CS6370: Natural Language Processing

Assignment 1

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

Name:	Roll No.:
Ishaan Agarwal	EE20B046

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]

A simplistic top-down approach to sentence segmentation for English texts could involve the following steps:

- **Identify Potential Sentence Boundaries**:
    - Start by considering common punctuation marks that typically denote the end of a sentence, such as periods (.), exclamation marks (!), and question marks (?).
    - Additionally, consider other punctuation marks that might indicate sentence boundaries in specific contexts, such as colons (:), semicolons (;).
- **Scan the Text**:
    - Scan through the text character by character, looking for potential sentence boundaries based on the identified punctuation marks.
    - Whenever a potential sentence boundary is encountered, mark it as the end of a sentence.

Potential Issues and Strategies:

- **Abbreviations and Acronyms**:
    - Issue: Abbreviations and acronyms might be mistaken as sentence boundaries.
    - Example: "The meeting will be held at 10 a.m. in room 101."
    - Strategy: Maintain a list of common abbreviations to avoid misinterpretation.
- **Ellipses and Dots in Non-Sentence Contexts**:
    - Issue: Ellipses and dots in non-sentence contexts can trigger false boundaries.
    - Example: ”Visit our website at www.example.com for more information…"
    - Strategy: Implement contextual checks to differentiate between punctuation and other usage.

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

The Punkt Sentence Tokenizer in NLTK:
1. Is a bottom up approach which utilises machine learning for accurate boundary identification.
2. Considers contextual cues like capitalisation and punctuation usage.
3. Is robust across text types, languages, and styles.
4. Achieves high accuracy compared to simpler approaches.

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]

Implementation for (a) and (b) can be found in /code/sentenceSegmentation.py.

Better than Punkt Tokenizer:

- The simplistic top-down approach may excel in scenarios involving texts with a very consistent and predictable structure, such as technical documents or legal texts. In these contexts, where sentences typically adhere closely to grammatical rules and punctuation conventions, the straightforward rules-based approach of the top-down method can effectively identify sentence boundaries without the need for complex machine learning models.

Worse than Punkt Tokenizer:

- Conversely, the simplistic top-down approach may struggle when confronted with texts that exhibit greater variability and complexity, such as informal conversations, social media posts, or creative writing. In these cases, where sentence boundaries may be less clear-cut and punctuation usage may deviate from standard norms, the rules-based approach of the top-down method may lead to inaccuracies or incomplete segmentation. The Punkt Tokenizer, with its ability to learn from diverse linguistic patterns and consider contextual cues, is better equipped to handle such nuanced language usage and accurately identify sentence boundaries.

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 involve the following steps:

- **Identify Potential Token Boundaries**:
  - Define common delimiters that typically separate tokens, such as spaces, punctuation marks, and special characters.
- **Scan the Text**:
  - Iterate through the text character by character, identifying potential token boundaries whenever a delimiter is encountered.
- **Refine Tokens**:
  - Trim any leading or trailing whitespace from tokens.
  - Convert tokens to lowercase for ease of computation in further steps.

Specific situations where this approach may fail to produce expected results include:

- **Complex Tokenization Rules**:
  - Situations where tokenization rules are complex, such as handling hyphenated words, compound nouns, or words with apostrophes (e.g., "mother-in-law", "New York", "don't").
  - Example: "The New York-based company is known for its state-of-the-art products." In this case, the simplistic approach might tokenize "New York" as two separate tokens, leading to incorrect tokenization.
- **Ambiguous Punctuation Usage**:
  - Cases where punctuation marks are used ambiguously, such as in abbreviations, URLs, or numerical values.
  - Example: "Please visit www.example.com for more information." Here, the simplistic approach might tokenize “www.example.com”as separate tokens, even though it should be considered as a single entity.

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 top-down approach to tokenization.

In a top-down approach, the tokenizer breaks the entire text into smaller units using predefined rules, such as spaces and punctuation marks, to create tokens.

The Penn Treebank tokenizer is based on the Penn Treebank Project developed in the University of Pennsylvania, it follows a set of rules derived from the Penn Treebank corpus, a large annotated corpus of English text. It employs a series of regular expressions and heuristics to identify boundaries between tokens. For example, it handles punctuation, contractions, hyphenated words, and abbreviations in a systematic manner.

By using a top-down approach, the Penn Treebank tokenizer efficiently segments text into tokens based on language-specific rules and patterns, making it suitable for a wide range of English text data.

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]

Implementation for (a) and (b) can be found in /code/tokenization.py.

Scenario where the simplistic top-down approach performs better than the Penn Treebank Tokenizer:

In the case of processing text with irregular punctuation or unconventional tokenization requirements, the simplistic top-down approach may outperform the Penn Treebank Tokenizer. For example, consider the tokenization of user-generated content from social media platforms where users often employ non-standard language, emojis, and unconventional punctuation. The top-down approach may be more effective in accurately segmenting such text into tokens without making assumptions based on predefined rules, which the Penn Treebank Tokenizer may struggle to handle.

Example:

Text: "Had a great time at the beach today! 😎🌊 #beachday #funinthesun"

Simplistic Top-Down Approach:

- Tokens: ["Had", "a", "great", "time", "at", "the", "beach", "today", "!", "😎", "🌊", "#beachday", "#funinthesun"]

Penn Treebank Tokenizer:

- Tokens: ["Had", "a", "great", "time", "at", "the", "beach", "today", "!", "😎", "🌊", "#", "beachday", "#", "funinthesun"]

In this example, the difference lies in the handling of hashtags (#). The simplistic top-down approach treats hashtags as part of the token, which might be important to highlight social media sentiment, while the Penn Treebank Tokenizer separates hashtags as individual tokens.

Scenario where the simplistic top-down approach performs worse than the Penn Treebank Tokenizer:

Words like "I'll" are tokenised into "I" and "'ll", this when lemmatised will result in "I" and "will", which is what is required, but if we were to use our simplistic top down approach, it just splits the string from the instance of " ' ", and thus the output after lemmatisation will be "I" and "ll" and thus some amount of meaning is lost.

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 reduces words to their root forms by chopping off affixes without considering linguistic meaning. It's simpler and faster but may produce non-existent words.

Example:
Word: "running"
Stemmed: "run"

Lemmatization reduces words to their base forms (lemmas) considering linguistic meaning and context. It's more accurate but computationally intensive.

Example:
Word: "better"
Lemmatized: "good"

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

Implementation can be found in /code/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]

Implementation can be found in the function "fromList" in /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:
1. Calculate the frequency of each word/token in the corpus.
2. Set an upper threshold for stopwords i.e words occurring more than threshold number of times (as a fraction of total words) are identified as stopwords.

This streamlined process focuses on efficiently identifying and refining stopwords specific to your corpus, optimizing for NLP tasks.

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]

Implementation can be found in the function "bottomUpStopwordRemoval" in /code/stopwordRemoval.py, outputs can be found at stopwordsBottomUp.txt. (5% threshold).
Notice that only "the" and "of" are identified as stopwords in the bottom-up approach.

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]

   We need to calculate the cosine similarity of each query with every document in the corpus, this needs Q*D computations of cosine similarity. Each cosine similarity calculation in the worst case takes O(V) computations. Thus overall, the number of computations are O(Q*D*V).

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]

The inverted index is a data structure used in information retrieval systems to speed up searches for terms within a document corpus. It maps terms to the documents in which they appear, allowing for efficient retrieval of documents containing specific terms.

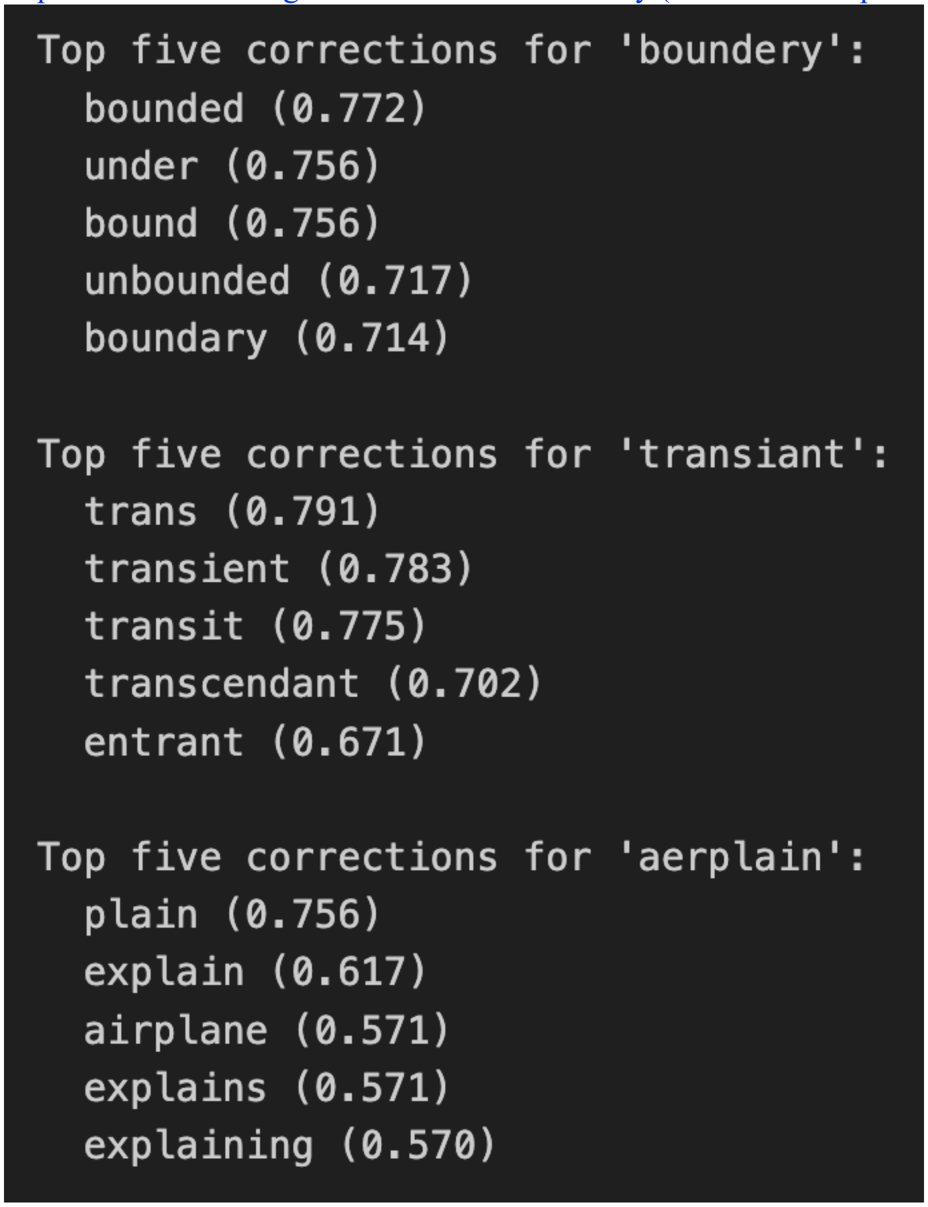
- **Inverted Index Construction**:
    - Build an inverted index that maps each term in the vocabulary V to the documents in which it appears and its corresponding TF-IDF weight.
    - Total computations: $|D| \times |V|$ to construct the inverted index.
- **Retrieval**:
    - For each query term in Q:
    - Retrieve the list of documents containing the term from the inverted index.
    - Total computations: $|Q| \times$average number of documents per query term
- **Cosine Similarity Calculation**:
    - For each retrieved document from the inverted index:
    - Compute the cosine similarity between the TF-IDF vectors of the query and the retrieved document.
    - Total computations: $|Q| \times$total number of retrieved documents$\times |V|$

By leveraging the inverted index, we significantly reduce the number of computations needed for similarity calculation because we only need to consider documents that contain at least one term from the query. This drastically reduces the number of documents to consider for similarity calculation, especially in large document corpora.

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]

Top 5 candidates along with their cosine similarity (code at /code/spellCheck.ipynb):



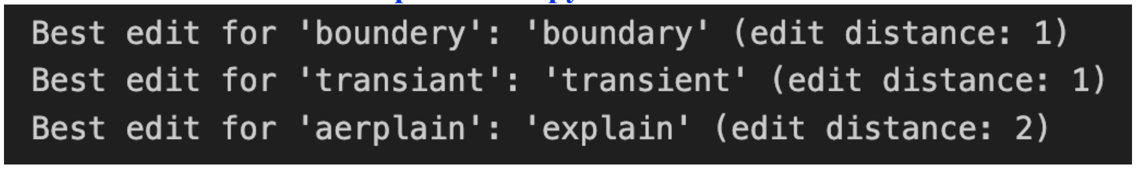
```
Top five corrections for 'boundery':
  bounded (0.772)
  under (0.756)
  bound (0.756)
  unbounded (0.717)
  boundary (0.714)

Top five corrections for 'transiant':
  trans (0.791)
  transient (0.783)
  transit (0.775)
  transcendant (0.702)
  entrant (0.671)

Top five corrections for 'aerplain':
  plain (0.756)
  explain (0.617)
  airplane (0.571)
  explains (0.571)
  explaining (0.570)
```

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]

Code can be found at /code/spellCheck.ipynb.



```
Best edit for 'boundery': 'boundary' (edit distance: 1)
Best edit for 'transiant': 'transient' (edit distance: 1)
Best edit for 'aerplain': 'explain' (edit 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]

For experimenting with different costs of insertion, deletion and substitution, we can just tweak the dynamic programming code that we already have to reflect the different costs.

Let's denote the cost of insertion, deletion and substitution as I, D and S respectively, then for edit distance to be a valid distance measure:

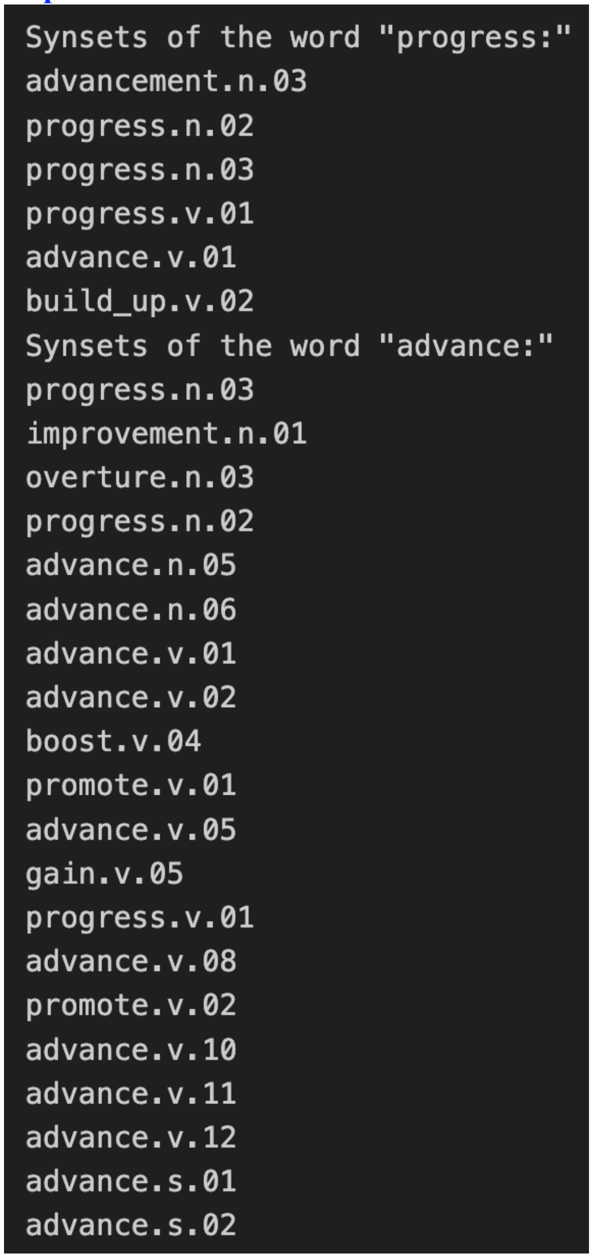
1. Non-negativity: All the costs must be non negative i.e I, D, S ≥ 0.
2. Identity of indiscernible: For the distance between a string and itself to be zero, the costs of insertion, deletion and substitution must satisfy: I+D $\geq$ S. This condition ensures that substituting a character with itself is never more expensive than deleting it and then inserting it back.
3. Symmetry: Cost of insertion should be equal to cost of deletion, so as to ensure words like "fair" and "fairy" have the same edit distance regardless of which word you start from i.e I = D.

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]

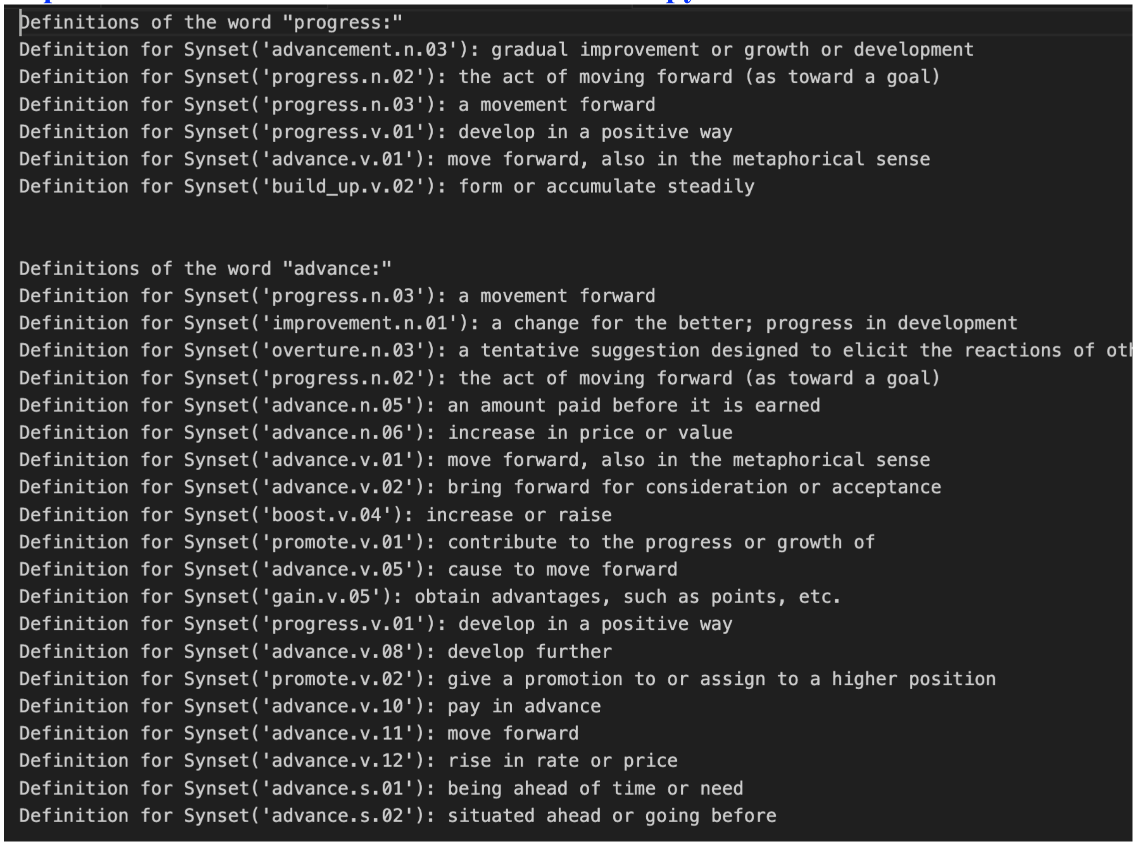
Implementation can be found at /code/wordNet.ipynb.



```
Synsets of the word "progress:"
advancement.n.03
progress.n.02
progress.n.03
progress.v.01
advance.v.01
build_up.v.02
Synsets of the word "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]

Implementation can be found at /code/wordNet.ipynb.

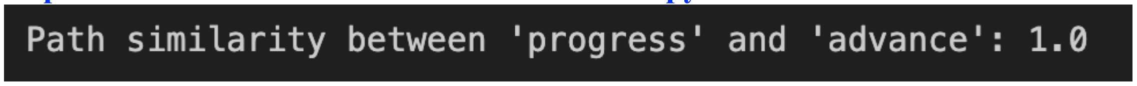


```
Definitions of the word "progress:"
Definition for Synset('advancement.n.03'): gradual improvement or growth or development
Definition for Synset('progress.n.02'): the act of moving forward (as toward a goal)
Definition for Synset('progress.n.03'): a movement forward
Definition for Synset('progress.v.01'):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


Definitions of the word "advance:"
Definition for Synset('progress.n.03'): a movement forward
Definition for Synset('improvement.n.01'): a change for the better; progress in development
Definition for Synset('overture.n.03'): a tentative suggestion designed to elicit the reactions of oth
Definition for Synset('progress.n.02'): the act of moving forward (as toward a goal)
Definition for Synset('advance.n.05'): an amount paid before it is earned
Definition for Synset('advance.n.06'): increase in price or value
Definition for Synset('advance.v.01'): move forward, also in the metaphorical sense
Definition for Synset('advance.v.02'): bring forward for consideration or acceptance
Definition for Synset('boost.v.04'): increase or raise
Definition for Synset('promote.v.01'): contribute to the progress or growth of
Definition for Synset('advance.v.05'): cause to move forward
Definition for Synset('gain.v.05'): obtain advantages, such as points, etc.
Definition for Synset('progress.v.01'): develop in a positive way
Definition for Synset('advance.v.08'): develop further
Definition for Synset('promote.v.02'): give a promotion to or assign to a higher position
Definition for Synset('advance.v.10'): pay in advance
Definition for Synset('advance.v.11'): move forward
Definition for Synset('advance.v.12'): rise in rate or price
Definition for Synset('advance.s.01'): being ahead of time or need
Definition for Synset('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]

Implementation can be found at /code/wordNet.ipynb.



```
Path similarity between 'progress' and 'advance': 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]

Since the algorithm to compute similarity between two words based on the similarities between their synsets involves calling the NLTK inbuilt path based similarity function for every pair of words corresponding to both the words’ synsets, we need a total of O(m*n) calls to the inbuilt path-based similarity function.