Object Oriented Programming and Software Design
Parth Antal(ACP23PA)

Part I

To use the GUI in the Java application, I have written down some steps to ensure everything is initialized and displayed correctly. Here's a step-by-step guide based on the provided code snippets and project structure:
=> Step 1: Compiling the Java Classes Before running the application, please make sure all the Java classes are compiled. If you're using an Integrated Development Environment (IDE) like Eclipse or IntelliJ IDEA, the build process might be automatic. Otherwise, classes can be compiled manually using the javac command in the terminal.
=> Step 2: Run the Application Before starting the application, please install the news.txt file in the local system by running the main method in the MyLanguageModel class. The reason for that is if the location of the news.txt file changes, you only need to update the file path in one place where you pass the parameter, rather than searching through the code to replace every hardcoded instance. So, I implemented filePath param rather than hardcoding the actual filePath in different classes in the code. After that, the code is executed typically by clicking a run button in an IDE(Eclipse/IntelliJ) or using the java command in the terminal. Please see the expected popup windows that should appear whilst running the code.

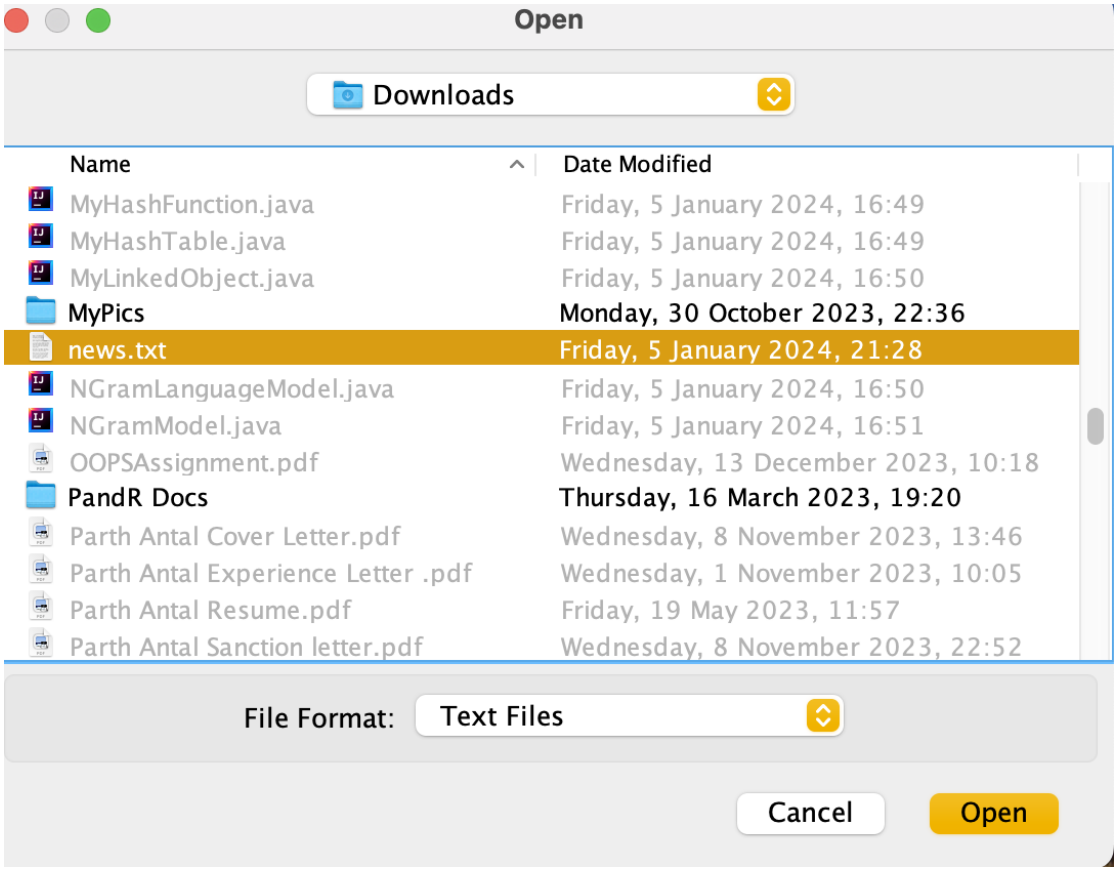
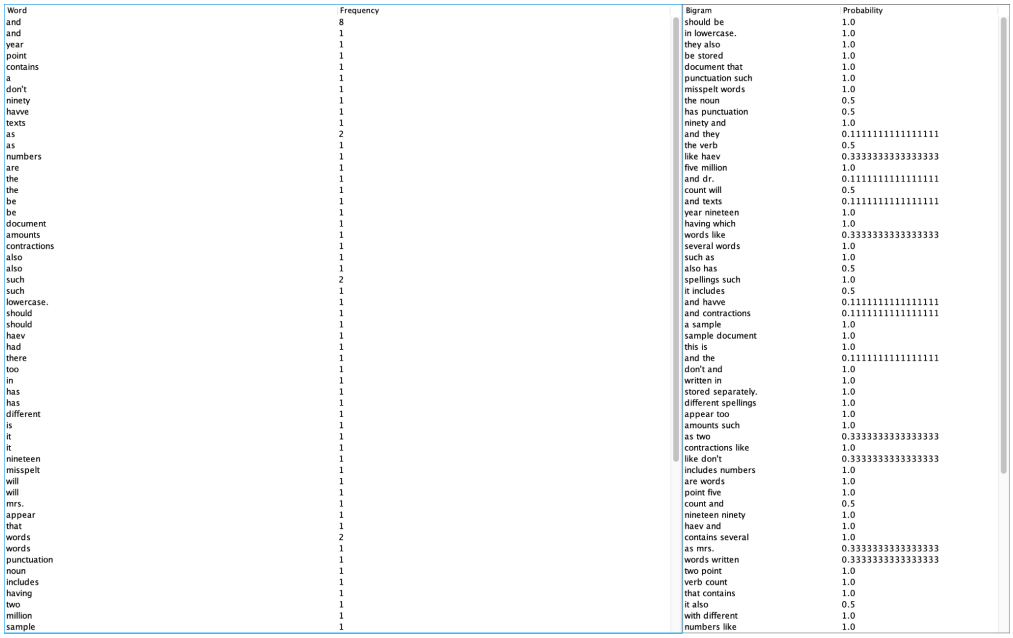


Fig1: News.txt select popup



Word	Frequency
and	8
and	1
year	1
point	1
contains	1
a	1
don't	1
ninety	1
havve	1
texts	1
as	2
as	1
numbers	1
are	1
the	1
the	1
be	1
be	1
document	1
amounts	1
contractions	1
also	1
also	1
such	2
such	1
lowercase.	1
should	1
should	1
haev	1
had	1
there	1
too	1
in	1
has	1
has	1
different	1
is	1
it	1
it	1
nineteen	1
misspelt	1
will	1
will	1
mrs.	1
appear	1
that	1
words	2
words	1
punctuation	1
noun	1
includes	1
having	1
two	1
million	1
sample	1

Bigram	Probability
should be	1.0
in lowercase.	1.0
they also	1.0
be stored	1.0
document that	1.0
punctuation such	1.0
misspelt words	1.0
the noun	0.5
has punctuation	0.5
ninety and	1.0
and they	0.1111111111111111
the verb	0.5
like haev	0.3333333333333333
five million	1.0
and dr.	0.1111111111111111
count will	0.5
and texts	0.1111111111111111
year nineteen	1.0
having which	1.0
words like	0.3333333333333333
several words	1.0
such as	1.0
also has	0.5
spellings such	1.0
it includes	0.5
and havve	0.1111111111111111
and contractions	0.1111111111111111
a sample	1.0
sample document	1.0
this is	1.0
and the	0.1111111111111111
don't and	1.0
written in	1.0
stored separately.	1.0
different spellings	1.0
appear too	1.0
amounts such	1.0
as two	0.3333333333333333
contractions like	1.0
like don't	0.3333333333333333
includes numbers	1.0
are words	1.0
point five	1.0
count and	0.5
nineteen ninety	1.0
haev and	1.0
contains several	1.0
as mrs.	0.3333333333333333
words written	0.3333333333333333
two point	1.0
verb count	1.0
that contains	1.0
it also	0.5
with different	1.0
numbers like	1.0

Fig2: GUI

Part II

The application's architecture illustrates a classic example of Java's object-oriented principles in action. Each n-gram type is encapsulated within its class embodying the concept of data encapsulation. This ensures that the internal representation of the n-gram data is hidden from the outer world, accessible only through well-defined interfaces. Inheritance is showcased in the GUI elements that extend from Java's Swing components, enabling the application to inherit a rich library of properties and methods for interactive elements. Polymorphism comes into play with the various button actions—despite the different functionalities each button provides, They are all managed by a unified event-handling mechanism. The Collections Framework is extensively used to manage the sets of n-grams. HashMaps are chosen for their efficiency in retrieval and insertion operations, which is vital for the performance of n-gram lookups and insertions. Exception handling is carefully implemented to catch and manage errors during file reading operations, ensuring the application's stability and providing user feedback for incorrect actions. Here's a detailed documentation for all the OOPS techniques that I implemented in my code:

1. **Data Abstraction**:
Data abstraction involves representing essential features without including implementation details. In our project, we define an `MyHashFunction` interface that declares a method `hash()` without specifying how the hashing is done. This allows us to abstract the concept of hashing and implement different algorithms as needed. Example:

```
public interface MyHashFunction{
 int hash(String word, int tableSize);  }
```

2. **Encapsulation:**
Encapsulation is achieved by bundling the data (variables) and code (methods) that operate on the data into a single unit, or class, and restricting access to some of the object's components. This is evident in our `MyLinkedObject` and `MyHashTable` classes, where we keep the `word`, `count`, and `next` fields private and provide public methods to manipulate these fields, like `setWord()`.

Example:

```
public class MyLinkedObject {
private String word;
private int count;
private MyLinkedObject next;  }
```

3. **Inheritance:**

Inheritance allows us to create new classes that are built upon existing classes. If our project had different types of hash tables, we could define a general `HashTable` class and extend it for various implementations. Hypothetical Example:

```
public class ExtendedHashTable extends MyHashTable {

   // Additional functionalities specific to ExtendedHashTable
}
```

4. **Polymorphism:**

Polymorphism in Java allows us to perform a single action in different ways. We use polymorphism in our hash function implementations. Each hash function can define its own `hash()` method, and the `MyHashTable` can use these functions interchangeably. Example:

```
public class SimpleHashFunction implements MyHashFunction {
 public int hash(String word, int tableSize) {
 // Implementation of hash
                                                    }
                                                                    }
```

5. **Use of the Collections Framework:**

The Collections Framework provides a unified architecture for representing and manipulating collections. We leverage it to manage collections of objects, such as lists of `MyLinkedObject` or maps for storing n-grams. Example: Map<String, Integer> unigrams = new HashMap<>();

6. **GUI Implementation:**

For the GUI, we employ Swing or JavaFX to build the user interface. This includes text fields for input, buttons, and areas for displaying the vocabulary list and n-gram statistics. Each component in the GUI is an object, encapsulating its state and behaviors, and interacts with other components through event handling. Example:

```
JFrame frame = new JFrame("Vocabulary Builder");
frame.setVisible(true);
```

7. **Exception Handling:**

Exception handling is crucial for robust applications. We implement try-catch blocks to gracefully handle potential errors, such as file I/O issues or parsing errors. Example:

```
try {
  processDocument(inputText);
     }
   catch (IOException e) {
      e.printStackTrace();
                              }
```

Part III

Here are the descriptions and discussion regarding my strategies and how I implemented the respective tasks:

1. **Description of Methods in `MyLinkedObject` Class**

-> Constructor: `MyLinkedObject(String w)`: Initializes a `MyLinkedObject` with the given word `w`. The `count` is set to 1, and `next` is set to `null`, indicating the end of the list.

-> setWord(String w): Inserts or updates a word in the linked list. If `w` is smaller than the current word or if `next` is `null`, a new object for `w` is created and linked. If `w` equals the current word, the `count` is incremented. Otherwise, the method is recursively called on the `next` object.

-> Getters `getWord()`, `getCount()`, and `getNext()`: Return the respective fields of the object.

->`incrementCount()`: Increments the `count` of the word.

-> `printList()`: Prints the entire linked list starting from the current node.

->`toString()`: Provides a string representation of the linked list starting from the current node.

2. **Choice of Hash Function Algorithm:**
For the hash function, a polynomial hash function is used. The hash value is computed as follows:

$$\text{hashVal} = \sum_{i=0}^{n-1} s[i] * a \text{ pow } (n-1-i) \bmod \text{tableSize}$$

where `s[i]` is the ith character of the string, `a` is a constant (31 in my implementation), and `n` is the length of the string. This method is a widely used approach in string hashing due to its ability to generate a wide range of hash values with minimal collisions. The source of this approach can be found in many computer science textbooks and online resources discussing hash functions.

3. **Hiding `MyLinkedObject` and `MyHashFunction` within `MyHashTable`:**
My implementation keeps `MyLinkedObject` and `MyHashFunction` as separate entities rather than encapsulating them within `MyHashTable`. This design choice can be justified as it allows more flexibility and reusability of the `MyLinkedObject` and `MyHashFunction` classes. However, encapsulating them within `MyHashTable` I agree could have provided better data hiding and a more compact API.

4. **Strategy for Rearranging Vocabulary Lists:**
For rearranging from alphabetical to frequency order, I used a sorting algorithm on the list of words. This would involve comparing the counts of words and sorting them in descending order. For words with equal frequencies, their alphabetical order can be the secondary sorting criterion.

=> Observations and Comparisons:

- Lengths of Individual Linked Lists: By comparing the lengths of linked lists in different buckets, I evaluated the efficiency of the hash function. A more even distribution indicates a better hash function.

- Choice of Hash Function: Comparing different hash functions would involve looking at how evenly they distribute words across the hash table and their impact on performance.

- Table Size (`m`) Variations: Using various values for `m` can show how the size of the hash table affects performance and collision rates.

5. **Handling Probabilities in N-Grams**
In n-gram models, probabilities of word sequences are calculated based on the occurrence of those sequences in the training data.
- For unigrams, the probability `p(w1)` is the frequency of `w1` divided by the total number of words.

- For bigrams, `p(w1, w2)` is calculated using `p(w2|w1)`, which is the frequency of the bigram `w1 w2` divided by the frequency of `w1`.

- For trigrams, `p(w1, w2, w3)` involves `p(w3|w1, w2)` and can be calculated similarly.
Implementing n-grams with larger `n` values increases complexity and the sparsity problem, where many n-grams may not appear in the training data, making probability estimation challenging.

Implementing n-grams with a value of "n" larger than 3 (trigrams) presents several challenges and issues, primarily related to data sparsity, computational complexity, and diminishing returns in terms of language modeling performance. Here are some key issues:

1. **Data Sparsity**: As "n" increases, the number of possible n-grams grows exponentially. Many n-grams that could theoretically occur are not observed in the training data, leading to data sparsity. This sparsity makes it difficult to accurately estimate the probabilities of these n-grams, as there may not be enough occurrences to make reliable statistical inferences.

2. **Increased Computational Resources**: Larger n-grams require significantly more memory and computational power to process and store. The amount of memory needed to store the n-gram counts and probabilities increases exponentially with "n". This can become impractical or infeasible, especially for large datasets or when computational resources are limited.

3. **Curse of Dimensionality**: With higher-order n-grams, the dimensionality of the data increases, exacerbating the issue of data sparsity. This phenomenon, known as the curse of dimensionality, leads to overfitting, where the model performs well on the training data but poorly on unseen data.

4. **Longer Contexts May Not Always Be Helpful**: In many natural language processing tasks, the immediate context (one or two words preceding) is often more relevant than a distant context. Therefore, increasing "n" beyond a certain point may not significantly improve the performance of the language model, as the additional context might not provide meaningful predictive power.

5. **Increased Complexity in Handling Unknowns**: Handling unknown words or unseen n-grams becomes more complex with larger "n". Smoothing techniques, which adjust probability estimates to account for unseen n-grams, become more critical and complex to implement effectively.

To address these challenges, various techniques like back-off models, smoothing techniques, and neural language models have been developed. These methods aim to balance the trade-offs between capturing longer dependencies and dealing with the practical limitations of higher-order n-grams.
