Marie Cho
mhc2323
September 12, 2024
CS314H

Prog 2 Report

1 Overview

Goals

This lab was focused on developing code to generate a written text using Markov chains with a given level of analysis *k*. The goal was to take in a source input and produce an output with similar character proportions by leveraging the probabilities of characters in the text. Through the implementation of this project, I explored how to handle file I/O, model, probabilistic modeling, and text generation. By the end of the lab, I successfully created a program that could mimic the style and character distribution of the source text.

Insights

Throughout the implementation, one surprising aspect was the simplicity of the algorithm once the frequency map was generated. After initially constructing the HashMap to gather character patterns in the text, generating the output became a straightforward process of selecting random characters based on their recorded frequencies, leading to methods with a time complexity of *O(1)* or *O(n)*. Another insight was how the quality (closeness to English words) of the output text is highly dependent on the value of *k*: small values lead to incomprehensible text, while larger values produce more coherent outputs. This is due to larger values of *k* capturing longer sequences of context from the source text, allowing the generated text to better mimic word structures and sentence patterns.

2 Solution

Assumptions

(a) All valid files contain ASCII characters (no cryptic or hieroglyphic-like symbols).
(b) Input is case sensitive (‘A’ != ‘a’).
(c) Input text is not cleaned, such as removing white space and unnecessary symbols.
(d) No carriage returns exist in the file.

Design Considerations

When designing the solution for RandomWriter, I had to consider the data structures to use for each variable to optimize both performance and usability. The variables *content*, *outputText*, and *frequencyMap* were declared as private class variables to ensure accessibility across all methods within the RandomWriter class. I used a StringBuffer for handling text manipulation (*content* and *outputText*), as it offers efficient string appending, which is critical when working with large datasets. For *frequencyMap*, I implemented a 2D HashMap to store character patterns and their respective frequencies. This choice allowed for fast lookups and updates without increasing time complexity, which is vital when processing large input files.

Additionally, I incorporated exception handling to properly manage potential failures and edge cases, including insufficient arguments, null cases, and invalid file names/types. By implementing error messaging and try-catch blocks, the program can smoothly handle these issues without crashing unexpectedly.

Finally, I created more helper and sub-functions than initially given in the lab to not only increase code readability but also make it easier to test or add features to individual components. This modularity ensures that the codebase is both scalable and maintainable for future extensions or improvements.

Algorithms and Implementation

Each effect was designed with edge cases in mind and was addressed in *main()* or in other specific methods, such as *checkOutput()*. Unexpected or uncommon scenarios, such as insufficient or invalid arguments (for any method) were handled through try-catch blocks, printing out an error statement, and throwing an appropriate exception. Implementations for each effect are described below:

(a) **main():** Handle potential invalid arguments, initialize the RandomWriter instance, and call the necessary methods for reading input, processing text, and generating output (listed below). Checks to ensure that *args* of length at least 4 are passed in, where all arguments are non-null, *args[2]* is less than the length of the input text, and *args[2]* and *args[3]* are nonzero integers. If so, proceed to check the validity of the output file and write the generated text inside it; if not, throw an *IllegalArgumentException*.

(b) **readText():** Read the entire text of an inputted file into StringBuffer *content*. Check whether the file exists and is valid, throwing an *IllegalArgumentException* if the file is unreadable. If no exception is thrown, read each character from the file and append it to *content*.

(c) **generateText():** Start with a random *k*-length substring, which becomes the initial *seed*, and iteratively appends the next characters to *outputText* based on *frequencyMap*. Use randomization and the probabilities to determine which character, if more than one per *seed*, gets appended. Reassign *seed* by removing its first character and appending the recently chosen character. If *seed* has no valid next character (is not a key in the map), reset to another random seed, and continue the process until the desired *length* is reached.

(d) **checkOutput():** Validate that the output file can be created or written to. If so, create the new file; if not, throw an *IllegalArgumentException*.

(e) **writeText():** Write the generated text to an output file *outputFilename*. Since *checkOutput()* is called before this method, write *outputText* onto the new file without checking its validity.

(f) **createFrequency():** Initiate *frequencyMap* with a *seed* and a corresponding HashMap with the next character and its frequency. Iterate over the input text and determine each *seed*. If *seed* does not already exist as a key in the map, add a new key-value pair. If *seed* does exist, add 1 to its respective letter value. Continue to map each substring of length *k* to the characters that follow it as well as the number of times they appear in this sequence, leaving 1 character at the end so that all determined *seed*s have a next character.

(g) **getRandomChar():** Sum up how many times a next character is found for the current *seed* and generate a random integer from 0 to this sum. Iterate through the *seed*'s values until the character corresponding to the random integer is found to match the frequency distribution. Return this character.

(h) **kEqualToZero():** Generate a random sequence of characters from *content* when the analysis level *k* is equal to zero, as there is no seed to start from/retrieve the next character.

3 Further Discussion

Scope

This project has real-world applications in fields such as natural language processing or text simulations. These methods might be useful in attempting to complete an unfinished written work, generating predictive text to mimic style, or providing text suggestions as seen in applications such as Google Docs and iMessage. By reading and analyzing a singular text file, this program was able to generate a new text that closely resembled the input. Therefore, with access to a mass number of text files, a more advanced version of this program might be able to make smarter or more accurate decisions and be applicable to real-world, complex scenarios in text generation and automated writing.

Yet, the program is limited by its scalability and model simplicity. While it handles small to medium-sized inputs efficiently, much larger, expansive datasets and higher values of *k* will lead to an exponentially growing frequency map, which eventually becomes too computationally expensive to manage. Handling larger volumes of data, as might be required in real-world corporate environments, would need more sophisticated algorithms and data structures. The current model is quite basic, as it selects the next character solely based on immediate patterns in the input text. In contrast, more advanced models, such as those based on neural networks, can produce far more accurate and meaningful results by considering a broader context and understanding of the text. These models can generate coherent, contextually-sensible text suggestions, but developing such models is beyond the scope of this project. Exploring this further in future work could lead to significantly improved results and applicability.

Solution Quality

The implemented methods succeeded in the following:

(a) Methods generally output maintains coherent structure and has similar character resemblance to the input, increasingly so as *k* increases. However, for small values of *k*, the output becomes too random and incoherent to be considered proper English.
(b) Methods were able to handle edge/unique cases, including text length of 0 and invalid arguments.
(c) Methods performed within a reasonable time frame for small to medium inputs.
(d) Code is clean, simple, and implements encapsulation for flexible changes in the future.

Interesting Results

Shown below are the before and after texts for the input file *pangram.txt*, which contains the text "the quick brown fox jumps over the lazy dog" and the output file *output.txt* for varying values of *k* and *length* = 100:

pangram.txt

```
the quick brown fox jumps over the lazy dog
```

With *k* = 0:

output.txt

```
uhptk  f zbbe  x viefgrebt gjrzdldonphd de hbe reeve tahemj  fyr n  dhw scuk hkyeyccertry
iofovt
```

With *k* = 1:

output.txt

```
ick the jumps quick dogerover fogzy down therogumps brown br brown the br jumps juick
foverox own la
```

With *k* = 5:

output.txt

```
e quick brown fox jumps over the lazy dog the lazy dogx jumps over the lazy dogown fox jumps
over th
```

With *k* = 10:

```
fox jumps over the lazy dogy doggjumps over the lazy dogox jumps over the lazy dogown fox
jumps over
```

For smaller values of *k*, starting with *k* = 0, the generated text is almost entirely random, with no recognizable structure or coherence. The characters are chosen independently, leading to an output that doesn't resemble English at all. As *k* increases, the generated text begins to exhibit more recognizable patterns. For example, with *k* = 5, the output starts to include familiar phrases from the input, although there is still some randomness. By the time *k* = 10, the output text is almost identical to the input (disregarding the difference in length) which reflects entire phrases and sentence structures.

Problems Encountered

One of the primary issues encountered during this project was related to improper argument handling and file I/O operations. Initially, my program would fail without any indication of error when provided with invalid file paths or non-existent files. In such cases, it wouldn't create or write in the output file nor would it display any error messages. To address these problems, I implemented thrown exceptions and error messages to ensure clear feedback when a file couldn't be accessed or created. This also improved my debugging efficiency.

Additionally, I had to keep thinking of new instances of edge cases, such as empty files, insufficient arguments, invalid arguments, and input files that were too short for the given value of k. Without proper handling, these edge cases could cause the program to behave unpredictably or produce poor-quality results. Therefore, I continually added validation checks and threw exceptions when appropriate.

Finally, I had to figure out a way to arrange my code so that white box testing would work as intended. For example, in my *writeText()* method, I originally had the check for the output file validation and creation in addition to actually printing the generated text into the file. This led to issues during testing, as although my intention was to test that the program would throw an appropriate error if an invalid file couldn't be created or accessed, the combined logic would always proceed without errors. To resolve this, I split the functionality into separate methods: *checkOutput()* to handle file validation and creation, and *writeText()* to focus solely on writing content to the file. By isolating the file validation logic, I could more accurately test scenarios where file access failed as well as improve code clarity.

4 Testing

Test Procedure
Using JUnit, I implemented methods to practice white and black box testing on the methods that have deterministic outputs. For example, *generateText()* and *getRandomChar()* are almost impossible to test because they create their output based on probabilistic randomization, making it difficult to validate since the result changes with each run.

For other deterministic methods, the testing procedure involved verifying each function with both valid and invalid inputs. For example, tests ensured that *readText()* correctly handled nonexistent or invalid files and that *main()* responded appropriately to unexpected arguments. Additionally, *createFrequency()* was tested by ensuring that it accurately tracked character frequencies for given *seed*s, allowing for reliable text generation.

Test Cases and Rationale

(a) **testCreateProcessor():** Ensure that *main()* can be called without any errors when given valid arguments. This test is a standard run where the processor is created and runs successfully.

(b) **testGetContent()/testContentRead():** Check that the content read from the input file "simple.txt" (“simple test”) is correctly read and stored in *content*.

(c) **testLessArgs():** Verify that if fewer than the required number (4) of arguments are passed, an *IllegalArgumentException* is thrown.

(d) **testInvalidThirdArgument()/testInvalidFourthArgument():** Test that passing a non-numeric value as the third or fourth argument (*k* or *length*) throws an *IllegalArgumentException*.

(e) **testFourArgs()/testMoreArgs():** Verify that *main()* executes without throwing exceptions when four or more valid arguments are passed in. This is similar to *testCreateProcessor()* by checking for normal execution.

(f) **testKEqualZero()/testLengthEqualZero():** Verify that the program can handle an analysis level of 0 or a text generation length of 0 without throwing any exceptions. This tests the edge cases of generating an output with no seeds or empty *outputText*.

(g) **testKGreaterThanLength()/testKGreaterThanLength():** If *k* is less than *length*, methods should proceed as normal. Otherwise, verify that an *IllegalArgumentException* is thrown. This tests the edge cases of attempting to generate a text using a seed that is longer than *content* itself.

(h) **testNullArgs():** Check that passing null as one or more of the arguments throws a *NullPointerException*.

(i) **testBlankStrings():** Ensure that passing blank strings as arguments results in an *IllegalArgumentException*.

(j) **testNegK()/testNegLength()/testInvalidK()/testInvalidLength():** Verify that passing a negative value or a *k* that is equal to or exceeds the length of the text results in an *IllegalArgumentException*.

(k) **testFileNotFound()/testFileNotValid()/testFileFoundAndValid():** Ensure that attempting to read from a nonexistent file or invalid file type (ex: .jpg) results in an *IllegalArgumentException*. On the other hand, ensure that a valid file can be read without throwing any exceptions.

(l) **testOutputFileValid():** Ensure that checking and creating an output file (even if it doesn't already exist) does not throw any exceptions.

(m) **testWriteText():** Test that the writing process can be completed without throwing any exceptions for a valid output file.

(n) **testCreateFrequency()/testCreateFrequency2()**: Verify that *createFrequency()* correctly builds the frequency map from a given input string by comparing a hardcoded expected frequency map with the actual frequency map built in the class.

In total, 25 test cases were implemented using varying parameters.

5 References and Citations

(a) I participated in high-level discussions of understanding the spec and algorithmic concepts with Sneha Shekhar.