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

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

| Name: | Roll No.: |
|---|---|
| MANGLESH PATIDAR | CS24M025 |

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]

Ans. 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: 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: 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: 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.

2.Predefined Lists:
- Maintain a list of common abbreviations and titles to avoid incorrect splits.

3. Use a Pre-Trained Tokenizer: Punkt Sentence Tokenizer.

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]

Ans. The Punkt Sentence Tokenizer in NLTK do differently from the simple top-down approach are:

- Handling Abbreviations: (Ex- "Dr.", "U.S.A.", "e.g.")
- Handling Decimal Numbers: (Ex-7.896)
- Handling Ellipses: (...) and avoid splitting
- Handling Quotations and Parentheses:
- 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.

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]

Ans Proposed Top-Down Method & Pre-Trained Punkt Tokenizer are performed on sentences segmentation in SentenceSegmentation.py

**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 (Using 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.

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]

Ans. It can be implemented using whitespace and punctuation-based splitting:
1. Split text into words using whitespace (using 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.

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

Ans: The Penn Treebank Tokenizer in NLTK follows a bottom-up approach.
It starts with raw text and progressively applies rules to build structured tokens.

• it identifies token boundaries based on linguistic patterns.

• It applies regular expressions and handcrafted rules to handle contractions, punctuation, and special cases.

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]

Ans. The proposed top-down method and Penn treebank tokenizer codes is in word tokenize.py
a. **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 "The state-of-the-art model achieved 95% accuracy."
• Naive Tokenizer Output:
['The', 'state-of-the-art', 'model', 'achieved', '95', 'accuracy'
• Penn Treebank Tokenizer Output:  Sometimes.
['The', 'state-of-the', '-', 'art', 'model', 'achieved', '95', '%', 'accuracy', '.']

**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 :"John's book isn't on the table."
• Naive Tokenizer Output: ['John', 's', 'book', 'isn', 't', 'on', 'the', 'table']
• Penn Treebank Tokenizer Output: ['John', "'s", 'book', 'is', "n't", 'on', 'the', 'table', '.']

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:
• Happily → Happi , 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:
• Happily → Happy , Studies → study

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

Ans. Stemming/lemmatization on the word-tokenized text from the Cranfield Dataset.
Code in 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]

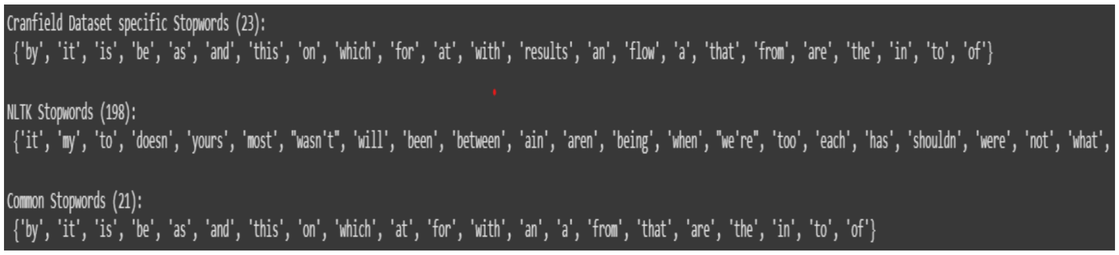
Ans. Remove stopwords from NLTK code stopwordRemoval.py

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

Steps in the Approach:
- Load the dataset.
- Extract text from documents.
- Tokenize & preprocess (lowercasing & filtering non-alphabetic tokens).
- Compute Document Frequency (DF) (how many docs contain each word).
- Compute IDF using IDF = log(total_document_count / (word_document_freq + 1)).
- Set threshold (words with IDF ≤ 1.0 are stopwords).
- Print stopwords list.

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]

Ans. Code removeStopword_tfidf.py



```
Cranfield Dataset 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',

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]

Ans. Q= Queries ,D=Document
TF-IDF vectors for queries Q and documents D over a vocabulary V
each query and document is represented as a TF-IDF vector :-

1.compute cosine similarity for every query-document pair, so the total number of similarity computations is Q×D
2.Each similarity computation (dot product) takes O(|V|).
3.Precomputing norms adds a O(D|V|+Q|V|) overhead.
Thus, the overall computational complexity is: O(Q×D×|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]

**Ans. The inverted index can help reduce the time complexity**
Compute the similarity between each query and each document by iterating through all terms in the vocabulary V.
Time complexity of O(Q×D×|V|).
However, many terms in V are **0(ZERO)** in most TF-IDF vectors (i.e., they do not appear in a document).
This sparsity can be exploited using an inverted index to improve efficiency.
Inverted Index is a data structure that maps each term in the vocabulary VVV to a list of documents that contain it. Specifically, it consists of:

- A posting list: For each term ttt, a list of document IDs where ttt appears, along with its TF-IDF weight in each document.

Construction of the Inverted Index

I[t]={(d,w)| d∈D, w>0}
where:

- I be the inverted index:
- t is a term in the vocabulary VVV,
- d is a document ID,
- w is the TF-IDF weight of term t in document d

.
**Reducing Complexity using the Inverted Index**

Step 1: Query Processing
For each query q, we retrieve only the documents that share at least one term with q. Instead of iterating over all documents, we:
1. Extract the terms present in q.
2. Use the inverted index to retrieve only the relevant documents.

Step 2: Sparse Dot Product Computation
Instead of computing the full dot product over all |V| dimensions, we only compute it over the nonzero dimensions (terms present in both the query and the document).
For a given query q:
• Let Tq be the number of nonzero terms in q.
• Each term t in q has a posting list of relevant documents I[t].
• The number of documents relevant to q is at most ∑t∈q|I[ which is much smaller than D in practice.
New Time Complexity
Let:
• Tq = Number of terms in query q <|V|.
• Nq = Number of candidate documents matched via the inverted index
• A= Average number of documents per term in the posting list (sparsity factor)
Time complexity per query is: O(Tq·A)
For queries, the total complexity is: O(∑Tq·A) q ∈ Since Tq≪|V| and A≪D in practice (due to sparsity).