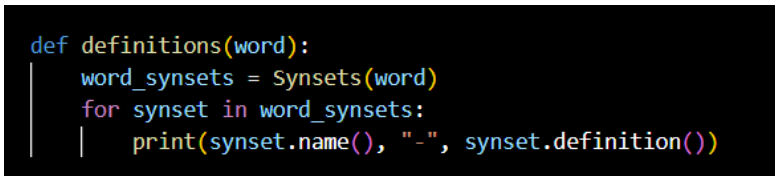
7.2 Print the definitions corresponding to the synsets obtained in the previous question.
Ans:
Definitions for ’progress’:

- **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
- **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

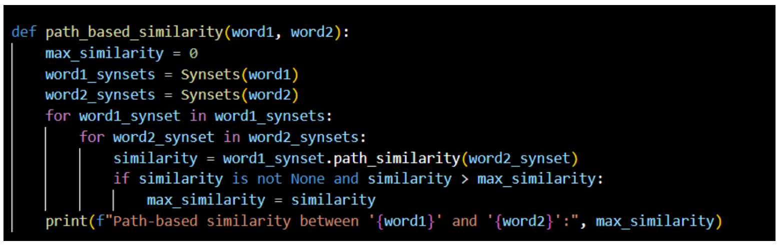


```python
def definitions(word):
    word_synsets = Synsets(word)
    for synset in word_synsets:
        print(synset.name(), "-", synset.definition())
```

Figure 13: List of Definitions code implementation

7.3 Path Based Similarity

7.3 Estimate the path-based similarity between the words 'advance' and 'progress' using the similarities between their synsets.
<u>**Ans:**</u>
The path_based_similarity function uses a nested loop to iterate through all possible pairs of synsets from the two words. For each synset of 'advance', it compares it to each synset of 'progress'. The path-based similarity between the words **'progress'** and **'advance'** is **1.0**.



```python
def path_based_similarity(word1, word2):
    max_similarity = 0
    word1_synsets = Synsets(word1)
    word2_synsets = Synsets(word2)
    for word1_synset in word1_synsets:
        for word2_synset in word2_synsets:
            similarity = word1_synset.path_similarity(word2_synset)
            if similarity is not None and similarity > max_similarity:
                max_similarity = similarity
    print(f"Path-based similarity between '{word1}' and '{word2}':", max_similarity)
```

Figure 14: Path based similarity code implementation

The path_based_similarity function takes two words as input. It retrieves the synsets for each word, iterates through all possible pairs of synsets, calculates the path-based similarity using the **path_similarity** method, and keeps track of the **maximum similarity score** found. Finally, it prints the maximum path-based similarity between the two words.

7.4 Number of Calls

7.4 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?

Ans:
Since there are **’m’** synsets for ’advance’ and **’n’** synsets for ’progress’, the number of pair-wise comparisons made in the nested loop will be **’m * n’**. Therefore, the inbuilt path-based similarity function will be called **’m * n’** times to compute the similarity between the two words.