CS6370: Natural Language Processing

Assignment 1 (Part-A)

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

Name: Roll No.:

|  |  |
| --- | --- |
|  |  |

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 for English texts could rely on punctuation marks like periods (.), exclamation marks (!), and question marks (?) as sentence boundaries. **Identify Sentence Boundaries**: Split the text into sentences by splitting at every occurrence of ., !, or ?. Foreseeable Issues with the Proposed Approach  1. **Abbreviations**:    * Example: "Dr. Satanu is a U.S.A. citizen."    * Issue: The algorithm might incorrectly split "Dr.", "U.S.A.", etc., into separate sentences.    * Solution: Maintain a list of common abbreviations and check against it before splitting. 2. **Decimal Numbers**:    * Example: "The value is 7.814."    * Issue: The algorithm might split at the decimal point.    * Solution: Use regex to exclude numbers with decimal points from splitting. 3. **Ellipses**:    * Example: "She said... and then left."    * Issue: The algorithm might split at each period in the ellipsis.    * Solution: Treat ellipses (...) as a single unit. 4. **Quotations and Parentheses**:    * Example: "He said, 'I am tired.' Then he left."    * Issue: The algorithm might split within quoted or parenthesized text.    * Solution: Use regex to handle nested punctuation.  Strategies to Handle Issues  1. **Rule-Based Enhancements**:    * Use regex patterns to handle abbreviations, decimals, and ellipses.    * Example: r'(?<!\w\.\w.)(?<![A-Z][a-z]\.)(?<=\.|\?|\!)\s' 2. **Predefined Lists**:    * Maintain a list of common abbreviations and titles to avoid incorrect splits.  **3 Use a Pre-Trained Tokenizer**: |

1. 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]

|  |
| --- |
| 1. **Handling Abbreviations**: (Ex- "Dr.", "U.S.A.", "e.g.") 2. 2. **Handling Decimal Numbers**: (Ex-7.896) 3. **Handling Ellipses**:  (...) and avoid splitting4. **Handling Quotations and Parentheses**:5. **Handling Multiple Punctuation Marks**:**Punkt Tokenizer is Better Than the Simple Top-Down Approach** **Learns from Data**: It uses unsupervised machine learning to identify sentence boundaries based on patterns in the text.  **Handles Ambiguity**: It can distinguish between periods used as sentence boundaries and those used in abbreviations, decimal numbers, etc.  **Adapts to Context**: It considers the context of the text (e.g., surrounding words and punctuation) to make better decisions. |

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

State a possible scenario where

1. Your approach performs better than the Punkt Tokenizer
2. Your approach performs worse than the Punkt Tokenizer [4 marks]

|  |
| --- |
| Proposed Top-Down Method & **Pre-Trained Punkt Tokenizer are performed on sentences segmentation in sentenceSegmentation.py****Scenarios for Sentence Segmentation Performance:****a. When the Naive Approach (Top-Down) Performs Better than the Punkt Tokenizer:**  * **Domain-Specific Texts**: If the dataset contains structured technical documents, research papers, or formal reports where sentences consistently end with periods, exclamation marks, or question marks, the **naive approach** (regex-based) might work well. * **Acronyms & Abbreviations**: The Punkt tokenizer sometimes mistakenly splits at **abbreviations** (e.g., "Dr.", "e.g.", "i.e.") while a well-designed naive approach that handles exceptions could perform better. * **Custom Formatting**: If documents contain specific **bullet points, numbered lists, or structured headings**, regex-based segmentation may be better suited to maintain logical sentence structures.  **b. When the Punkt Tokenizer Performs Better than the Naive Approach:**  * **Conversational or Informal Text**: If the dataset includes **emails, chat logs, or literature**, the Punkt tokenizer can **adaptively recognize sentence boundaries** even when punctuation is inconsistent. * **Complex Sentence Structures**: Punkt handles **ellipsis ("..."), quotations, and parenthetical statements** better, avoiding incorrect splits. * **Unconventional Punctuation**: If the document contains unusual punctuation styles, such as **"Mr. Smith went to Washington. He said, 'I’m happy.'"**, Punkt's **ML-based heuristics** would perform better. |

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** for tokenization in English text can be implemented using **whitespace and punctuation-based splitting**:   1. **Split text into words** using whitespace (split() method). 2. **Remove punctuation** while keeping contractions intact. 3. **Convert all tokens to lowercase** for consistency.  **Where This Approach Fails:****a. Handling Complex Word Structures**  * **Hyphenated words**: "state-of-the-art" → Splits into ["state", "of", "the", "art"] instead of keeping it together. * **Emails & URLs**: "user@example.com" → Splits incorrectly.  **b. Handling Special Characters**  * "C++ programming" → Becomes ["c", "programming"], losing ++. * Mathematical expressions like "5.5 + 3.2 = 8.7" get fragmented incorrectly.  **c. Handling Contextual Meaning**  * "U.S. is a country" → May mistakenly split "U.S." into ["U", "S"]. * "Mr. Smith went to Washington." → "Mr." might be split incorrectly. |

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

|  |
| --- |
| The **Penn Treebank Tokenizer** in **NLTK** follows a **bottom-up** approach.  It **starts with raw text and progressively applies rules** to build structured tokens.   * Instead of splitting based on predefined sentence structures (top-down), it **identifies token boundaries based on linguistic patterns**. * It applies **regular expressions and handcrafted rules** to handle contractions, punctuation, and special cases.  **When the Penn Treebank Tokenizer Fails:****a. Handling Non-Standard Language**  * **Social media text:** "gonna", "wanna", "u" might not be tokenized properly. * **Emojis & hashtags:** "I ❤️ NLP! #AI" → Fails to tokenize properly.  **b. Handling Special Characters**  * **Email addresses & URLs:** "contact@example.com" gets fragmented incorrectly. * **Code snippets:** "int x = 10;" is not preserved correctly. |

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

State a possible scenario along with an example where:

1. Your approach performs better than Penn Treebank Tokenizer
2. Your approach performs worse than Penn Treebank Tokenizer

[4 marks]

|  |
| --- |
| The proposed top down method and penn treebank tokenizer codes is in word tokenize.py**Scenarios for Word Tokenization Performance****a. When the Naive (Top-Down) Approach Performs Better than Penn Treebank Tokenizer** **Scenario:** Handling **technical terms, hyphenated words, and custom domain-specific text**   * A simple **whitespace & punctuation-based tokenizer** may work better when text contains **hyphenated words** or **compound terms** that should remain intact.   **Example:**  text  CopyEdit  "The state-of-the-art model achieved 95% accuracy."   * **Naive Tokenizer Output:** ['The', 'state-of-the-art', 'model', 'achieved', '95', 'accuracy'] ✅ (Preserves "state-of-the-art") * **Penn Treebank Tokenizer Output:** ['The', 'state-of-the', '-', 'art', 'model', 'achieved', '95', '%', 'accuracy', '.'] ❌ (Splits unnecessarily)  **b. When the Penn Treebank Tokenizer Performs Better than the Naive Approach** **Scenario:** Handling **contractions and possessives correctly**   * The **Penn Treebank tokenizer** applies specialized rules for handling **apostrophes, contractions, and possessives**, which a basic regex-based approach might fail to capture.   **Example:**  text  CopyEdit  "John's book isn't on the table."   * **Naive Tokenizer Output:** ['John', 's', 'book', 'isn', 't', 'on', 'the', 'table'] ❌ (Incorrect splitting of "John's" and "isn't") * **Penn Treebank Tokenizer Output:** ['John', "'s", 'book', 'is', "n't", 'on', 'the', 'table', '.'] ✅ (Correct handling of possessives and contractions) |

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]

|  |
| --- |
| **Difference Between Stemming and Lemmatization****1. Stemming**  * Removes suffixes to get the root form of a word. * Uses rule-based truncation. * May produce **non-real words**. * Faster but **less accurate**. * Example:   + "Running" → "runn" (incorrect truncation)   + "Studies" → "studi"  **2. Lemmatization**  * Converts a word to its **dictionary base form** using linguistic rules. * Uses **morphology and POS tagging**. * Always produces **valid words**. * Slower but **more accurate**. * Example:   + "Running" → "run" (correct base form)   + "Studies" → "study"  **Comparison Table (Plain Text for Copying)** | Word | Stemming (Porter Stemmer) | Lemmatization (WordNet) |  |-----------|--------------------------|-------------------------|  | Running | runn | run |  | Studies | studi | study |  | Caring | care | care | |

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

|  |
| --- |
| stemming/lemmatization on the word-tokenized text from the Cranfield Dataset. inflectionReduction.py |

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]

|  |
| --- |
| Code give |

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

|  |
| --- |
| Code |

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

|  |
| --- |
| Corpus-Specific Stopwords (23):  {'by', 'it', 'is', 'be', 'as', 'and', 'this', 'on', 'which', 'for', 'at', 'with', 'results', 'an', 'flow', 'a', 'that', 'from', 'are', 'the', 'in', 'to', 'of'}  NLTK Stopwords (198):  {'it', 'my', 'to', 'doesn', 'yours', 'most', "wasn't", 'will', 'been', 'between', 'ain', 'aren', 'being', 'when', "we're", 'too', 'each', 'has', 'shouldn', 'were', 'not', 'what', "they'd", 'o', 'him', 'very', 'because', 'her', 'his', 'ours', 'a', 'd', "he'll", 'into', "it'll", 'should', 'that', "we'd", 'until', "mightn't", 'than', 'before', 'shan', 'does', 'all', 'haven', 'these', "hasn't", "she's", "it's", 'hers', 'himself', 'if', 'can', 'did', 'your', 'you', 'couldn', 'no', "wouldn't", 'those', "i'm", "you're", 'weren', 'mustn', "she'll", 'was', 'or', 'further', "weren't", 'about', 'above', 'won', "won't", "you'll", 'didn', 'had', 'an', 'i', 'again', "you've", 'off', 'they', 'the', "we've", 'who', 'mightn', 'in', 'me', 'ourselves', "that'll", 'm', 'where', 'out', "doesn't", 're', 'more', 'is', 'our', "hadn't", 'don', 'having', 'have', 'he', 'be', "couldn't", "needn't", 'there', 'as', 'after', 'them', 'doing', 'both', 'down', 'while', "you'd", 'then', 'at', "haven't", 'with', 'same', 'any', 'myself', 'once', 'such', 'how', 'during', 'isn', 'their', 'only', "shan't", 'll', "mustn't", 'few', 'needn', 'itself', 'wasn', 'whom', "we'll", "don't", 'themselves', 've', 'own', "i'll", 'of', 'by', "she'd", 'so', 'nor', 'against', "i'd", "they're", "isn't", "i've", 'am', "it'd", 'but', "should've", "aren't", 'up', 'ma', 'and', 'yourself', 'do', 'this', 'on', 'over', "they'll", 'for', 'through', 'herself', "didn't", 'which', 'below', 'wouldn', "shouldn't", 'we', "they've", 'yourselves', 'its', 'are', 'from', 'y', 'hadn', "he'd", 'she', "he's", 'just', 'other', 'some', 'why', 'under', 'now', 'here', 't', 's', 'hasn', 'theirs'}  Common Stopwords (21):  {'by', 'it', 'is', 'be', 'as', 'and', 'this', 'on', 'which', 'at', 'for', 'with', 'an', 'a', 'from', 'that', 'are', 'the', 'in', 'to', 'of'} |

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]

|  |
| --- |
| **Step 1: Understanding TF-IDF Representation** Each query and each document is represented as a **TF-IDF vector** over a vocabulary V, which consists of the unique terms across all queries and documents.   * **TF (Term Frequency):** Measures how often a term appears in a document/query. * **IDF (Inverse Document Frequency):** Measures the importance of a term across the entire document corpus.   Given a **query set** Q (with ∣Q∣=Q queries) and a **document set** D (with ∣D∣=D documents), we need to compute the similarity between each query and each document. **Step 2: Computing TF-IDF Similarity** The similarity between a query and a document using **TF-IDF dot product** is computed as:  Similarity(q,d)=t∈V∑​TF-IDF(q,t)×TF-IDF(d,t)  where:   * t is a term from the vocabulary V, * TF-IDF(q,t) is the TF-IDF score of term t in query q, * TF-IDF(d,t) is the TF-IDF score of term t in document d.  **Step 3: Number of Computations** We compute this similarity for every query-document pair. The computational cost is determined by:   1. **Iterating over all queries:** Q 2. **Iterating over all documents:** D 3. **Iterating over all terms in the vocabulary:** V (assuming full vocabulary usage for each computation)   Thus, the **total number of computations** required is: O(Q×D×V). |

1. 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]

|  |
| --- |
| **1. Without Inverted Index:****Time Complexity Analysis**  * **Step (a): Compute TF-IDF Vectors**   + **For all queries**: O(QV) (Each query processes V terms)   + **For all documents**: O(DV) (Each document processes V terms) * **Step (b): Compute Cosine Similarity**   + **For all query-document pairs**: O(QDV) (Since each of the QD query-document pairs requires computing similarity across all V terms)   Thus, the overall time complexity:  O(QV)+O(DV)+O(QDV)=O(QDV)  ✅ **This is correct**. **2. With Inverted Index:** An **inverted index** allows direct access to documents that contain specific terms, thereby reducing the number of computations by eliminating unnecessary comparisons. **(a) Create Inverted Index**  * **Step (i): Construct the index**   + Tokenize each document and map terms to corresponding document lists.   + **Let Dt​ be the number of documents per term** (on average).   + This requires processing all terms across all documents.   + **Time complexity:** O(DV) (scanning all D documents with V terms each)   ✅ **Correction:** The given solution states O(DDt​), but **the correct time complexity for building the index is O(DV), as we process all document-term pairs**. **(b) Retrieve Relevant Documents**  * **Step (i): Find relevant documents for query terms using the inverted index**   + For each query, fetch the list of documents containing query terms.   + **Time Complexity:** O(QV+Dt​) (retrieving the document list for each query term and merging)   ✅ **This is mostly correct, but Dt​ should be interpreted as the total number of relevant documents for all query terms.** **(c) Compute Cosine Similarity**  * Instead of computing **similarity for all QD pairs**, we compute it only for relevant document-query pairs. * Let Li​ be the number of relevant documents for the ith query. * **Time Complexity:** O(i=1∑Q​Li​V) Since Li​≪D, this significantly reduces the computational burden.   ✅ **This is correct**. **Final Complexity Comparison** ApproachTime Complexity**Without Index**O(QDV)**With Index**O(DV)+O(QV+Dt​)+O(∑i=1Q​Li​V)  Since Li​≪D, the complexity of the inverted index approach is significantly lower than the brute-force O(QDV). **Key Points Verification** ✅ **(a) Inverted index allows for quick access to relevant documents**. ✅ **(b) It reduces the search space, avoiding unnecessary computations**. ✅ **(c) The document list is retrieved efficiently for each query term**. ✅ **(d) Similarity computation is streamlined using term-level indexing**. ✅ **(e) Redundant computations are eliminated, improving efficiency**. |