Copy Models for Data Compression

Organization:	Universidade de Aveiro
Course:	TAI - Teoria Algorítmica da Informação
Date:	Aveiro, Portugal, 29/03/2023
Students:	Afonso Campos, 100055 Daniela Dias, 98039 Hugo Gonçalves, 98497
Project abstract:	Copy Model for Data Compression. Text Generation using the model.

Index

Introduction

A copy model is a method for data compression that views data sources as being produced of previously-seen, replicating parts and that each new symbol will have already occurred in the past.

For this to work a copy model maintains copy pointers to past symbols in order to estimate how likely it is that the new outcomes are the same as the currently predicted symbol, given to us by using the pointer.

This estimation is done by progressively counting hits and misses. Hits represent the number of times we were right and our prediction matched reality, misses represent the contrary. In order to avoid assigning null probabilities to some symbols a copy model can't simply rely on relative frequency and must introduce a smoothing parameter alpha (α) on the probability calculation:

$$P(hit) = \frac{\#Hits + \alpha}{\#Hits + \#Misses + 2\alpha}$$

With the model and the symbol probabilities computed we can then estimate the amount of information taken up by the newly read symbol (should it be correctly predicted, or not) with the equation:

$$-\ log_2(P(character))$$

As such, we were tasked with studying the behavior of a copy model by estimating and analysing its compression metrics for various hyperparameters as well as using copy model logic to create a prompt-based text generator.

Methodology

Our project is composed of two main components, the **Copy Model** (a.k.a. CPM) and the **Generator** module. The Copy Model implements code that receives a file as input and calculates an estimative of the total amount of information in that file using the Growing windows (explained more in detail in the next section). The Generator uses a different type of model (a fixed and sliding window) that is run over an input file and is used to, given an input string, generate X more characters of text using the text that was seen in the past.

In the next sections, we will explain in more detail the methodology adopted to develop each one of these modules. In the next Chapter (*Implementation*) we will also explain in detail how we coded the solution.

Copy Model

As stated above, our Copy Model tries to calculate the total amount of information by taking into account past occurrences of a sequence of text. Our implementation of the copy model is located in "`src/cpm/builder`".

There are a lot of customizable options in our Copy Model implementation. Parameters such as Alpha, K, Threshold, the input file, etc. can be defined by the user with resort to input arguments. In **Table 1** we can see a list of all input arguments and a short description for each one of them.

Name	Description	Flag	Options
Alpha	Used to smooth Hit Probability	-a	]0, 3]
Window Size	Set size of the window (k characters)	-k	Integer
Threshold	The limit to stop Copy Model	-t	]0, 0.5]
Input File	The Input file that will be analyzed.	-i	String
Output File	The file to which the Generator model will be output	-o	String
Serializer Type	The Type of Model to use for the Generator	-s	0 - Positional 1 - Probabilistic
No Information	Only run the model for generator	-nl	None
Logging Level	Set the Logging Level (Info by default)	-l	1 - Info 2 - Debug

No Serialization	Only run the Copy Model	-nS	None

Table 1: Input arguments for Copy model

Our Copy Model will read the input file window by window. Each window is a sequence of K characters. When a window is read, it is added to a data structure that "reminds" the sequence read and the position where that sequence is. Every time we read a sequence, whether it is a new one (never appeared in the file before) or it is a known one (we already saw it) the Copy Model will store the position for that sequence in that data structure.

After we "saved" the sequence (a.k.a. window), we will check if our current window already appeared in the file, if it did, then we will try to predict the future, based on the past. This means that if the sequence of K symbols we are in at this moment is also in the same file in a previous position, we will assume that the symbols after that sequence may be similar to the symbols that will become after our current window with a certain probability - the primary principle of the Copy Model. If we can calculate this probability of the next symbol in our current window being the same as the one that appeared in the past then we can determine the amount of information for that symbol at that point in the file using the work done by Shannon C. in 1948. By summing up the amount of information for all characters taking into account the past and the probability of that character being the correct one at this point, we can have a good estimate for the total amount of information in the file we are reading at this moment.

Therefore, for each window, we are currently analyzing, and if we saw that sequence in the past, we will calculate the probability of the next character in the sequence to be the correct one (according to the past). To calculate this probability (**Hit Probability**) we use the formula presented in the Introduction and take into account the number of hits and misses for that sequence. For the first character after the K sequence of characters, the probability will be equal to 0.5 (because we have 0 hits and 0 misses), but as we keep expanding our current window to the next characters and keep comparing them to the characters observed in the past sequence, we will increment the number of hits and misses and this probability will variate. A **Hit**, in this context, is when the current character is equal to the one in the past sequence, and a **Miss** is the opposite.

If we had a Hit, we will increment the number of hits for our current sequence and we will increment the total amount of information with the -log2 of the Hit Probability because we had a Hit. On the other hand, if we had a Miss, we will increment our total amount of information with a **uniform distribution** of the complementary probability (1 - Hit Probability / Number of Symbols in Alphabet-1).

As you may notice, we would keep following this logic forever as we don't have anything that would stop us from keeping expanding the first window that we have seen in the past forever and keep computing probabilities and amounts of information. This would not, however, be a great implementation of a Copy Model. Therefore, we need to have a stop

condition, a **threshold**, that will indicate to us when we should “turn off” the current Copy Model and start a new one, i.e., when we should stop expanding the current window and start expanding another one instead. This threshold, in our implementation, is a decimal value that is compared against the Hit Probability. If the Hit Probability gets below that threshold, then we will “give up” on our current sequence and try to create another Copy (i.e., expand another window). We tested several values for the threshold and realized that some provide better results compared to others (see Results for more information).

We will also want to not only get the total amount of information in our file but also get the average amount of information per symbol in our file. To do this we simply divide the total amount of information by the total amount of characters in our file as we already calculated the total amount of characters and the alphabet in our file in our **First Pass** (refer to Implementation chapter for more details).

In order to speed up our Generator and avoid unnecessary work when we run the Generator multiple times for the same input file (i.e., same file as source), the Generator **does not** run this model. Instead, the model (very different from the Copy Model) is run by the CPM module and serialized into a text file. This text file is then loaded by the Generator and used to generate text.

This model that is used by the Generator is very different from the Copy Model we implemented. Instead of trying to expand the current window based on the Hit Probability and using that probability to estimate the total amount of information, this model will always keep the same window size (K) and will slide the window at each step. This allows us to know, for each sequence, which character comes next.

We read the input file and keep sliding the window of size K. We keep track of all the positions (for the Positional Model) and all the characters and their counts that show after the window (for the Probabilistic Model) for each and single window of size K we “saw” in our file. This will allow this model to know, in one way or the other, which characters can appear in front of a certain sequence by taking into account the input file.

Generator

Our Generator will try to generate text taking as input a sequence of characters provided by the user. To do this, it needs to have information, i.e., a model that can determine, based on some text of a file that was analyzed in the past, which characters are the ones more probable or which ones appeared indeed after some sequence of size K. Our Generator also has a list of input arguments that can be used to customize its behavior. In **Table 2** there is a list of all the options available as well as a small description for each one of those options.

Name	Description	Flag	Options
Model Path	The path for the serialized model	-m	String
Generation Size	The number of characters to generate	-g	Positive Integer
Interactive Mode	Use interactive mode (Non-Interactive by default)	-i	None
Logging Level	Set the Logging Level (Info by default)	-l	1 - Info 2 - Debug

Table 2: Input arguments for Generator

As already stated before the Generator does not run this “past-based model”, instead it only loads the model from a file where it was serialized. Also as already said before, the Generator can use 2 different types of models. A **Probabilistic** model and a **Positional** model. The first one, stores, for each sequence of K characters (defined by the User) the characters that may come next (based on the input file) and the probability of those characters coming next. The second model - the positional model - stores, for each K characters sequence - a list of pointers where that sequence was seen in the original file.

The Generator’s basic behavior is the same for both models. It takes the input provided by the user and reads the last K characters. Next, it will try to find the last K characters in the model; if it can find them, it will complete the text with the next character, otherwise, it will stop the generation of text. If it was able to find a character, it will move the window, i.e., it will update the last K sequence of characters (adding the one it just generated) and repeat the process. Although this behavior is followed independently of the model we are using there are some specificities that depend on the type of model (the Generator can automatically realize which model to use and adapt itself - see Implementation for more details).

If we are using the **Probabilistic** model, the generator will look for the K sequence of characters in the model, get all the characters that can come after that sequence as well as their probabilities and generate, randomly, one of those characters with basis on their relative probabilities. On the other hand, if we are using the **Positional** model, the generator will look for the K sequence and obtain a list of all the “pointers”, i.e., places where that sequence is in the original file. Next, it will choose randomly (with a uniform distribution) one of those pointers and complete the user input with the character that comes after the pointer it just chose. Each one of these models has different specificities as we will analyze in more detail in the Results section.

If we are using the Positional model we need to have fast and **random access** to the input file contents. One way to achieve this is to read the entire file to memory inside the generator before starting the next generation. Then, when we need to access a position on the file to know which character comes after that position, we will simply do random access in memory which is very fast. However, this has a drawback. If our input file is very large, we will

consume a lot of memory and resources. Therefore, to mitigate this, in our Generator implementation, we are not reading the file contents to memory. Instead, we are using `fseek()` to randomly access the contents in the file directly without loading it to memory. With this approach, we may lose some performance, as we are reading from this, but we save a huge amount of memory (more details in the *Implementation* chapter).

Our Generator also implements an **Interactive** and **Non-Interactive** modes. With the interactive mode, it will ask for the user input, generate the text, and when it is finished it will ask for another input. In the Non-Interactive mode, it will also ask for input but will only provide the result once. Besides that, the **Interactive** mode also prints the characters immediately after they were generated giving it a "typing" animation and a sensation of progress in the generation, while the **Non-Interactive** mode only prints the text after the entire text was generated by the user.

Copy Model Implementation

Obtain Copy Model Parameters

The **CopyModelInputArguments** class is used to obtain the arguments from the command line and set the **Copy Model** parameters. The class has a method **parseArguments** that parses the arguments and stores them in the private member variables. It also has the method **checkArguments** that verifies if the parsed arguments are valid. If the arguments are not valid, the program exits. Finally, it has a method **printUsage**, which prints the usage of the program to the console.

Member Variables

This class sets the following Copy Model parameters:

- **alpha**: A double representing the "smoothing" parameter used in the model. It is initialized to 0.5 as a default value.
- **k**: An int representing the window size used in the model. It is initialized to 3 as a default value.
- **threshold**: A double representing the threshold used in the model (to reposition the pointer associated to the Copy Model). It is initialized to 0.4 as a default value.
- **inputFilePath**: A string representing the file path of the input data for the model. It is initialized to an empty string.
- **outputModelPath**: A string representing the file path where the output model will be saved. It is initialized to an empty string.
- **loggingLevel**: An int representing the level of logging that the program should perform. It is initialized to 1 as a default value.
- **serializerType**: An int representing the type of serializer to use when saving the model. It will be set by the program.
- **serializeForGenerator**: A bool representing whether the model should be serialized for a generator. It is initialized to true as a default value.
- **calculateInformation**: A bool representing whether additional information should be calculated by the program. It is initialized to true as a default value.

Parse Arguments

The **parseArguments** function is a member function of the **CopyModelInputArguments** class**,** which takes two input arguments: **argc** and **argv.** The function is responsible for parsing the input arguments and updating the corresponding member variables of the class based on the provided command-line arguments.

First, the function checks whether any arguments were entered. If none were entered (**argc** is equal to 1, which means no arguments were passed except for the name of the program itself), it prints an error message, calls the **printUsage()** function, and exits with a failure status. Next, the function initializes some temporary variables that will be used to store the parsed values from the command-line arguments.

The function then iterates through the command-line arguments using a for loop. It checks each argument against a set of predefined strings (such as -h or --help) to handle the arguments that do not have a value associated with them. If a matching argument is found, the function sets the corresponding member variable of the class to the appropriate value.

The function then iterates through the command-line arguments again, this time checking for arguments that do have a value associated with them.

For arguments with flags such as **-a** (--alpha), **-k**, **-m** (--model), and **-t** (--threshold), the value must be a valid number. In this case, the function checks if the next argument in **argv** is a number using the **isNumber()** function and parses the value if it is a number. If the value is successfully parsed, the corresponding temporary variable is updated. Meanwhile, for flags **-i** (--input) and **-o** (--output), the value stays as a string.

After all arguments have been parsed, the function checks whether the parsed values for **alpha**, **k**, and **threshold** are non-zero. If any of these values are zero, the function prints a message indicating that a default value will be used instead and sets the corresponding member variable of the class to the default value. Finally, the values for all of the parameters are assigned to the class member variables.

Check Arguments

The purpose of the function **CheckArguments** is to check the validity of the input arguments that are passed to it. The function first checks if the arguments are valid by checking if **calculateInformation** and **serializeForGenerator** are true. If both are false, it prints an error message and returns false. If at least one of them is true, the function continues to check the remaining arguments.

The next check is to see if **inputFilePath** is empty. If it is, it prints an error message and returns false. Next, we see if the value of **k** is between 0 and 100, inclusive. If it's not within that range, it prints an error message and returns false.

If **calculateInformation** is true, the function checks the validity of the **alpha** and **threshold** values. These values should be between 0 and 5 and 0 and 1, respectively. If they are not within the valid ranges, the function prints an error message and returns false.

If **serializeForGenerator** is true, the function checks the validity of the **serializerType** value. It should be either 0 or 1. If it's not 0 or 1, the function prints an error message and

returns false. Additionally, if **outputModelPath** is empty, the function prints an error message and returns false. If all the checks are passed, the function returns true, indicating that all the input arguments are valid.

Logger

A **Logger** class can be used to print log messages with different levels of severity (debug, info, and error) depending on the set logging level.

This class has a private member variable, **loggingLevel**, which represents the current logging level. The constructor initializes this variable to **LoggingLevel::INFO**. The setLevel function allows the logging level to be changed.
This allows the logging level to be set dynamically at runtime based on the user's input.

The class also has three public functions: **debug()**, **info()**, and **error()**. Each of these functions takes a string message as input and prints it to the console if the current logging level is equal to or greater than the level associated with the function. debug is the lowest level, info is the default level, and error is the highest level.

For example, if the logging level is set to **LoggingLevel::INFO**, the debug function will not print anything, but the info and error functions will print messages to the console. If the logging level is set to **LoggingLevel::ERROR**, only the error function will print messages.

File Reader

In order to implement the Copy Model, we need to be able to read the input file character by character and maintain a sliding window over the characters in the same file. The **FileReader** class provides this functionality.

The constructor of the **FileReader** class takes two arguments: a file path and a window size. The **filePath** argument specifies the path of the file that should be read, and the **windowSize** argument specifies the size of the sliding window that should be used.

The **FileReader** class contains several private member variables, including the file pointer **targetFile**, the current position in the file **currentPosition**, the current window of characters **currentWindow**, the previous window of characters lastWindow, the current sequence of characters **currentSequence**, and the next character in the sequence **nextCharacterInSequence**. The cache variable is a vector that keeps track of all the characters that have been read so far.

Member Variables

In more detail, **FileReader** contains the following:

- **filePath**: A string that stores the path to the file being read.
- **windowSize**: An integer that represents the size of the sliding window used by the file reader.
- **targetFile**: A pointer to the file being read.
- **currentPosition**: An integer that stores the current position in the file.
- **currentWindow**: A pointer to a character array that stores the current window of characters being read from the file.
- **lastWindow**: A pointer to a character array that stores the last window of characters read from the file.
- **currentSequence**: A vector of characters that stores the current sequence of characters being read from the file.
- **nextCharacterInSequence**: A character that stores the next character in the current sequence.
- **cache**: A vector of characters that stores a cache of characters read from the file.

Main Methods

The **openFile()** method of the **FileReader** class opens the file specified by the **filePath** and returns a file pointer to the opened file. The **closeFile()** function closes the file opened by the **openFile()** function, stored in the **targetFile** member variable. If the file is not open, it does nothing.

The **readInitialWindow()** function reads the first window from the file. It reads **windowSize** characters from the file and stores them in the **currentWindow** buffer. It also reads the next character using the **nextCharacter()** function.

The **shiftWindow()** function shifts the current window by one character and reads the next character using the **nextCharacter()** function. It returns true if a character was successfully read, and false if there are no more characters to read.

The **next()** method reads the next window of characters from the file. If this is the first window of characters to be read, it calls the **readInitialWindow()** method to read the initial window of characters. If this is not the first window of characters, it calls the **shiftWindow()** method to shift the current window by one character and read the next character in the sequence.

The **resetCurrentSequence()** function clears the currentSequence vector and initializes it with the characters in the **currentWindow** buffer.

The functions **shiftWindow**, **next** and **resetCurrentSequence** are only called when we reset the Copy Model.

First Pass

Get File Info

The **getFileInfo** function takes a **CopyModelInputArguments** object as an argument, which contains information about the input file to be processed, such as the input file path and the window size k.

The function first calls **getFileReaderInstance** to get an instance of the FileReader class, which is responsible for reading the input file. **getFileReaderInstance** creates and returns a **FileReader** object with the input file path and window size k passed as arguments.

After obtaining the **FileReader** instance, **getFileInfo** opens the input file using **fileReader.openFile()**. This method is used to open the target sequence file and it returns a pointer to the opened file, which can be used to read data from the file. If the file opens successfully, the method also sets the **targetFile** member variable to point to the opened file. If the file fails to open, the method prints an error message to **standard error output stream** and returns a null pointer.

Afterwards, **fileReader.getFileInfo()** reads the file character by character using a loop, and keeps track of several pieces of information about the file, such as the size of the file, the characters present in the file (stored in a set), and the number of occurrences of each character (stored in a map).

Once the end of the file is reached, the function creates a **FileInfo** object from the collected information and returns it. In other words, the obtained **FileInfo** object contains three pieces of information:

- **alphabet**: a set of unique characters present in the file
- **symbolsCount**: a map that maps each character to the number of occurrences of that character in the file
- **fileSize**: an integer representing the total number of characters in the file

Finally, **getFileInfo** closes the file by calling the method **fileReader.closeFile()** and returns the **FileInfo** object.

Second Pass

Get File Reader

The function **getFileReaderInstance** is called to create a new **FileReader** object for the second pass of the input file. The **inputArguments** object is passed as an argument, which is used to initialize the **filePath** and **windowSize** variables of the new **FileReader** object.

Along with the initialization of the previous variables, we initialize the **currentPosition** with 0, indicating that the reader is initially positioned at the beginning of the file. We also initialize the **currentWindow** and the **lastWindow**, which are dynamically allocated arrays of characters with a size of **windowSize**. These arrays will be used to store the current and last window of characters read from the input file.

Calculate Information

This is an optional step in the program that computes compression metrics (such as total amount of information and information per character) should we compress a text file using copy model logic. This step can be skipped with the use of the "-nI" command line argument.

The **runModelBuilder** function takes **CopyModelInputArguments**, **FileInfo**, **FileReader** and **Logger** objects as input and it then creates a **CopyModelBuilder** object associating the info and reader objects to it. This class contains all the logic and variables for the metrics' computation, such as:

- **pastSequencesPositions**: this is a map that takes strings representing sequences of characters with size k captured from the text file as keys, and a vector of the positions in the text file where the sequence occurs;
- **currentPointerIndexForSequence**: another map with the sequences as keys but the index of the current position being considered for a given sequence in the copy model logic;
- **totalAmountOfInformation**: the amount of information resulting from the compression (updated as it's calculated).

After creating the object and associating a **Logger** instance to it, the **buildModel** class function is called with the user-defined **alpha** and **threshold** as arguments (retrieved from the **CopyModelInputArguments** object). This function is where the bulk of the calculations and file parsing take place and it outputs a file (information.txt) containing the information required to compress each one of the file's characters.

The function advances the window along the file's contents, storing the various sequences and their positions in **pastSequencesPositions**. Whenever we read a window we've

previously seen (and therefore already stored in **pastSequencePositions**), we start the copy model logic.

We first select a pointer to one of the positions where the current sequence has previously been seen. Then, we expand our current window by reading the next character and compare it to the corresponding character in the past sequence. Should the characters match, we get a hit, otherwise, a miss.

The number of hits and misses is used, at each new letter read, to compute the probability of a correct prediction (as shown in the introduction section) while the complementary probability is distributed by the remaining alphabet characters. Finally, we use the probability corresponding to the real character to calculate the amount of information to compress the new character.

This process repeats until the file ends or the probability of a correct prediction goes below the defined threshold, in which case we reset the copy model and change the specific sequence's pointer.

Should a read sequence have not been seen previously, aside from storing the sequence and position in the class variables, the model simply assumes equal probability for all symbols in the alphabet and computes the symbol's information with that probability.

All information for all symbols is summed up in the **totalAmountOfInformation** class variable which then allows us to calculate the information per symbol by simply dividing the total information by the total amount of symbols in the file (function **calculateInformationByCharacter**).

After the **buildModel** function is finished and the model is built we log the compression results as well as some file info using the **logCopyModelResults** function.

Serialize for Generator

This is an optional step in the program that exports, as a text file, a model for text generation. There are two types of models:

- a **probabilistic** model that stores the probability of seeing a certain character after a certain sequence of size k;
- a **positional** model that stores the positions of the various sequences of size k in the file.

The two different types of outputs result in different text generation techniques, explained later in the report. Should the user not wish to export a model for text generation this step can be skipped with the "-nG" command line argument.

The **generateModel** function takes **FileInfo**, **FileReader** and **Logger** objects as input and it then creates a **ModelGenerator** object associating the info and reader objects to it. This class contains all the logic and variables for the model's construction such as:

- **pastSequencesPositions**: this is a map that takes strings representing sequences of characters with size k captured from the text file as keys, and a vector of the positions in the text file where the sequence occurs;
- **probabilitiesForSequence**: another map with the sequences as keys but which values are other maps with alphabet symbols as keys. Initially, these nested maps store the occurrences of each symbol after each sequence, after the model is fully computed, these counts are turned into occurrence probabilities.

After creating the object and associating a **Logger** instance to it, the **generateModel** class function is called. This function's purpose is to populate the class's variables according to the file associated with the **FileReader** instance.

The function advances the window along the file's contents, storing the various sequences and their positions in **pastSequencesPositions** as well as updating the sequence-symbol-count in **probabilitiesForSequence**. Finally, after reaching the end of the file, the **probabilitiesForSequence** is updated to represent symbol probabilities. This is done by dividing the sequence-symbol counts by that sequence's number of occurrences.

After having the **ModelGenerator** object created and populated we call the **serializeModel** function whose purpose is to export the model stored in memory to a text file. As such it takes the **ModelGenerator** object, a string **modelPath** indicating the path where to store the model file, a string inputFilePath indicating the file used to generate the model, an integer **generatedModelType** indicating the type of model we'll be storing and a **Logger** instance as arguments.

The **generatedModelType** will determine which serializer is created:

- A **ProbabilisticModelSerializer** will use the data in the **probabilitiesForSequence** class variable from the **ModelGenerator** object and store the sequence-symbol-probability combinations, line by line, separated by semicolons;
- A **PositionalModelSerializer** will instead use the data in the **pastSequencesPositions** class variable from the **ModelGenerator** object and store the position list for each sequence in the format "<sequence>;<pos1>,<pos2>,...,<posN>,".

Both serializers also include headers indicating the model type and the file from which the model was generated.

Generator

Obtain Generator Parameters

The **GeneratorInputArguments** class is a concrete implementation of the abstract class **AbstractInputArguments**, which represents the input arguments for a text Generator using a given Copy Model.

Member Variables

This class has several private member variables: **modelPath**, **amountOfCharactersToGenerate**, **loggingLevel**, **interactive**.

- **modelPath**: A string that represents the path to the Copy Model file.
- **amountOfCharactersToGenerate**: An integer that represents the number of characters that the program should generate.
- **loggingLevel**: An integer that represents the logging level of the program.
- **interactive**: A boolean that indicates whether the program should run in interactive mode or not.

Main Methods

The public section of the class defines several member functions to access these private member variables. Additionally, the class also defines two member functions: **parseArguments()** and **checkArguments()**.

- **parseArguments()**: This function takes in the command line arguments and sets the values of the corresponding member variables.
- **checkArguments()**: This function checks the validity of the member variables and returns a boolean value indicating whether the arguments are valid or not.

Finally, the class also has a static member function **printUsage()** which is used to print the usage information for the program. This function does not take any arguments and simply prints a message to the console.

Parse Arguments

The **parseArguments()** method of the **GeneratorInputArguments** class is responsible for parsing the command line arguments passed to the program and updating the corresponding fields of the **GeneratorInputArguments** object.

The method takes two arguments, **argc** and **argv**, which are the number of arguments passed to the program and an array of pointers to the argument strings, respectively.

The function first prints out the number of arguments entered and checks if any arguments were entered at all. If no arguments were entered, the function prints an error message, calls another function called **printUsage()**, and exits the program with a failure status.

If arguments were entered, the function loops through the arguments to check for two specific flags (-i and --help). If the **-i** flag or the **--interactive** flag is present, the interactive variable is set to true. If the **-h** flag or the **--help** flag is present, the **printUsage()** function is called and the program exits with a success status.

Next, the function loops through the arguments again to check for three more flags: -m, **--model**, **-g**, **--generationCharacters**, **-l**, **--logging**. If the **-m** flag or the **--model** flag is present, the **modelPath** variable is set to the next argument. If the -g flag or the --generationCharacters flag is present and the next argument is a valid number, the **amountOfCharactersToGenerate** variable is set to that number. If the -l flag or the --logging flag is present and the next argument is a valid number, the **loggingLevel** variable is set to that number.
Next, the function loops through the arguments again to check for three more flags: -m,

Finally, if the **amountOfCharactersToGenerate** variable is still 0, the function sets it to the default value of 50 and prints a message informing the user.

Check Arguments

The purpose of the function **parseArguments** is to parse the command-line arguments and store the values in member variables of an object of the **CopyModelInputArguments** class.

The function first checks if no arguments were entered and prints an error message and the usage instructions if that is the case. It also handles the help flag **-h** or **--help**, which prints out usage instructions and exits the program.

Then the function iterates over the arguments and checks for specific flags while parsing the values provided after each flag. For example, if the **-a** flag is encountered, the function checks if the next argument is a number and sets the parsed **alpha** value accordingly. If the **-i** flag is encountered, the function sets the parsed input file path value to the next argument, and so on.

If any of the parsed values are not provided, the function sets them to default values and prints out a message indicating this. Finally, the function sets the member variables of the **CopyModelInputArguments** object to the parsed values.

Get Model Type

In order to obtain the Model Type of a given Copy Model, we have implemented the function **getModelType** that takes a string **modelPath** as input and returns a **ModelType** enum indicating the type of the model.

The function first opens the file located at **modelPath** and checks if it was successfully opened. If the file could not be opened, the function returns **ModelType::UNKNOWN**.

If the file was opened successfully, the function reads the first line of the file and checks if it is empty. If the first line is empty, the function returns **ModelType::UNKNOWN**. Otherwise, the function searches for the first occurrence of a colon in the line. If a colon is not found, the function returns **ModelType::UNKNOWN** once again.

If a colon (:) is found, the function extracts the substring after the colon and checks if it matches any of the two ModelType enum values: **POSITIONAL** or **PROBABILISTIC**. If there is a match, the corresponding ModelType enum value is returned. If there is no match, the function returns **ModelType::UNKNOWN**. Finally, the function closes the file before returning the **ModelType** enum value.

Model Type: Positional

For Positional Model Type, we first intialize a **PositionalModelSerializer** with the model path obtained from the command line arguments. This class is a derived class from the **AbstractModelSerializer** class, and it has its own implementation of the **outputModel()** and **loadModel()** methods.

The PositionalModelSerializer class also has a private member model which is a **map** that stores the positional model. Additionally, it has a **positionsPerSequenceLimit** variable that determines the maximum number of positions allowed per sequence.

The **getModel()** and **setModel()** methods are public methods that allow getting and setting the model, respectively.

Load Model

This method, **loadModel()**, loads a positional model from a text file with a specific format. The method first checks if the model is empty: if not, it clears the model. Then, it opens the text file specified by the model path. If the file is open, it reads the second line of the file to get the path of the original input file.

After that, it reads the file line by line. For each line, it parses the line to extract the **window** and the **sequence** (i.e. the list of positions). It then creates a positions vector, **sequenceVector**, by extracting each integer value from the sequence string and adding it to the vector. Finally, it inserts the window and sequence vector to the **model** map.

If any errors occur while parsing the file or creating the sequence vector, the method returns **false**. Otherwise, it returns **true**.

Positional Generator

The **PositionalModelSerializer** class provides methods to load and store the model in a map, which maps a sequence of characters (called a "window") to a vector of integers. The **PositionalGenerator** class uses this model to generate text interactively or non-interactively.

Interactive Mode

When generating text interactively, the program prompts the user to enter a sequence of characters and then repeatedly generates the next character in the text until the desired length of text is generated.

For this, the function **generateTextInteractively** enters an infinite loop that prompts the user to enter a sequence - if the user enters "exit", the loop breaks and the function ends. The input sequence, **initialText**, is passed as a parameter to the **generateText** function, along with the positional model, **model**, and a boolean flag, **real-time**, indicating that real-time output should be displayed.

The method **generateText** first creates a **RandomAccessReader** object and opens the input file associated with the positional model. Then, it checks if the length of the **initialText** argument is less than the size of the model's window. If so, it prints an error message and returns an empty string.

Next, the method sets **currentWindow** to be a substring of **initialText** that is the size of the model's **window**. If the model does not contain any data for **currentWindow**, the method prints an error message and returns an empty string.

The method then initializes an empty string called **completedOutput** and enters a loop that generates characters until **completedOutput** is the desired length. In each iteration of the loop, the method updates **currentWindow** by shifting it one character and adds the next character to **completedOutput**. The next character is generated using the **generateNextCharacter** method, which takes the **currentWindow** and **randomAccessReader** as arguments, and appended to the end of **currentWindow**.

For each character generated, after updating the current window, if the window is not present in the model, the function stops generating characters and returns the completed output.

If real-time output is enabled, each generated character is immediately printed to the console. Once all characters have been generated, the function prints a newline character to the console to separate the generated text from the input prompt. The completed output is returned by the function.

Non-Interactive Mode

When generating text non-interactively, the program prompts the user to enter a sequence of characters and then generates the completed text with the desired length.

For this, the function **generateTextOnce** takes an integer argument **generationSize**, prompts the user to input a string sequence, generates text based on the provided sequence and the model data, and outputs the generated text to the console.

The input sequence, **initialText**, is passed as a parameter to the **generateText** function, along with the positional model, **model**, and a boolean flag, **real-time**, indicating that real-time output should be displayed. In contrast with the Interactive Mode, the real-time output is set to **false**.

Model Type: Probabilistic

For the Probabilistic Model Type, we have the **ProbabilisticModelSerializer** class, which inherits from the **AbstractModelSerializer** class and is responsible for loading and storing probabilistic models.

The class contains a private member **model**, which is a map that associates a string (a sequence of characters) with another map, which in turn associates a character with a probability value.

Similarly to the **PositionalModelSerializer**, the **ProbabilisticModelSerializer** class has the public methods **outputModel()** and **loadModel()**, which respectively output and load a model from a text file. The text file format has two lines: the first one is ignored, and the second one specifies the path of the original file used to create the model.

Load Model

The method **loadModel()** loads a probabilistic model from a file specified by **modelPath** and stores it in a map called **model**. The format of the file is assumed to be as follows:

- The first line of the file is ignored.
- The second line contains the path to the original file that was used to generate the model.
- The rest of the lines contain three values separated by semicolons (;): a sequence, a character, and a probability. The sequence represents a prefix (i.e. window), the character represents the next symbol in the sequence, and the probability represents the probability of observing the character given the sequence.

If any errors occur during the parsing process (e.g., if the line does not contain the expected number of semicolons), an error message is printed to **cerr** and the method returns **false**. If the file cannot be opened for reading, an error message is printed to **cerr** and the method also returns **false**. If the method completes successfully, it returns **true**.

Probabilistic Generator

The **ProbabilisticModelSerializer** class provides methods to load and store the model in a map, which maps sequences, characters, and probabilities. It contains information about each sequence, each possible character following the sequence and its probability.

The **ProbabilisticGenerator** class uses this model to generate text interactively or non-interactively.

Interactive Mode

When generating text interactively, the program prompts the user to enter a sequence of characters and then repeatedly generates the next character in the text until the desired length of text is generated.

For this, the **generateTextInteractively** method retrieves the model from the **probabilisticModelSerializer** object and starts an interactive loop, where the user inputs a sequence, **initialText**, and generates text based on that sequence.

The **generateText** method generates text based on the provided **initialText** and model. It checks if the length of **initialText** is at least the window size of the model. If it is not, it prints an error message and returns an empty string. It then sets the **currentWindow** variable to the last **getModelWindowSize()** characters of **initialText** and the **generatedText** variable to **currentWindow**.

It then enters a loop where it generates the next character based on the current window and adds it to **generatedText**. For generating this next character, the function **generateNextCharacter** takes the generated text as input and extracts the last **getModelWindowSize()** characters to create the current window. It then retrieves the probabilistic model from **probabilisticModelSerializer**, and using the current window, it

retrieves the corresponding probability distribution map of characters, **charactersProbabilityDistributionMap**.

Here, we can apply the **getCharacterUsingUniformDist** function to generate the next character using the probability distribution map, **charactersProbabilityDistributionMap**. The **getCharacterUsingUniformDist** function selects a random character from the probability distribution map using uniform distribution. This returns our next character.

If **realTime** is true, it prints the generated character to the console. It then updates the **currentWindow** variable to the last **getModelWindowSize()** characters of **generatedText**, continuing the loop. Finally, once it has generated enough characters, it returns the generated text starting from **getModelWindowSize()** characters to the end.

Non-Interactive Mode

When generating text non-interactively, the program prompts the user to enter a sequence of characters and then generates the completed text with the desired length.

For this, the function **generateTextOnce** function prompts the user to enter a sequence of characters and generates a text using the probabilistic model - with the given sequence as the initial window. The generated text is printed to the console, along with the initial sequence used to generate it. This function also uses the **generateText** function to generate the text.

If the user enters "exit" instead of a sequence, the function returns immediately without generating any text. The function also logs the time taken to generate the text.

Results & Discussion

Copy Model Results

As we can see in **Figure 1**, as we increase the threshold, the average amount of information per symbol tends to decrease. This was expected because when we have higher values of threshold, we will tolerate fewer failures a.k.a. Misses, and will not expand the Copy model as much as we would otherwise. Once we tolerate fewer failures it is normal that our Copy Model is closer to what was seen and takes fewer risks when trying to predict. This also justifies more information per symbol (and therefore worse compression) when the threshold is lower - as the Copy Model will risk too much.

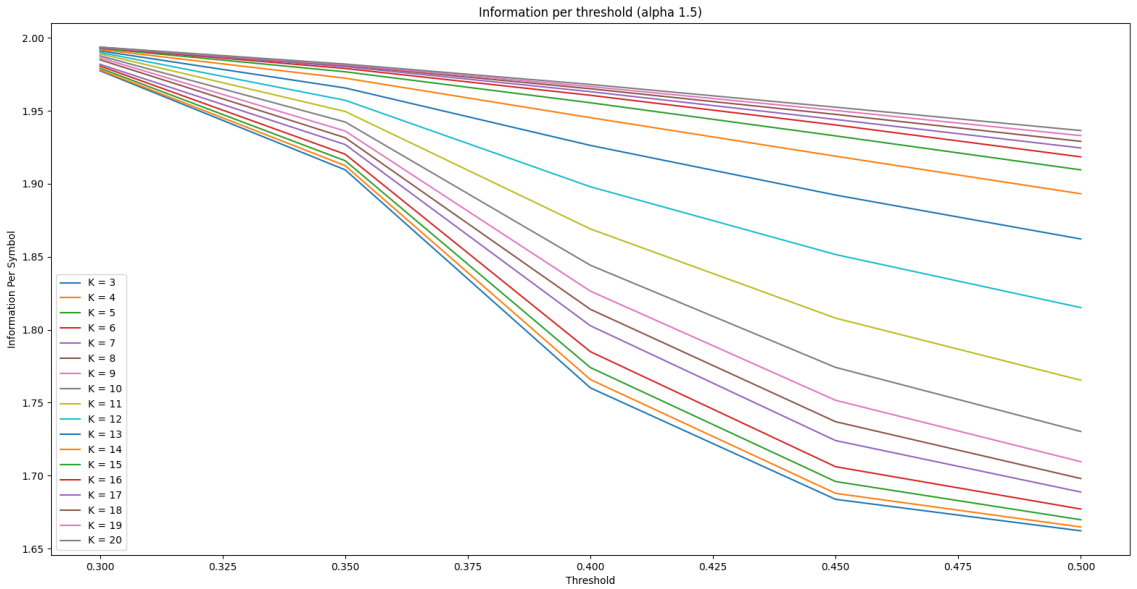

Figure 1: Evolution of Information p/ Symbol as we variate the Threshold for the different values of K.

In **Figure 2** we once again take notice that lower values for K and higher values for the threshold result in better compression for the same reasons explained above.

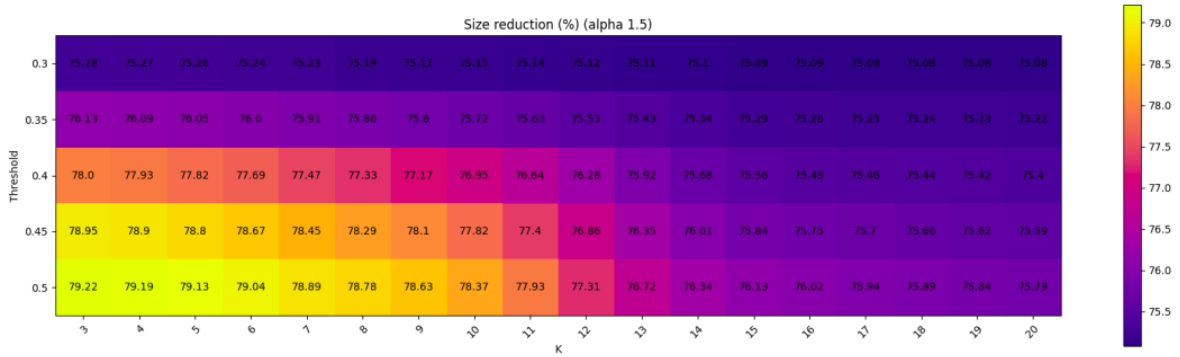

Figure 2: Size reduction (compression in percentage) for various values of K and threshold.

In **Figure 3** , aside from observing how the higher thresholds generally lead to better compression results, we may also note that higher alphas tend to work best with higher thresholds, this is due to the fact that higher alphas will initially produce lower initial hit probabilities. For example, if we have 1 hit and 0 misses at first, an alpha with a value of 3 will produce a hit probability of 0.56 whilst an alpha with a value of 0.1 would produce a probability of 0.91, so for higher alphas the copy model will give up on (potentially good) pointers more easily being more sensitive to misses. In conclusion, the values for alpha and the threshold must be pondered in conjunction and the tendency seems to favor higher values in both parameters for best results.

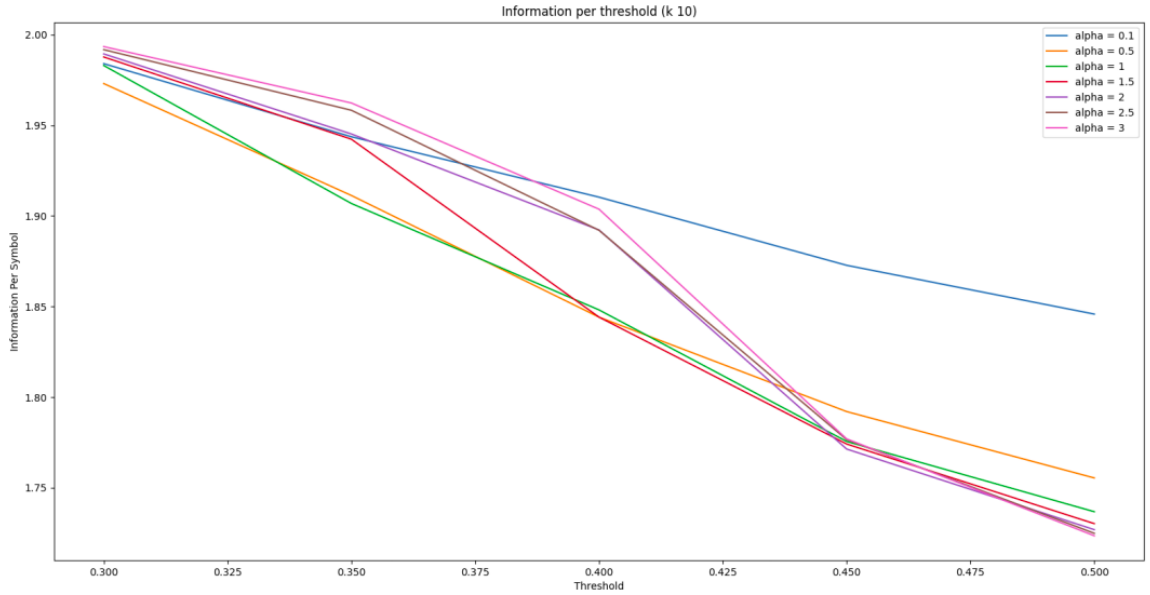

Figure 3: Evolution of Information p/ Symbol as we variate the Threshold for the different values of alpha.

Build Time

For probabilistic models that rely on a window of k characters to predict the next character in a sequence, the computational complexity of building the model **increases significantly as k decreases**. This is because a smaller window size requires the model to process more unique subsequences of characters, leading to a larger number of possible combinations and a much larger search space.

As the window size decreases, the number of possible subsequences of characters that need to be processed grows exponentially. This means that the time required to build the model increases rapidly as the window size decreases.

It is generally more computationally expensive to build a probabilistic model with a smaller window size than with a larger window size. We can observe this in Figure 3, with k = 3 displaying the greatest average build time.

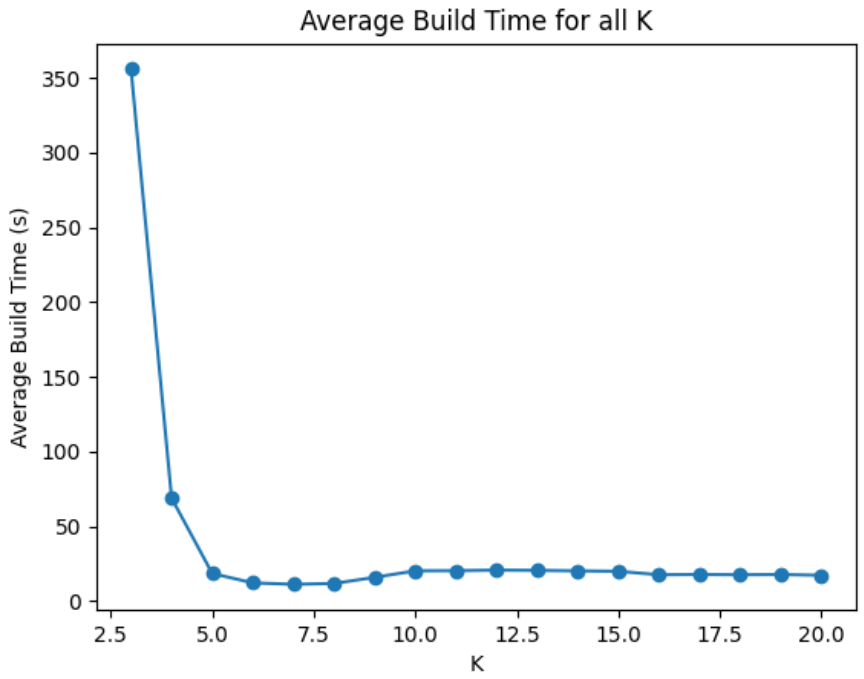

Figure 3: Average Build Times for all possible K.

Generated Text

As already explained, our Generator implementation accepts two different types of models - a **probabilistic** and a **positional** model. To demonstrate the differences between the two models we have generated texts with **500 characters** for different values of K.

We have fixed the **alpha** and **threshold** values (alpha = 0.5 and threshold = 0.45) for all the text generated so we can emphasize the comparison of the K values. The results obtained for the different K values are displayed below. For each K value, we will also compare the 2 different models (probabilistic vs. positional). The model on which the generator is based was built using the book "A Tale of Two Cities" by Carles Dickson.

K = 1

```
(I break down before the knowledge)the pytho bend OOn. hen, t. aces d cind y, ito Yerou " wat ces ale tnio herawindendancthive ashed ine Teje, hiley, " nwand ased y e te athouthefid wisof, "I inest conde Derehildald mmitond, imacisenthe oul are sthind Lores rke stcomenke l on hoccangomy, agete. ily hawam s wan's t Thathe, msblontondag careass fibld, t lomexpathand ocamexarsinerchechit irere rr, ad coud, has nthreend it isif adr thotorundais, " yshathtideng as_pin idedou' led t trre cereif t th, and anirse wemt ond anit tathade an
```

Output 1: Text generated for K=1 using Positional Model.

```
(I break down before the knowledge)xat hioofat s himalartedimyowe, ancos sn's, ll memangel tholis ainly, way an s a ara halle t plet edissaty feld " worm. t ourd wherrimed. " tovel, mell S un hemeancesert llldompind as. tatlomathe Yediro ss cherlout san t Mo ove g tt, furt aburealerril tweee as, I th, wngigh, tethar, the? I ge, ftren r andat jese hes te asiceronthe les so, fan delorurur exig, wherrer w-Lue Inetinerry t wawh s. t se f bror y ark, me ty ig wag ilay. tstiselecr d m sir k, blfr. ts he t eamsun l XChanesisse fosoutere
```

Output 2: Text generated for K=1 using Probabilistic Model.

As expected, for K=1, i.e., for windows with 1 character, the generator failed to generate meaningful text. Indeed, it wasn't even able to generate words that exist in the English vocabulary. This was already expected because the generator doesn't have enough context to generate the next character as it will only take into consideration the last character to generate the next one. Therefore, it is expected that as we increase the value of K the generator will be capable of generating words that exist in the English vocabulary, and if we increase it sufficiently it should even be able to generate meaningful sentences. Both models generate similar text, i.e., texts that have a similar amount of valid words. Obviously, the generator incorporates some degree of randomness in both models and therefore, the text generated by both models won't be the same.

K = 2

```
(I break down before the knowledge)d and beare she of the suanot houl othat matid, witte, delled, and art olive my putchady was lon thaver, and ste cied." "Ven therve made, dearoseligh den, the in mintuessid Defor Chand th ace ands the of hom of the expe." Thady bell yes: "Yahe laguld th parenuive. As foread: "Goodoge dre to therhat me, astrout and thess onectileept by che rated it mand youned mencithe in of the ney. goldly was by glit's a frodled hit, and han th the upon the sonst dan to shmanceenet _yon of thiss ing to goll ret
```

Output 3: Text generated for K=2 using Positional Model.

```
(I break down before the knowledge) and youst wortle comession hiss way it, the and they ther ing this ve shad the flut thertioutell. Loaten onearmis ifideed haver re cons they diager, drom waskentut the stred eyearsto rearquille craceshe reacted Dard all known ble therry of fid, I gaid: an as tharfuld a samend but, begs saing handir_ hime and a phe the cons ingersaing aned. as all, "Som son was flas ton conal the cup of be yough amessivery fer a re uppy. A flood the th ate. and tow--I was onat of as hishowen Dargess, nothedithat
```

Output 4: Text generated for K=2 using Probabilistic Model.

With K=2 we already start to see some meaningful words. However, we still see some words that do not exist at all in the English vocabulary. Again, this may be explained by the lack of sufficient context. We do not see meaningful sentences yet. Both models are very similar and do not present significant differences.

K = 3

```
(I break down before the knowledge)d, sity, quasist and eith Threws right tree ther she her whened of a mothe groubt. A golder. Lorry," punch the doercised one business Projectful othe gents of Two othis excition, and withe intem, with a word so," purcher all I musty habitator hapted thate after ago. "But, from ther! Mons, andy ever notioness lattack mention, tere to old, int. Acces have bothis here enjoyalterty of bad ent parter what at night was that vereupon in and back, to seen the befor cour: the man a nevery thand bad had p
```

Output 5: Text generated for K=3 using Positional Model.

```
(I break down before the knowledge)structionated assat a linger, her you meth qualiken them passionseizing out?" said or in one muting he me. Now a hard strue." "That her," saken in fathe to-more a like the to you." "Never you play in to to oth a light much wais, suppression some, that, most, and hid Cly, ast? Is the stancy to the lish of a ver her bread that know." "Yes, hand seemed of the its wors fit it. Hund loving an I wing do numsy litting awful wised writion a feith, wild from think to, hop his least of the sher their visi
```

Output 6: Text generated for K=3 using Probabilistic Model.

With K=3 we already see even more meaningful words and fewer words that do not exist at all. Both models are, again, very similar and do not present significant differences.

K = 5

```
(I break down before the knowledge), and towards the tried at their hardly employees expect to made the might deep such a chancing two was date common and the memorial bearing. "That was arose of in them, under his reflect that lists of flames in the weather's flounded in lieu of a horses. T'other and from the first Minister straw in ten marked upon him. "Good Miss Manette keep again, was medley dropped time to precipitation. Characted pallet-bed, and honour husband oiled them how to hold one day. I may had been, for anything any
```

Output 7: Text generated for K=5 using Positional Model.

```
(I break down before the knowledge) of the linen and correction exactly on the first encompany. the strong and sort of his parts a larger similar condemned their name?" asked to us. height hand, fee or the silently praying her little practical spirit, he head thinking-glass over his head that the more the was a reasons to be surely not--for it." "Again raising kinds of France is it?" "Your beginning unreal hope them all about inquirement, and of life in them and in her. Like Monseigned resemblages and round his father's table. "
```

Output 8: Text generated for K=5 using Probabilistic Model.

With K=5 we start seeing, for the first time, text that is filled with meaningful words. However, we cannot see valid sentences yet. To create relevant sentences the generator would need even more context (i.e., a higher value of K). Again, both models are, very similar and do not present significant differences in terms of generated text.

K = 10

```
(I break down before the knowledge), and I know that, if it were the fingers on the side of the mortally stricken. Now, the sun was so low that not a sound was heard, Mr. Lorry bustled into the burial-places of business. A crown to drink. I may read in the natural to it than sunshine. The whole jury, as a jury of dogs empannelled to try the deer. Every eye then sought some other heap of poor grass. Monseigneur, without looking through stone walls, eight great towers. Slight displaced a leaf of the hands that held his fortnight, w
```

Output 9: Text generated for K=10 using Positional Model.

```
(I break down before the knowledge)?" "Not even so. I might (mistaken or not mistaken) have that had been drowned and filled with great ardour to the acquitted herself, and lowered his voice. and the action of the Abbaye, that were any) there
```

```
was no other answer the question." "Willingly, sir," said the men at the table in the innermost personality, to me, or the air, almost like the top of each nostril. In those two passenger a small measure of oil for the dawning Republic should demand of you than of these two hands laid on Dar
```

Output 10: Text generated for K=10 using Probabilistic Model.

With K=10 we start seeing, for the first time, text that contains meaningful words and some small sentences that make sense (e.g. "`and lowered his voice`"). These results can already be considered very good for a generator that generates text one character at a time. However, these results should be improvable with even more context, i.e., with a longer window (i.e. a higher value of K).

K = 15

```
(I break down before the knowledge) of the step, as a step taken, and not in the balance of suspense and doubt. How much of the incompleteness of his situation engendered came through the brown upon his cheek, showing the soul to be stronger than you, I bless Heaven for it. I hold you till one or other of us faints or dies!" Madame Defarge had not moved from the wall of the front hall--as if he had been spirited away. if it had not been difficult to guess to what dreadful place, though by no means new to his own mind, had been hu
```

Output 11: Text generated for K=15 using Positional Model.

```
(I break down before the knowledge) of the step, as a step taken, and not in the balance of suspense and doubt. How much of the incompleteness of his situation was referable to her father, through the painful anxiety to avoid reviving old associations of France in his mind, he did not discuss with himself. But, that circumstance. Mr. Darnay, good night, God bless you." He put her hand to his lips. It was a gentle action, but not at all gently done. a very remarkable transformation had come over him in a few seconds. He had no goo
```

Output 12: Text generated for K=15 using Probabilistic Model.

As mentioned above, with a higher value of K we should be able to get even better text, i.e., text with longer valid sentences. With K=15 (higher than 10), we can have long sentences that are meaningful and that can be interpreted. An example in the first text is "`I hold you till one or other of us faints or dies!`" - this is a meaningful sentence! We considered that for a generator that only produces a single character at a time, these are great results and therefore, we did not continue to increase the value of K. It is worth noting that as we use higher values of K, the size of the generated model increases and leads to a slower generation of text as, in our opinion, it is not worth the higher overhead generated by the larger size of the model.

Another interesting point to consider is that as we increase the value of K the randomness of the text reduces. This can be demonstrated by the fact that for K=15, both generators (using the Positional and Probabilistic models) generate the same first sentence - "`(I break down before the knowledge) of the step, as a step taken, and not in the balance of suspense and doubt. How much of the incompleteness of his situation`".

Positional Model vs. Probabilistic Model

As we can see in the examples below, the Positional model beats the Probabilistic model in terms of execution times. In the examples presented, the Probabilistic model takes more than 22 seconds to generate a 500 characters text compared to the Positional model time. Therefore, although we succeeded to implement both models and make them fully functional, and although both models can generate somewhat meaningful text (for high values of K) the Positional model should be used instead of the Probabilistic model if we are aiming for performance.

```
(We have oftentimes the honour to entertain your gentlemen) do. Ever borrow money
of the people from the dark window. It was a heavy mass of scarecrows of that
woman's, to-night, when delay was the epoch of belief, it was the careless wave of
her right hand in the mouth. and one tall joker writing up his joke, he
called?--in a little moment." "You spoke of the young lady, who were seated at a
low door, put a key in a clashing lock, swung the door: "At last it is come, my
dear Manette," said Defarge, and quietly walked in, took his seat. "I received it,
```

Output 13: Text generated for K=10 using Positional Model. Time = 123076ms

```
(We have oftentimes the honour to entertain your gentlemen)." "Had they conferred
together, Mr. Lorry. "If you had been so complemented, and worked, in silence. The
bank closed, the ancient chum is a man whom he saw making his head. for he gained
in firmness of purpose, seemed to be no customers." "Story!" He seemed wilfully to
mistake the words of explanation in his hearing as just spoken--distinctly and
composedly in his smiling face, and his look immediate object of this plotting and
locking their heads upon the door he sought. And that aid was his f
```

Output 14: Text generated for K=10 using Probabilistic Model. Time = 145627ms

```
(In a very clear and pleasant young voice). This man, whom he had studied the
figure was hauled up forty feet high, poisoning the walls with a determined to
hold it, moved after them with his usual white linen, and with success, I speak of
success, and with an air of mystery to every head was lifted up the most violent
Quarter, near the happiest frame. Now, I hear Somebody's step coming to him
affectionate brother you have to do with it, Aggerawayter?" "I was returning from
France a few days. And yet I think--I know--he does." "If you k
```

Output 15: Text generated for K=10 using Positional Model. Time = 125901ms

```
(In a very clear and pleasant young voice). The pen dropped upon his journey in
France, and Jacques Three, in a lower voice, striking in to help him to any other
parent. Thus the evening air, as if he were old. but when it was finished, two
centuries ago. Up the broad flight of shallow steps, Monsieur the Marquis. It was
an ordinance of the harbour. He did it for myself--_my_self, Stryver on the shore.
My friend is dead, you learn that I heard strange feet upon the birds was loud and
harsh an accompanied with anything about the joints?
```

Output 16: Text generated for K=10 using Probabilistic Model. Time = 146449ms

Conclusion

In this assignment we successfully implemented a Copy Model to estimate the amount of information in a text file. We also use this Copy Model implementation to estimate the average amount of information per character in the file.

We also implemented a Generator that taking an input file as a basis can generate text that obeys some of the text it was able to process on that input file. This Generator can operate with two different types of model: a Probabilistic Model - which takes into consideration the probability of a certain character after a certain sequence of text to generate the text character after character and a Positional Model which uses the positions in which sequence of text appears in the input file to get the next character.

Finally, we have some results for the Copy Model such as the time taken to run in the function of K, and other important metrics. We also show some results of text generated by the Generator we implemented. We also do a brief discussion and interpretation of the results we obtained.