Report: Language Model (COM6516)

Part 1: Using the GUI

1. **Launch the application:**
   - javac *.java
   - java LanguageModel.java
2. **Input:**
   - Select a '.txt' file by clicking the "Select a file" button on the top of the frame. It will automatically process the file and all the text inside.
   - Input the text by clicking on the right-most text area and type the text you want to process.
   - Input text words to calculate a likely sequence of words.
3. **Processing the document:**
   - To process the user inputted document and display the vocabulary list, click on the "Process text" button.
   - Click on the "Collect Bigrams", "Collect Trigrams" buttons to display the bigram and trigrams fetched from the given text/document.
   - Use the "Show Stats", "Show NGram stats" to view the text/N-Gram statistics.
   - Click the "Calculate likely sequence" button to calculate the likely sequence (using unigram, bigram, and trigram) of the input text in the text area on the left side of the button.
   - Click the "Calculate likely U&B" button to calculate the likely sequence (using unigram and bigram only) of the input text in the text area on the left side of the button.
4. **Viewing result:**
   - The first table structure represents the vocabulary list with word and frequency count of that word.
   - For the Bigram and Trigrams, there are two tables with the corresponding words and their frequency of occurrence indicated.
   - Stats are shown in a popup dialog.
   - Likely sequence is shown in a popup dialog after clicking the "Calculate likely" or "Calculate likely U&B" button.

Part 2: Java Constructs

1. **Data Abstraction:**
   - The linked list's nodes which are used as word representations in the document is an abstraction provided by the 'MyLinkedObject' class. It includes information like word count and reference to the next node.

2. **Encapsulation:**
   - The word, count, next object are encapsulated inside the 'MyLinkedObject' class. The getWord(), getCount() and getNext() getter functions allow access to these fields.

3. Inheritance:

- An abstract superclass 'MyHashFunction' offers a standard interface for hash functions. The 'FirstLetterHF' and 'RollingHash' classes implements certain hash algorithms and is derived from 'MyHashFunction'.

4. Polymorphism:

- The abstract 'hash' method declared by 'MyHashFunction' is implemented differently in the subclasses 'FirstLetterHF' and 'RollingHash'.
- The MyLinkedObject's 'setWord()' method uses word comparison and recursion to handle various instances, demonstrating polymorphism.

5. Use of Collections Framework:

- To store bigram and trigram counts, the code makes use of the collections like 'HashMap'. The word frequency data is managed using the 'List' and 'Set' interface.

6. GUI:

- The graphical user interface is represented by the 'LM_GUI' class. To manage user input, there are buttons and action listeners.

7. Exception Handling:

- File reading and possible user interface interactions make use of exception handling.

Part 3: Code description

1. Brief description of each method in the 'MyLinkedObject' class:

- Constructor MyLinkedObject(String w): This constructor initializes a new 'MyLinkedObject' object along with setting the word passed in the parameter as 'w', sets the count to 1 and next reference to 'null'.
- setWord(String w): Sets the word field. If the input word passed as an argument 'w' is equal to the current word, it increments the count. If the next object does not exist, or if the current word is alphabetically smaller than the next word, it inserts a new object for 'w' between this and the next object, Otherwise, recursively passes on 'w' to the next object.
- getListLength(): Returns the total count of nodes in the list. Uses while loop iteration to count the number of nodes in the list.
- getWord(): Returns the word associated with the current linked object.
- getCount(): Returns the count associated with the current linked object.
- getNext(): Returns the next linked object in the list.
- getObject(): Displays the word and its count. Uses recursion to call the next linked object.

2. Choice of hash function algorithm:

- FirstLetterHF (First Letter Hash Function):

    - This hash function takes the Unicode value of the first letter of a word sequence and calculates the remainder of its division by the hash table size ‘m’. The formula used is ‘hash = Unicode_of_first_letter % m’.
- RollingHash:
    - The Rolling hash algorithm is used for more complex hashing based on the entire word sequence.
    - It employs a polynomial rolling algorithm to compute the hash value for a word.
    - Formula used: ‘hash = (hash * p + character) % m’
    - Here, ‘p’ is a prime number and when each character is processed one at a time, the hash value is updated continually.
    - Preventing collisions and encouraging a more uniform distribution of hash indices are two benefits of this approach.
    - Reference: https://www.geeksforgeeks.org/introduction-to-rolling-hash-data-structures-and-algorithms/

3. Hiding of the ‘MyLinkedObject’ and ‘MyHashFunction’ classes within the ‘MyHashTable’ class:

- The provided code does not encapsulate the ‘MyLinkedObject’ and ‘MyHashFunction’ classes within the ‘MyHashTable’ class.
- Each of these classes is declared separately in its own file, making them individually accessible from outside the ‘MyHashTable’ class.
- However, this can be achieved to encapsulation and enhance the overall organization and structure of the code.

4. Strategy for Rearranging Vocabulary list:

- The code implements sorting the list based on the alphabetical order or frequency.
- The list is sorted in alphabetical order if the variable ‘sortByFrequency’ is false, else, the list is sorted by the frequency of words.
- ‘Collections.sort’ is used to sort the list which takes a comparator which defines our custom logic of sorting. It compares two objects ‘a’ and ‘b’ based on their frequency counts and word’s alphabetical order if the counts are equal.
- Inside the comparator, ‘Integer.compare(b.getCount(), a.getCount()) compares the counts in descending order. If the counts are equal (‘countComparison == 0’) it falls back to comparing the words alphabetically by using ‘a.getWord().compareTo(b.getWord())’.

5. Observations made from vocabulary list:

- It provides a clear view of the words and their frequencies.
- Provides the insights on the stats such as most frequently used words and distribution of word frequencies.

6. Comparison between hashing algorithms:

1. **First Letter Hash Function**:
    - Easy to use and efficient for smaller datasets or situations where the distribution of the initial letter is enough.
    - May cause clustering if there are a lot of words that begin with the same letter.
2. **Rolling Hash Algorithm:**
    - Stronger and more complex for longer word sequences.

- Minimizes the impact of clustering and encourages a more uniform distribution of hash table indices.
- Ideal for managing extensive vocabularies and a variety of datasets.

It has been observed that when using the first letter hash algorithm, significantly fewer linked lists are produced, which leads to the formation of very long lists. In contrast, the rolling hash method builds linked lists that are shorter in length by using the entire hash table capacity.

The choice of the hash function depends on the characteristics of the dataset. 'FirstLetterHF' may be useful for datasets where words are evenly distributed alphabetically.

For a dataset with varying word lengths and patterns, 'RollingHash' is better at distributing and reducing clustering.

7. Sorting Linked lists by frequency:

- **Benefits of sorting by Frequency:**
  - Provides information on the words that appear most frequently in the dataset.
  - Easier to identify important terms for analysis.
  - Helpful when word frequency is an important measure.
  - Enhances efficiency in searching and retrieval.

8. Obtaining bigram and trigrams.

- **How bigrams are obtained:**
  - The input text is split into an array of words using the regex '\s+' (used for matching one or more whitespaces).
  - The words in the array are iterated over in a loop up to the second-to-last word.
  - For each iteration, a bigram is formed by joining the current word and the following word, with one space gap between them.
  - Only those words which does not contain Unicode symbol, full stop or an apostrophe or a lowercase English letter are processed as bigrams.
- **How Trigrams are obtained:**
  - Like how bigrams are calculated, an array of words is created from the input text.
  - Here, a loop iterates across up to the third-to-last word in the array.
  - Every iteration result in a trigram that is formed by joining the current word with the next two words, spaced apart.

9. Calculating most likely sequence:

- **Using unigrams, bigrams:**
  - For 'you have', it starts with finding unigrams and bigrams starting with 'you have' and chooses the next word based on the unigram/bigram probabilities.
  - The same approach for 'this is one'.
- **Using unigrams, bigrams, and trigrams:**
  - For 'you have', it starts with finding unigram, bigram, and trigram starting with 'you have' and chooses the next word based on trigram probabilities.

    - The same approach for ‘this is one’.
- **Calculating probabilities for unigrams, bigrams, and trigrams:**
    - The method ‘calculateUnigramProbabilities()’, ‘calculateBigramProbabilities()’ and ‘calculateTrigramProbabilities()’ calculates the probabilities in the given dataset based on the counts of unigrams, bigrams and trigrams.
    - It iterates through the unigramCounts/bigramCounts/trigramCounts which contains all the respective.
    - After getting an entry the from the ‘n-grams’, it is split into an array (for bigram/trigram). Checks if the entry is valid bigram/trigram.
    - Calculates the probability by dividing its count by the count of the first word (in unigram).
    - Stores the probabilities in the map and returns the result.
- **Getting the next word based on ‘n-grams’:**
    - For unigram: Checks if the provided unigram exists in the probabilities map. If it exists, ‘chooseRandomWord()’ is called, which uses the provided probabilities to choose the next word.
    - For bigram/trigram: Calculates the bigram/trigram probabilities using ‘calculateBigramProbabilites()’/’calculateTrigramProbabilites()’.
    - Calls ‘chooseRandomWord()’ to select the next word based on the updated bigram/trigram probabilities. Returns the selected next word.
- **Getting random word:**
    - The ‘chooseRandomWord()’ checks if the probabilities map is empty, returns null if empty.
    - Converts the words of the probabilities map to a list, generates a random index within the range of the list size and returns the word at the randomly chosen index.
- **Calculating likely sequence:**
    - ‘calculateLikelySequence()’ generates the likely sequence based on the input word using unigram, bigram and trigram probabilities. It starts with the provided input data, then predicts, and appends subsequent words to the sequence until ‘20‘words count.
    - ‘calculateLikelySequenceUB()’ generates the likely sequence based on the input word using unigram and bigram only. It starts with the provided input data, then predicts, and appends subsequent words to the sequence until ‘20‘words count.

10. Handling three-word sequences:

- Finding likely sequences for ‘today in sheffield’ would involve using trigrams. The trigram probabilities $p(wk|wk-2,wk-1)$ help determine the next word.

11. Challenges with implementing n-grams with a larger value of n than ‘3’:

- Larger values of ‘n’ present difficulties when implementing because of the increased sparsity of the data.
- As ‘n’ grows, there are exponentially more distinct n-grams, making it harder to estimate probabilities. It increases computation complexity and processing.