University of Glasgow | School of Computing Science

Automated Genome Informatics Workflow Development

Zoe Gerolemou - 2023949

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 25, 2016

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: _____ Signature: _____

Contents

Abstract

DNA Sequencing became a very emerging topic in the past few years, especially after the vast advancements in the development of Next-Generation Sequencing (NGS) techniques. Apart from the research that is currently in progress which aims to improve these methods, a significant amount of study is devoted to the optimisation of the algorithms that are used for the analysis of the resulted data. The aim of this project was to study the current version of an Origin Classification script used for Genetic Sequencing purposes by performing deterministic and memory profiling on it and spotting any specific operations that may add a significant overhead on its execution. According to the results of this process, a number of different optimisations were applied and evaluated in order to determine which of them were the most capable for a future adaptation to the script. The evaluation procedure followed a very careful and sophisticated experimental approach. The process focused on the optimisation of String Concatenation which is a very costly operation that was repeated multiple times within the script. The evaluation process proved that the most optimal alternative to it was the use of List Comprehensions. This solution managed to improve the performance of the script to a very significant degree and it is highly recommended for introduction to the system.

Acknowledgements

I would like to thank my supervisor, Dr. Simon Rogers, for providing his guidance, support, expertise and knowledge throughout the project duration.

In addition, I would like to thank Dr. Nick Dickens from the Wellcome Trust Centre of Molecular Parasitology for his assistance and insights on the operation of the Origin Classification tool.

Chapter 1

Introduction

DNA is a molecule that serves as the major carrier of genetic information in the living organisms. It is consisted of two strands that are coiled around an imaginary axis. These strands contains a number of nucleotides; each one of them is formed by one of the following nitrogen-containing nucleobases: *guanine* (**G**), *adenine* (**A**), *cytosine* (**C**) or *thymamine* (**T**) [46].

DNA has recently become subject of particular interest for Computing Science. The rise of a new scientific field that deals with the DNA sequencing of whole genomes for research and medical purposes and the computational analysis behind it has led to extensive study and multiple publications regarding algorithms and technologies that could be possibly proved useful to support all the underlying analysis. From a purely computing science perspective, this process acts as a very good example of the risks and challenges that lie behind a String Manipulation problem in the larger scale.

The Wellcome Trust Centre for Molecular Parasitology (WTCMP), based at the University of Glasgow, uses a computer-based sequence classification tool which acts as a pipeline for the manipulation of information stored in DNA molecules. The samples they are working on are collected using Next-Generation Sequencing (NGS) techniques. Their analysis aims to provide results for the occurrence of viruses like the trypanosome, leishmania, toxoplasmosis and plasmodium.

This project aims to improve the performance of this tool in order to be capable of scaling up in the future. That will be achieved by analysing and profiling the current version of the script to determine the points where the performance is downgraded and proposing possible optimisations that are intended to mitigate their effect and increase the efficiency of the script both in terms of speed and memory usage.

1.1 Report Outline

The rest of the report is structured as follows:

- **Backgroud.** A brief introduction to the process of Genetic Sequencing along with some important milestones from its history including the Human Genome Project.
- **Underlying Algorithms and Technologies.** An overview of all the important algorithms and the different technologies used by the script.
- **Current System Evaluation.** A thorough evaluation of the current system with a detailed discussion of the results generated from its Deterministic and Memory Profiling.

- **Study of Possible Optimisations.** A list of possible optimisations for improving the performance of the script along with an experimental process of applying and evaluating them within the context of the Origin Classification tool.
- **Conclusions.** A summary of all the key conclusions and personal reflection that resulted from the optimisation process and the project in general.
- **Future Work.** Some suggestions for further optimisations that could be applied in any future iterations of the project.

Chapter 2

Background

The following chapter provides a brief historical overview on the field of Genetic Sequencing. It describes the different Sequencing Techniques that were implemented over the years and the context in which each of them was developed and applied. A big portion of this chapter is dedicated to the Human Genome Project which is considered to be a great milestone in the history of DNA Sequencing.

2.1 First-Generation Genetic Sequencing

The numerous new advancements in medicine, virology and biotechnology demanded the analysis of the DNA sequences. Therefore, in the 1970s, Scientists started to develop a new process, called ”Genetic Sequencing”, that is dealing with the specification of the precise order of nucleotides within a single DNA molecule. The first attempts were made by two American molecular biologists, Allan M. Maxam and Walter Gilber that proposed the *Maxam-Gilbert method* and Frederic Sanger, an English biochemist, who suggested the *Sanger method* [17]. The Sanger method will be discussed further in one of the next subsections.

2.2 Human Genome Project

Along with the different developments on the techniques used in Genetic Sequencing over the years, the rapid improvements on the performance of microchips and processors enabled the faster manipulation of enormous amounts of data that resulted from the process [69]. The DNA Sequencing reached a peak with the beginning of the US Human Genome Project. The project was proposed by the National Research Council in 1988 as an attempt to map the human genome and was officially planned as a 15-year project in 1990 by the U.S. Department of Energy and the National Institutes of Health [12].

A large number of volunteers from diverse backgrounds and localities was recruited to provide DNA samples for sequencing. In order to ensure that the identity of these volunteers would not be disclosured, a very careful process was followed to protect their anonymity. Almost 5 to 10 times more people had donated blood than the number of samples that were used to make sure that even the volunteers themselves could not know whether their samples were eventually used or not [25].

It was completed in April 2003, two years ahead of schedule. All the data produced was made openly available on the Internet almost immediately. Its results fueled the discovery of more than 1800 diseases. Due to the success

of this project, researchers are now capable of finding the genes that are suspected for causing an inherited disease in only a few days [26].

2.3 Next-Generation Genetic Sequencing

In the early 90s a new wave of Genetic Sequencing techniques were proposed and patented. The manufacture of computer machines of high computational performance provided the means for creating sequencing systems that can process huge amount of data instantaneously. Some of them were, eventually, patented as Commercial DNA Sequencers by the beginning of the 21st Century. These techniques are capable of providing high-throughput sequencing in which DNA and RNA can be sequenced more quickly and cheaply than in the traditional techniques that were proposed during the First Generation era [15].

2.4 Sequencing Techniques

2.4.1 Chain Termination

In 1977, Frederic Sanger proposed the very first DNA sequencing technique both known as *Chain Termination* or *Sanger method*. The development of this technique is considered to be one of the greatest milestones in the history of Genetic Sequencing since it is the one on which all the other techniques were based and it is up until today the most commonly used method [68].

There are four types of *deoxynucleotides* (**dGTP**, **dATP**, **dCTP** and **dTTP**) and *dideoxynucleotides* (**ddGTP**, **ddATP**, **ddCTP** and **ddTTP**); one for each of the four nucleic bases (**G**, **A**, **C** and **T**). This specific type of altered nucleotides have the special property of preventing the addition of any further nucleotides when they are integrated into a sequence [65]. This technique also requires the conversion of double-stranded DNA into single stranded DNA by denaturing it with *Sodium Hydroxide* (**NaOH**) to break the weak linkages and bonds [16].

The *Sanger Reaction* is prepared by adding together the single strand of DNA that was produced above with DNA primers which are short pieces of DNA that are complementary to this strand, a mixture of a particular *dideoxynucleotide* (**ddNTP**) with its own normal *deoxynucleotide* (**dNTP**) and the rest of the *deoxynucleotides*. This reaction is repeated four times; one for every *dideoxynucleotide*. Then, the *PolyAcrylamide Gel Electrophoresis* (**PAGE**) process is performed in order to separate the components of this mixture with respect to their size [67]. The gel is analysed using the process of *Autoradiography* where only the bands with the radioactive label on the *5' prime end* will be presented. The above steps can indicate the particular order in which a *dideoxynucleotide* was added by noticing which band has migrated the furthest and, accordingly, assume that its complementary base appears on the *3' prime end*. The *5' prime* and the *3' prime* that were mentioned earlier refer to the carbon number in the DNA's sugar backbone where the former has a *phosphate* group attached to it whilst the latter is attached to a *hydroxyl* one [56]. By repeating this process, one is capable of reading the whole DNA sequence that was used as input [11].

2.4.2 Pyrosequencing

In 1996, Mostafa Ronaghi and Pål Nyrén who were located at the Royal Institute of Technology in Stockholm published a new Genetic Sequencing technique called *Pyrosequencing*. This method takes advantage of the identification of *pyrophosphate* (*PPi*) released during DNA Sequencing [47].

Firstly, a DNA segment is undergoing the process of *Amplification*. During *Amplification*, a region of DNA that contains a gene is duplicated. In parallel, the template that is going to be used for the *Pyrosequencing* is attached to a *conezyme R* called *biotin* [61]. This process is known as *Biotinylation* [63]. After the modification of the molecular structure of DNA through *Denaturation*, the resulted *amplicon* that is consisted of a single DNA strand is isolated and cross-bred with a sequencing *primer*, a short nucleic acid that acts as the starting point for DNA synthesis [52]. Later, the result of this hybridisation is incubated with various enzymes, including the *Adenosine Triphosphate* (**ATP**) *sulfurylase*. After a while, a *deoxyribonucleotide triphosphate* (**dNTP**) is added to the above reaction and causes the release of *pyrophosphate* (**PPi**). *ATP sulfurylase* converts **PPi** to **ATP** and generates visible light. The amount of light produced is proportional to the amount of **ATP** [44].

The light is detected by a *Charge Coupled Device* (**CCD**) camera which represents them visually in a Pyrogram that shows the raw output data. The enzyme of *Apyrase* causes the periodical degradation of any unincorporated nucleotides. By the end of the process, the complete nucleotide sequence is built and determined by the peaks of the signal in the aforementioned Pyrogram [44].

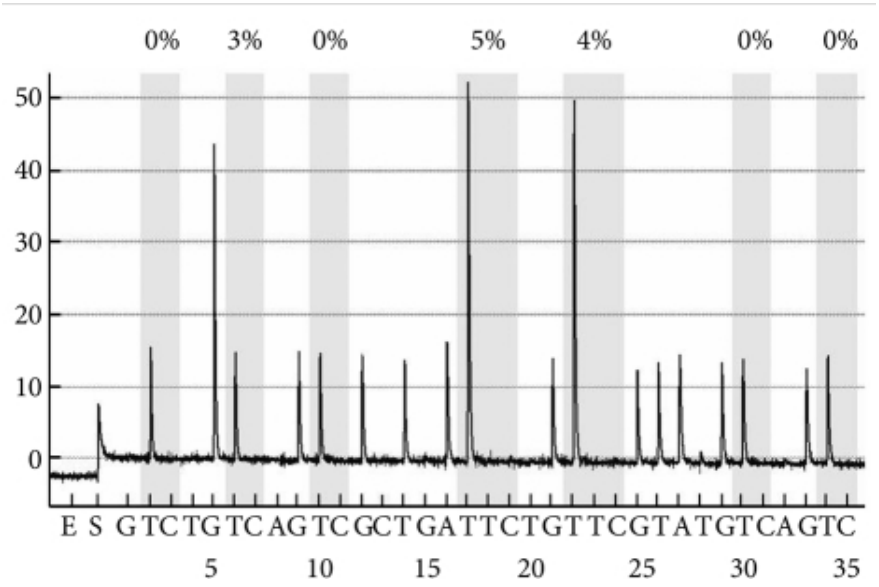

Figure 2.1: A representative **Pyrogram** of the *5HTT* gene promoter [32]

From its very early steps, *Pyrosequencing* was considered to be of very high potential. And indeed the history did not disprove this claim since up until today it is one of the most precise Genetic Sequencing techniques due to the facy that it provides **99.75%** accuracy for each individual DNA read [18]. In addition, the length of a single read is equal to **7000dp** which is one of the longest available. On the other hand, it is also significantly more expensive than other Sequencing methods with a cost of **$3** per test [54].

2.4.3 Sequencing by Synthesis

Sequencing by Synthesis (also known as *Illumina Sequencing* [9]) is by far the most commercially used DNA Sequencing technique [19]. It was initially developed by Shankar Balasubramanian and David Klenerman, two professors at the University of Cambridge, who were using fluorescently-labeled nucleotides to study the motion of *polymerase* at a single molecule level. *Polymerase* is an enzyme that helps the creation of new DNA molecules from the four nucleotide bases (**G**, **A**, **C**, **T**) by having the existing DNA as a template [70]. The success of this method led them to investigate whether this approach could be used for Genetic Sequencing and, eventually, theorise their findings [19]. They made an attempt to commercialise their solution by establishing a new company

named *Solexa* in 1998. Almost 10 years later, in 2007, Illumina acquired Solexa [19] and made its name a synonym for Sequencing by Synthesis.

In order to perform this sequencing technique, it is essential to break the DNA into smaller, more manageable fragments. These are usually consisted of 200 to 600 base pairs. They are attached to some small sequences of DNA which are called *adaptors*. To become single-stranded, these fragments are denatured by being incubated with *Sodium Hydroxide* (**NaOH**). As soon as this process is done, the fragments are washed across the flowcell to cause the complementary DNA binding to *primers* on the surface. Any remaining DNA that is not attached anywhere is removed. The portions of DNA that were indeed attached to the flowcell are replicated so that small clusters of identical DNA are built. At the process of Sequencing, every cluster would emit a signal that can be detected by a camera. Later, a number of unlabeled nucleotide bases and DNA polymerase are appended to increase the length and join the strands of DNA. The resulted double-stranded DNA is split into single-stranded DNA with the use heat. This creates a huge number of identical clusters of DNA sequences [70].

Primers as well as fluorescently-labeled *terminators* that stop the DNA synthesis are added to the flowcell where the former are attached to the DNA that is being sequenced. The aforementioned *DNA polymerase* binds to the *primer* and attaches the *terminator* to a new DNA strand where it prevents the addition of any extra bases to it [70].

Lasers scan the flowcell to activate the fluorescent label. The fluorescence substance is captured by a camera and saved on a computer. The four bases can be distinguished because their corresponding *terminator* emits different colour. Finally, the *terminator group* is removed from the first base and the next one is appended alongside. This process is repeated multiple times until the point where all of the clusters have been sequenced successfully. The generated sequence can be aligned to a reference sequence that search for any modifications or matches in the DNA [70].

2.4.4 Ion Torrent Sequencing

One of the most recently discovered Sequencing Technologies is the *Ion Torrent Sequencing* (also known as *Ion Semiconductor Sequencing*). It was discovered by Ion Torrent Systems Inc. in 2010 [2]. Unlike other Genetic Sequencing techniques, this method does not make use of optical signals at all. On the contrary, it is built on the fact that when **dNTP** is added to a polymer of DNA, *Hydrogen* cations ($\mathbf{H^+}$) are released [14].

This technique is based on the translation of chemically encoded information (**G**, **A**, **C**, **T**) into digital information (**0**, **1**) using a *semiconductor* chip. A *semiconductor* is a solid chemical element that enables the conduction of electricity under specific conditions but not in others. It is extremely useful for controlling the flow of electrical current [60].

As it holds with other DNA Sequencing techniques, the input DNA is fragmented and some *adaptors* are attached to each fragment. Then, each molecule is placed onto a *bead* and amplified by the application of emulsion *Polymerase Chain Reaction* (**PCR**). *Polymerase Chain Reaction* is a fast and costless molecular technique to *amplify* (copy) relatively short segments of DNA [66]. The emulsion version of this method uses tiny water bubbles which float on an oil solution. Afterwards, each one of these beads is placed onto a single well of a slide. Later, this slide is flooded into a mixture of a single species of **dNTP**, *polymerase* and *buffers*. Each $\mathbf{H^+}$ decreases the value of the **ph** and therefore it makes possible to determine whether any bases were added to the sequence during each cycle and construct the values of the DNA sequence read.

Chapter 3

Underlying Algorithms and Technologies

The Sequence Classification tool that was developed by the Wellcome Trust Centre for Molecular Parasitology used some well-established algorithms to support the different functionalities of the tool. These algorithms were traditionally applied for Sequence Classification purposes and are mostly related to String Manipulation.

Apart from these algorithms, the tool was built using various toolkits developed to support scientific projects. These technologies are widely available from the *Python Package Index* (**PyPI**) and they can all be downloaded by running the **pip** command either plainly or with elevated privileges using the **sudo** command:

```
$ pip install toolkit-name
```

```
$ sudo pip install toolkit-name
```

These toolkits and algorithms will be presented thoroughly in the next sections. Understanding what each of them does, gives a very good and detailed insight on the functionalities of the tool themselves.

3.1 Algorithms

3.1.1 k-mer

The term *k-mer* is very frequently used within the fields of Computational Genomics and Computational Linguistics. It refers to all the substrings of length **k** that can be possibly contained within a string. In Computational Genomics, it is used to describe a DNA sequence of small length that is consisted of bases of a fixed number (**k**).

The mathematical origins of *k-mer* trace back to the concept of *n-grams*. The notion of *n-grams* is also very widely used in Computational Linguistics. According to Jurafsky and Martin (2014), *"an **n-gram** is a sequence of **n** words"* [20]. As it can be concluded from this definition, a *k-mer* is just special case of an *n-gram*.

In order to determine and generate all the possible *k-mers* of a single read, one must use the corresponding **k-mer** algorithm. This algorithm states that in order to produce all the possible **k-mers** of size **k** from a **dna** string of length **len(dna)**, one should iterate for **length-k+1** times. In each iteration **i**, the substring of the **dna** that is stored between the $\mathbf{i^{th}}$ and $\mathbf{i+k^{th}}$ indices of the original **dna** is appended to a dictionary of **kmers**. A Python implementation of the **kmer** algorithm can be shown in the figure below:

```
def kmer(dna, k):
        kmers = {}
        length = len(dna)

        for i in range(0, length-k+1):
            kmers.append(dna[i:i+k])

        return kmers
```

Figure 3.1: A Python implementation of the **k-mer** algorithm

In the context of this project, kmerization is used to produce all the possible **k-mers** of size **k** from the input DNA sequence that is consisted of the four nucleic bases (**G**, **A**, **C**, **T**). The number of occurrences for all the generated kmers will be, then, counted by the script in order to be used for the classification process.

3.2 Technologies

3.2.1 Scientific Python

According to its documentation, Scientific Python (abbreviated as *SciPy*) is *"an ecosystem of open-source software for mathematics, science and engineering"* [51]. It was designed to provide a stack of useful tools and technologies for people who wish to use Python for Scientific Computing purposes.

One of the fundamental packages that comprises this module is *NumPy*. It supports numerical computation for mathematical topics such as Linear Algebra and Fourier Transformations. *NumPy* provides a powerful and sophisticated Base N-dimensional array object that can serve as an multi-dimensional container of generic data [27] and facilitate the manipulation of large arrays and numerical matrices [53].

Another extremely useful library that is part of the *SciPy* package set is *pandas*. *pandas* managed to combine both easiness and high-performance for the implementation of data structures and data analysis tools [33]. The two new data structures that are introduced are **Series** and **DataFrame**. The former is a single-dimensional object that it is very similar to a column of a table or an array. The latter is a multi-dimensional data structure consisted of rows and columns. In a higher level, a **DataFrame** is a collection of numerous **Series** [45].

Apart from the above, *SciPy* also offers a series of other useful libraries and tools targeted for scientific projects. *Matplotlib* is an enhanced tool used for the creation of quality 2D plots in a variety of formats [23] and designs. In order to support Symbolic Mathematics, *SciPy* also includes *SymPy* which is capable of providing the appropriate scientific notation for both simple and complex operations in Mathematical context [58]. Last but not least amongst the core packages provided by *SciPy*, *IPython* is a console for interactive computing that can be used for Parallel Computing [59].

Due to all the aforementioned features, *SciPy* is very often used as a Dependency in projects that require extensive and complex operations. Because of its ubiquity in the scientific community, *SciPy* has a large number of active contributors that are constantly working on the enhancement and extension of the software so that it supports even more aspects of Scientific Computation.

3.2.2 scikit-learn

scikit-learn is an open-source Python library for Machine Learning designed to provide a wide range of either supervised or unsupervised learning algorithms. It was firstly developed by David Cournapeu and it is currently maintained by more than 30 active contributors.

The *scikit-learn* project is build upon *SciPy* and it acts as an extension of the aforementioned module to support Machine Learning. It is a very powerful and robust tool designed to be used directly in production systems.

Although its interface is built on Python, it uses some technologies based on C that take advantage of the high compilation and execution speed of such lower level languages in order to boost up the performance of the toolkit whenever it is necessary. A common example of such technologies is *C-Libraries* that are used to support the operation of *NumPy* arrays. Also, *Cython* which is based on C acts as an extension of the Python Programming Language [13] and it is widely used by the most *scikit-learn* models due to its optimal speed.

scikit-learn is developed to support several aspects of Machine Learning. Firstly, it provides the appropriate infrastructure that can perform the **Classification** of an object in the sense that it can identify the category to which it belongs from a list of different categories as well as the automatic grouping of similar objects into sets (**Clustering**). Additionally, it supports **Regression** which allows the statistical analysis of a property in order to predict its values with respect to a variable that is strongly related and associated with it in the context of a continuous function. Additionally, it prepares the input data for use for Machine Learning purposes by applying Feature Extraction and Normalisation (**Preprocessing**) Last but not least, it provides tools that support some statistical operations such as the reduction of the number of dimensions involved in a specific problem in order to decrease the number of random variables that interfere and, subsequently, eliminate their effect on the final results (**Dimensionality Reduction**) as well as a form of parameter tuning for comparing, assessing and selecting between different parameters and models (**Model Selection**) [49].

scikit-learn is continuously growing to provide even more functionalities for use with projects that involve the highly emerging techniques of Machine Learning. It is also very well-documented and accompanied by an in-depth guide for its use through some of its most common application contexts.

The **scikit-learn** is a very important module for the operation of the Origin Classification tool. It provides the Machine Learning infrastructure that performs the classification of the different samples through its *Support Vector Machine* (**SVM**).

3.2.3 SAMtools

The short DNA sequence read alignments that are used as input for the Classification process are stored in **SAM** files. **SAM** stands for *Sequence Alignment/Map* and the files that are associated with this extension are text-formatted files which use **TAB** as their delimiter. Their two-part structure is consisted of the **Header** and the **Alignment**. The **Header** section can be omitted whilst the **Alignment** Section is mandatory. What distinguishes the **Header** section lines from the ones in the **Alignment** section of a *SAM* file is the fact the former start with an @ character. The **Alignment** lines contain a number of fields with useful data about the reads associated with them. From these fields, 11 should always be filled and appear in the same order because they carry some very essential information about the alignment. In addition, there is a number of optional fields that can be either completed or not which enhance the alignment process [62].

BAM (abbreviation for *Binary Alignment/Map*) is simply the binary equivalent of **SAM** files. This means that while the **SAM** files are human-readable, **BAM** files are written in binary format. They are compressed using **BGZF** which is a special type of block compression which is based on the **gzip** format [62]. The fact that **BAM**

files are compressed makes them more suitable and efficient for use along with relevant sequencing software than their corresponding **SAM** files.

The two aforementioned file types are part of *SAMtools*. *SAMtools* is a collection of libraries that support the interaction with sequencing data that are characterised by high-throughput. In its turn, it is included in *HTSlib* package that is openly available for the developers.

In addition to the two aforementioned formats, *SAMtools* also provide a third, newer type named **CRAM**. **CRAM** files are also written in binary format but they have some major advantages over the **BAM** type but still without losing their full compatibility with the files of the other format. The transition from **BAM** to **CRAM** is also effortless and quick. According to its specification, it offers lossless compression which is better than the one provided by the **BAM** format. Finally, **CRAM** provides support for the loss of **BAM** data in a controlled environment [48].

```
@HD VN:1.5 SO:coordinate
@SQ SN:ref LN:45
r001   99 ref  7 30 8M2I4M1D3M = 37  39 TTAGATAAAGGATACTG *
r002    0 ref  9 30 3S6M1P1I4M *  0   0 AAAAGATAAGGATA    *
r003    0 ref  9 30 5S6M       *  0   0 GCCTAAGCTAA       * SA:Z:ref,29,-,6H5M,17,0;
r004    0 ref 16 30 6M14N5M    *  0   0 ATAGCTTCAGC       *
r003 2064 ref 29 17 6H5M       *  0   0 TAGGC             * SA:Z:ref,9,+,5S6M,30,1;
r001  147 ref 37 30 9M         =  7 -39 CAGCGGCAT         * NM:i:1
```

Figure 3.2: An example of a **SAM** file

3.2.4 Pysam

Apart from the specification of the **SAM**, **BAM** and **CRAM** files, *Samtools* also provides *C-API*, a Python module for reading and processing them. The main disadvantage of this library is the fact that it is pretty heavyweight. In order to overcome the different hardships caused by the use of this package, a lighter wrapper of *C-API* was implemented. This module is called *Pysam* and it is protected by an MIT Licence [34].

According to its documentation, *Pysam* follows a very simple workflow for manipulating data stored in the previously mentioned file types. The user creates a new instance of the **AlignmentFile** object by giving the **BAM** file as an argument in order to open it. Then, he/she simply iterates over all the reads that correspond to a specific region using the **fetch()** method. Each subsequent iteration returns an **AlignedSegment** object that represents one read from the input file. Apart from reading from a **BAM** file, *Pysam* also provides a writer functionality that allows the writes on an **Alignment** file [35].

3.2.5 Amazon Web Services

The *Amazon Web Services* is a collection of reliable and scalable services that are intended to support Cloud Computing applications. They are hosted by Amazon and they were officially launched in 2006. Amazon Web Services are currently spread across 12 different geographical locations [7].

Some of the provided services are described below:

Amazon Elastic Compute Cloud (**EC2**) provides resizable computing capacity in the cloud. The flexible way that it was designed makes the process of cloud computing easier for the developers. It is possible to increase or reduce the used capacity to serve any changes in the requirements immediately. New server instances can be obtained and launched within minutes [4].

Amazon Simple Storage Service (**S3**) offers robust, highly-scalable and completely secure cloud storage. It offers three ranges of storage classes designed to suit different requirements. *Amazon S3 Standard* is built for general-purpose storage of data that are accessed frequently. *Amazon S3 Standard - Infrequent Access (Standard - IA)* is the best option for data that are intended to live for a long amount of time but without being accessed very frequently. Finally, *Amazon Glacier* aims to be used for long term archiving [6].

Amazon Elastic MapReduce (**EMR**) is a web-based service that supports the high-speed and cost-effective process of vast amount of data. It offers the appropriate infrastructure to support Big Data tasks by providing a modified version of the *Apace Hadoop* framework. It distributes large amounts of data across different **EC2** instances for processing. It is also possible to use alternative Big Data processing frameworks such as the *Apache Spark* and *Presto* [5].

Amazon DynamoDB is a No-SQL database which is extremely flexible in its use. It is suitable for applications that require high latency and consistence. It is a form of cloud database that support the use of both *key-value* and *document store* models [3].

The Origin Classification tool script is currently hosted on Amazon Web Services where it is benefited by some of the tools and technologies that are provided. For instance, it makes use of numerous **EC2** clusters to run the script as well as **S3** buckets to store the input files on.

Chapter 4

Current System Evaluation

The Origin Classification tool which is the system under investigation is written in Python (filename: **bedNbam-Classifier.py**) and it is used to classify input samples whose DNA sequence alignment data are stored in **BAM** files into either origins or non-origins for a specific given parasite with respect to the coordinates provided in two separate **BED** files.

In order to start proposing modifications that would optimise the execution of the script, the current system needed to be evaluated. The evaluation of a system can be done by assessing some of its different aspects and producing various metrics. Some qualities of a system that can be studied during this process are its performance and the design of its User Interface. The aspects that would eventually be investigated are determined by taking the objectives of the Evaluation into consideration.

For the purposes of this project, the most appropriate aspect to be assessed is current system's performance. Since there is an intention of scaling up the system, its Performance should be evaluated in order to prove whether it is capable of undergoing the aforementioned extension. This process is called Performance Testing.

According to Meier *et al.*, Performance Testing is the process that is concerned with the technical investigation that is conducted in order to determine the ability of a product to execute quickly, scale up and remain stable. It deals with the achievement of the desired response times, throughput and resource-utilisation levels as required by the objectives of the system under testing [24]. Most of the Performance Testing methods are based on the application of a specific load upon the system and the generation of metrics for the qualities under investigation. This load could be either data for input, users for concurrent access or something else relevant to the specific system's context.

The type of Performance Testing that is chosen to be used for evaluating the Sequence Classification Tool was *Load Testing*. It involves the application of a normal stress ("load") to the software in order to determine whether it will behave as it was expected to under ordinary conditions [55]. The load in this case was a set of **BED** files and a set of **BAM** files. The **BED** files contain tables of chromosome coordinates which are either origins or non-origins. The **BAM** files are consisted of sequence alignment data which are either used for training or testing purposes.

In addition to the selection of the appropriate type of Testing, the metrics that are going to be generated need to be decided as well. Whichever these metrics are, they are produced during a call of the specific process that is undergoing investigation. In order to be able to scale up the project, three important aspects of the software required assessment; the speed of execution, the number of operation calls and the memory usage. All the metrics collected at this stage will be used as a baseline for comparisons for all the optimised versions that would be presented in the following chapter.

4.1 Preparing the System Environment

To perform the experiments on the current system but also to try out the different optimisations in a later stage, an **Amazon Web Services EC2 cluster** was provided (**zoe.wtcmp.net**) as well as a user account (**zoeg**) with full administrative permissions and an environment free from any programs and other installations. The files that consisted the Origin Classification tool were migrated to the cluster as a directory using the **recursive scp** linux command that is shown in the following figure:

```
$ scp -r /local/path/to/the/folder/
 ↪  zoeg@zoe.wtcmp.net:/path/on/the/cluster/
```

Figure 4.1: The **recusrive scp** command used to send the tool files on the cluster

Before trying to run the script on the cluster, all of its dependencies needed to be installed on the cluster to guarantee that no problems would arise during its execution. These dependencies were:

- **Python (>= 2.6)**
- **scipy**
- **scikit-learn**
- **pysam 0.8.3**
- **pandas**
- **numpy** (required for **scikit-learn** and **pandas**)
- **biopython** 1.65

Figure 4.2: All the dependencies that are needed for the execution of the script.

The structure of the script command allows the configuration of certain parameters for its execution. Three files were needed as input. Two **BED** files named **Lmaj_origins.bed** and **Lmaj_nonorigins.bed** provided the origin and the non-origin coordinates respectively. These files are denoted on the script command with the flag **–classBed**. Additionally, a **BAM** file named **LmjFcombi.g2.bam** was used as training file for the purposes of the Machine Learning features provided by the script and declared with the **–testBam** flag of the script command. Some other parameters that can be configured by the script command are the name of the machine (**–machineOut**) that will be instantiated by the application of the Machine Learning process on the script, the length of **kmers** used in the process of kmerization (**–kmerSize**) and the number of CPU cores that will act as processing units during the execution of the script.

A minimal version of the command used to run the script is shown below:

```
$ python bedNbamClassifier.py --trainBam LmjFcombi.g2.bam --classBed
 ↪  Lmaj_origins.bed --classBed Lmaj_nonorigins.bed --machineOut LmjF_run1
 ↪  --kmerSize 6 --cpu 2
```

Figure 4.3: A minimal version of the bash command used to run the **Origin Classification tool**

4.2 Deterministic Profiling

The concept of Deterministic Profiling of a program is based on the fact that every event that happens during its execution including function calls, function returns and exception throwing is being monitored on the background and the accurate timings are measured for the intervals between these incidents [37]. This type of Profiling deals with providing the developer with all the related metrics; from precise values for the timing to call count statistics.

The measuring of **timing** is of particular importance since it can be source of useful conclusions regarding the performance of a script. Since time as a metric is strongly correlated with whether a software solution is considered to be efficient or not, being able to collect precise timings for each portion of the code and determine whether it is costly for the execution of the program can be very beneficial.

In addition to the time, another important metric for software delopment is **call count**. It calculates how many times a particular operation is repeated amongst the same block of code. This measurement can be proved to be very useful for identifying any bugs in the code by spotting any unexpected call counts. Furthermore, it can be used to find any **hot loops** that will require extra, careful optimisation in order not to give any extra overhead to the execution of the program.

Python provides a very extensive support for the Deterministic profiling of its scripts. It offers two very powerful built-in modules that can take up this type of tasks and return detailed reports about the performance. The first one is called **cProfile** and it is the one that is recommended by the official Python Documentation itself since the overhead that it adds to the execution of the program it profiles is fairly reasonable. This is why it is capable of undertaking the profiling of long-running programs. The optimal performance of this module can be explained by the fact that it is actually an extension built in the C programming language and therefore it does not add any extra load to the compiler for its interpretation. The latter module is **profile**, a pure Python module. Even though its interface is actually immitated by cProfile, the module itself is not very actively used since it adds a significant overhead to the execution of a program. An advantage of this method over the previous one is the fact that it can be extended in a very easy way. Additionally, there is a third module that is responsible for Python profiling called **hotshot**. It was initially developed to minimise the overhead in a higher degree but due to a critical bug identified in the timing core, it was completely abandoned after a while. It is currently not maintained and one of the most possible scenarios for its future is that it is going to be dropped out completely at some point [43].

For the purposes of this project, the module that was chosen to be used for evaluating the performance of the original script was **cProfile**. No changes on the script were necessary in order to use this Profiler since it is included in the Python language pack by default. A cProfile report presents a number of different information and statistics regarding the script it profiles.

cProfile also provides the option of sorting the output with respect to one or more of the aforementioned statistics. This feature was exploited in order to determine which operations were the most costly in terms of timing (**execution time**) and frequency (**call count**). These two criteria corresponded to '**cumtime**' and '**calls**' listed in the table below.

Valid Flag Argument	Meaning	Valid Flag Argument	Meaning
'calls'	call count	'line'	line number
'cumtime'	cumulative time	'name'	function name
'filename'	file name	'nfl'	name/file/line
'module'	file name	'stdname'	standard name
'pcalls'	primitive call count	'tottime'	internal time

Figure 4.4: The different statistics provided by **cProfile** in the form of keys to be used for sorting its output [43]

4.2.1 Total Execution Time

The execution time of a program can affected by several different factors. These factors are either program-related or they originate from the machine that the code runs on.

One of these factors is the programming language on which a program is written on. The selection of the most appropriate programming language for a particular purpose is a vital issue in software development. Concepts such as performance, scalability and portability should be taken into consideration when a software engineer chooses a programming language for a specific task. There is a large number of trade-offs amongst the different languages and one should be aware of what he or she is sacrificing by selecting a specific language for his/her project. From the previously mentioned qualities of a programming language, the one that affects the execution time of a program the most is performance. Therefore, for achieving small execution time, one should prefer a language that guarantees high performance over any other characteristics.

Another important issue that affects the execution of a program is the machine it runs on. It is surprising how different the execution time for the same program may be between different machines depending on their specifications. For example, the processing unit can affect the execution of a program to a great extend since the number of cores it possesses determines the level of threading and multi-tasking it allows. Additionally, the speed of the processor on a specific machine affects the time a single operation takes to execute.

Finally, an essential aspect that controls the speed with which a program executes is the complexity of the algorithm that it is built on. The efficiency of the algorithm can control how quickly or how slowly a program will run. This final factor is the one that provides the most freedom and the largest number of opportunities to the developer to leverage it since it is solely affected by his/her judgment. This is the property that the Deterministic Profiler uses to evaluate the execution performance of a program. It calculates the time intervals and counts the number of calls during the runtime; features that are mostly dependent on the nature of the algorithm in use.

Despite the aforementioned known qualities that control the execution performance of a program, the element of randomness still persists. Even when running the same program (same programming language and underlying algorithm) in the same machine, it is highly probable and almost expected to measure different execution times amongst several, consecutive runs. Therefore, it can be said that the timing of the execution of a program has a **stochastic** behaviour since it involves the random element that makes the precise prediction of the actual execution time almost impossible, no matter how many details are currently known about its structure and nature [10]. This variation is extremely difficult to be accurately explained because it usually originates from certain conditions that are formed during an individual execution of a program and it cannot be repeated again. Due to their unknown origins, one cannot really tackle the problem of random interference. But still, it is feasible to mitigate its influence on the metrics that are collected during Profiling by repeating the process multiple times and study the aggregated results rather than the individual ones.

This is the approach that was used for profiling the Origin Classifier Tool. A bash script was written that was configured to repeat the Deterministic Profiling on the program for 20 iterations and store the results on separate files marked by the number of the corresponding iteration within the same results folder.

```
$ for i in {1..20}; do python -m cProfile -s cumtime bedNbamClassifier.py
 ↪ --trainBam LmjFcombi.g2.bam --classBed Lmaj_origins.bed --classBed
 ↪ Lmaj_nonorigins.bed --machineOut LmjF_run1 --kmerSize 6 --cpu 2 >
 ↪ <results_folder>/<results file name>_run$i.txt 2>&1
```

Figure 4.5: A bash script that initiates 20 consecutive runs of the Deterministic Profiler with the **'cumtime'** sorting flag.

Run	Execution Time	Run	Execution Time
1	21.982	11	21.783
2	21.741	12	21.684
3	21.696	13	21.915
4	21.824	14	22.113
5	21.574	15	21.689
6	21.749	16	21.809
7	21.698	17	21.846
8	21.634	18	21.887
9	21.868	19	21.693
10	21.621	20	21.711

Figure 4.6: The execution times (in seconds) for the 20 consecutive times for which the Deterministic Profiler was run on the Original script.

To draw some conclusions regarding the performance of the system in general, it was required to produce some aggregated results for the above measurements. Therefore, the **Mean Value** and the **Standard Deviation** were calculated. Additionally, to study the behaviour of the script in terms of individual operations, the report file for one of the performed runs ought to be selected to be studied further. In order to find the most representative one amongst the all the generated files, the absolute difference in the execution time between each individual run and the mean value of the execution time was calculated and the run whose difference was the smallest was chosen to be used as the baseline of the study. The results of the previous process can be seen in the table below:

	Baseline System
Mean Value	21.7759
Standard Deviation	0.1320
Most representative performance	21.7830
Run with the most representative performance	11

Figure 4.7: The aggregated results of the execution time (in seconds) from the Deterministic Profiling of the Original script.

A combination of all the aforementioned results was used to plot the a column chart that presents the execution time for all the different runs along with a straight line that represents the **Mean Value** for the execution time as it calculated above. The unit used to measure the execution time of each run on the graph is **seconds** since it is the one that was originally used by **cProfile** itself.

As it can be seen on the graph, most of the runs had a better performance than the average. The **Mean Value** became higher though due to the existence of two runs (**Run 1 and Run 14**) whose execution took significantly longer time than the others. A possible explanation for the existence of delays on these two particular runs may have been the presence of random system factors. For example, a process that may have run on the background during the execution of these two runs may have reduced the processing resources available for them and they were forced to slow down. But these processes do not usually last for long and that is why they did not affect the rest of the runs. By taking all the values into consideration, it is safe to say that their distribution is normal since each individual value is no further than **3 times** the value of **Standard Deviation** from the calculated **Mean Value** [21]. Even the two outliers that were mentioned above fall within this interval.

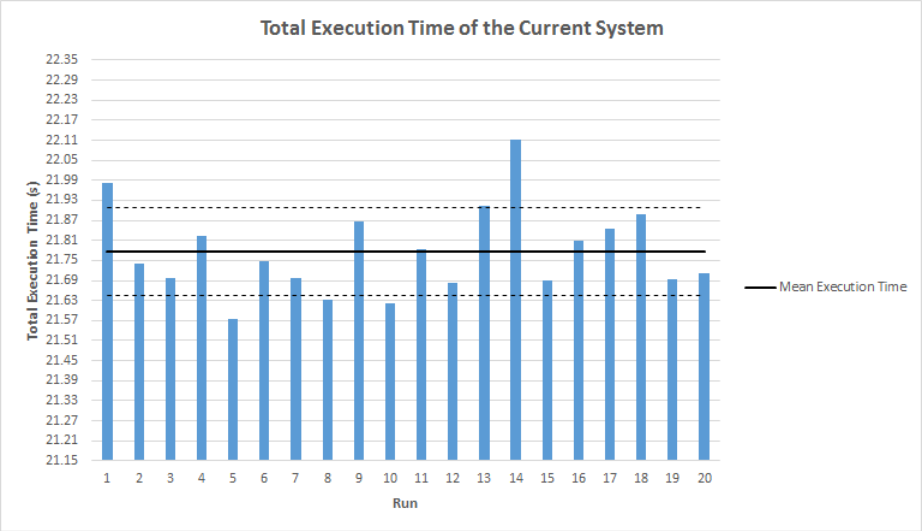

Figure 4.8: A Graph that shows the Total Execution Time (in seconds) of the Original System for 20 Consecutive Runs

Now that the overall properties and distribution of values were analysed and the appropriate conclusions were drawn regarding them, it was also important to study the metrics collected from an individual run. Since, as it was observed above, the most representative amongst the runs appeared to be the **11th** one, its generated report was used for further investigation. A shorter version of this report can be found in the appendix section.

As it can be seen from the report, the operations that have the largest total cumulative execution time is **kmerizeAlignments** and **kmerize()** functions. If one takes a closer look at the structure of the **kmerizeAlignments()** that is provided in the following figure, he/she can notice that it is consisted of a **for each** loop that calls the **kmerize()** multiple times. Therefore, it can be deduced that one of the possible reasons for which this function takes this significant amount of time to be executed is the multiple invocation of **kmerize()**. Apart from this, the delay may also originate from the String Concatenation operation that is present in **Line 5** which, as part of the loop, is repeated multiple times as well. String Concatenation might be cause of delay for the **kmerize()** function itself (**Line 6**) since it is also performed multiple times inside this function.

```
1  def kmerizeAlignments(alignmentsIterator, k):
2      kmerizedText = ""
3      for alignment in alignmentsIterator:
4          kmerData = kmerize(alignment.query_sequence,k)
5          kmerizedText = kmerizedText + kmerData + '\n'
6      return kmerizedText
```

Figure 4.9: The **kmerizeAlignments()** function

```
def kmerize(dna, k):
    thisString = ""
    for i in range(0, len(dna)+1-k, k):
        start = max(0, i)
        stop = min(len(dna), i + k) + 1
        thisString = thisString + dna[start:stop] + " "
    return thisString
```

Figure 4.10: The **kmerize()** function

4.2.2 Operation Call Count

Another important deterministic metric that should be used in order to evaluate the performance of the existing system is the number of times that each operation is called within the program. This number denotes the frequency of each operation and illustrates the amount of overhead that each one of them adds to the execution of the program.

In contrast with the measurement of execution time, the process of counting the number of operation calls within a program is not a stochastic process. No matter how many times a program is run or the existence of any random interference, if there is no error in the code itself that prevents it from running properly, the final count of the operation calls should be the same. Therefore, the collection of results for this specific metric required only a single run that was accomplished using the following bash script:

```
$ python -m cProfile -s calls bedNbamClassifier.py --trainBam
 ↪ LmjFcombi.g2.bam --classBed Lmaj_origins.bed --classBed
 ↪ Lmaj_nonorigins.bed --machineOut LmjF_run1 --kmerSize 6 --cpu 2 >
 ↪ <results_folder>/<results file name>_call_run.txt 2>&1
```

Figure 4.11: A bash script that initiates a single run of the Deterministic Profiler with the **'calls'** sorting flag.

The final report (a part of it is provided in the Appendix) states that the most common amongst the different operations present in the script are the **min()** and **max()** functions. These are two built-in functions that are always available by Python. As it can be seen in the specification of **kmerize()** that is provided above, these two methods are called multiple times within this function and the effects of their multiple invocation aggregated may add some important additional delay to the execution of the whole script. Furthermore, the third most called function is the **append()** from the Python **array** object. Due to the fact that the appending of an element to an array does not occur in-place but it actually creates a new array, a chain of significant, undesired side effects on the execution of the system may be caused that could include delays on the execution of the program and additional memory usage.

4.3 Memory Profiling

Apart from the deterministic profiling of a system, there is another type of profiling that could be proved to be of particular interest for the evaluation of a system. The Memory Profiling of a program monitors the consumption of memory by its different operations [42]. It can easily determine which of them are memory-exhaustive by producing a precise metric for the total memory they use during runtime.

The Memory Profiler that will be used for the evaluation the Origin Classification Tool is **memory_profiler**. It is a pure Python module that does line by line analysis of a script [42] and generates the appropriate memory consumption report in **Bibytes**. Its one (and only) highly recommended dependency is **psutil** which can be imported at the top of the script body as it is shown below:

```
import psutil
```

The **memory_profiler** module itself is available from the Python Package Index (*PyPI*) and can be downloaded using the **pip** (alone or accompanied by the **sudo** command for declaring system administrator permissions):

```
$ (sudo) pip install toolkit-name
```

This profiler cannot assess the whole script in one go but it can profile one individual function at a time. Therefore, for the case of the Origin Classification tool, only the **kmerize()** and **kmerizeAlignments()** functions can be evaluated at two different runs. In the script body, the function that is going to be profiled should import the module and be marked with the corresponding **@profile** decorator. The modifications that should be made to each of the aforementioned functions before running the Profiler are shown below:

```
from memory_profiler import profile
@profile
def kmerizeAlignments(alignmentsIterator, k):
    kmerizedList = ""
    for alignment in alignmentsIterator:
        kmerData = kmerize(alignment.query_sequence,k)
        kmerizedList = kmerizedList + kmerData + '\n'
    return kmerizedList
```

Figure 4.12: The appropriate additions to the **kmerize()** function in order to invoke the **memory_profiler**.

```
from memory_profiler import profile
@profile
def kmerize(dna, k):
    thisString = ""
    for i in range(0, len(dna)+1-k, k):
        start = max(0, i)
        stop = min(len(dna), i + k) + 1
        thisString = thisString + dna[start:stop] + " "
    return thisString
```

Figure 4.13: The appropriate additions to the **kmerizeAlignments()** function in order to invoke the **memory_profiler**.

Since these two functions include **for each** loops, the total memory consumption can be seen only at the section of the report that is dedicated to their last iteration. The corresponding sections from the two generated reports can be seen below:

```
Filename: bedNbamClassifier.py

Line # Mem usage Increment Line Contents
================================================
   115 104.5 MiB 0.0 MiB @profile
   116 def kmerize(dna, k):
   117 104.5 MiB 0.0 MiB thisString = ""
   118 104.5 MiB 0.0 MiB for i in range(0, len(dna)+1-k, k):
   119 104.5 MiB 0.0 MiB start = max(0, i)
   120 104.5 MiB 0.0 MiB stop = min(len(dna), i + k) + 1
   121 104.5 MiB 0.0 MiB thisString = thisString + dna[start:stop] + " "
   122 104.5 MiB 0.0 MiB return thisString
```

Figure 4.14: The section of the report generated by the **memory_profiler** for the **kmerize()** function that refers to its last iteration.

```
Filename: bedNbamClassifier.py

Line # Mem usage Increment Line Contents
================================================
   126 104.8 MiB 0.0 MiB @profile
   127 def kmerizeAlignments(alignmentsIterator, k):
   128 104.8 MiB 0.0 MiB kmerizedList = ""
   129 104.8 MiB 0.0 MiB for alignment in alignmentsIterator:
   130 104.8 MiB 0.0 MiB kmerData = kmerize(alignment.query_sequence,k)
   131 104.8 MiB 0.0 MiB kmerizedList = kmerizedList + kmerData + '\n'
   132 104.8 MiB 0.0 MiB return kmerizedList
```

Figure 4.15: The section of the report generated by the **memory_profiler** for the **kmerizeAlignments()** function that refers to its last iteration.

As it can be noticed on the figures provided above, the two functions seem to consume a significant amount of memory to satisfy their resource needs. Therefore, they seem to be memory-exhaustive. Again, a possible explanation for this observation is the multiple String Concatenation that takes place in their body. Apart from the time delay, it is also capable of causing excessive memory withholding because of the continuous allocation/reallocation of memory to objects.

Chapter 5

Study of Possible Optimisations

After the evaluation of the current system was completed, the findings from this process were used to propose the appropriate optimisations that were believed to bring improvement in its performance to the greatest degree possible.

A careful process was followed for implementing and assessing all the optimisations that would be presented in the next sections. Firstly, they were applied directly on the original script by making sure that nothing irrelevant with them would be changed. Later, these optimisations were assessed by running the **cProfile** Deterministic Profiler against the new version of the script 20 times and collecting the values for the execution time of all the different runs. As it was discussed in the previous chapter, that repetition of the Profiling process was done in order to eliminated the random system interference on the results for as much as possible. Subsequently, some aggregated values such as the **Mean Value** and the **Standard Deviation** of the execution time for all the runs were calculated. A column chart that represented the execution time per run was plotted in a graph that also included their **Mean Value** as a bold straight line and the boundaries of **Standard Deviation** as two dashed lines above and below it. The graphical representation of the results was extremely useful in order to spot general patterns and common behaviours amongst the different runs. If the **Mean Value** of the execution time for the new Configuration showed some improvement over the corresponding value of the baseline system then the performance results of the system under investigation became subject of statistical analysis.

This analysis was consisted of various steps. Firstly, a **null hypothesis** was formed which claimed that "*the applied optimisation did not improve the performance of the system*". In parallel, the **alternative hypothesis** was also constructed which acted as the opposite of the null one. It claimed that "*the applied optimisation actually improved the performance of the system*". This hypothesis is considered as *one-tailed* since it implies that an expected result will be found (i.e. the proposed system will provide **better** performance than the baseline). Then, a *Paired Samples T-Test* was conducted to prove or disprove the significance of the results. A constant **a = 0.05** that represents the significance level was used for comparison with the value **p** which was the outcome of the T-Test. If the value of **p** was less than the constant **a** then the null hypothesis was **rejected** in favour of the alternative hypothesis. If this was the case then it was statistically proved that the results from the performance testing of the new system were significant and the proposed optimisation can be accepted as a viable solution.

An additional assessment of the optimisations was conducted to evaluate their correctness. In order to ensure that the modifications did not alter the output of the **kmerization** process, a few changes were made to the two versions of script (the original and the modified one) that wrote the produced **kmers** into two separate files. In their turn, the two generated files were compared using the **diff** linux command (as it can be seen in the figure below) with the purpose of having their similarity assessed. If these two files were identical (i.e. the **diff** command returned nothing) then it meant that the proposed modifications did not interfere somehow in the **kmerization** process resulting in any errors on its prescribed functionality.

```
$ diff kmerize_original_output kmerize_optimisation_name_output
```

Figure 5.1: The syntax of the **diff** linux command used for comparing the outputs of **kmerization** for the original and the proposed version of the script

Finally, the memory usage of the functions on which the different modifications were applied was monitored with the use of the **memory_profiler** that was introduced in the previous chapter. The optimisation under investigation should be proved efficient both in terms of execution time and memory usage, therefore the latter factor must be assessed as well. In case a possible optimisation provides high performance but also uses an excessive amount of memory, second thoughts should be made before adapting this alternative due to the fact that the machine on which the script will run may not be capable of executing it.

5.1 String Immutability

As it was observed at the current system evaluation stage, one of the operations that seemed to be the most time- and memory- consuming for the Origin Classification script was String Concatenation. It was also one of the operations that were repeated more often, therefore it added a very significant amount of overhead to the execution of the program.

String Concatenation is the process of joining two or more strings into a new string that is consisted of the two string slices combined together [36]. This pattern and technique is very frequent in software development since a lot of applications deal with String Manipulation in general. However, it is considered to be a very costly operation due to the *String Immutability*.

In Programming, an *Immutable Object* is an object whose state cannot be modified after it is constructed. This means that in order to change the value of an immutable object, one must create a new object instead of updating the existing one in-place [28]. These actions are not instructed explicitly by the programmer but they are still performed in the background; in bytecode level. The operation of creating a new object can be costly, especially if it is repeated multiple times during the execution of a program.

String Immutability is a common property between a large number of different programming languages. Python is one of them. Since the current stable version of the Origin Classification tool is written in Python 2, the issue with String Immutability persists and it is fairly acute as the results from the current system evaluation suggest.

A study of various possible alternatives have been made so that the best one would be eventually selected to replace the simple String Concatenation in the script. Each of them was introduced to the program and ran against the same data set to produce the metrics of interest that were described earlier.

5.1.1 String List

Amongst the most commonly used data structures within Python's programming environment is *List*. *List* is defined as a container of objects that stores the given elements in a specific order. It is member of the *Python Sequence Types* along with String literals, Unicode Objects, Tuples, Bytearrays and xrange objects [39]. Python Lists are pretty versatile since they are capable of keeping elements of different type together in the same list object without any problems [64].

A Python List can be represented as a list of values which are separated by commas and enclosed in square brackets, as it can be seen in the examples below:

```
list1 = [1, 2, 3, 4]
list2 = ["jan", "feb", "mar", "apr"]
list3 = ['m', 't', 'w', 'f']
list4 = ["jan", 1, "feb", 2, "mar", 3, "apr", 4]
```

Figure 5.2: Some examples of Python Lists that illustrate the versatility of Python Lists

Each element in a Sequence Type such as the Python List is assigned an integer number which denotes its position. This number is called **index** and can take values from **0** to **N-1** where **N** is the number of elements stored in the List. These indices are used to access the elements of the list, either individually or as list slices, as it is shown below:

```
print(list1[0])
>> 1
print(list2[1:3])
>> ["feb", "mar"]
print(list3[1:])
>> ['t', 'w', 'f']
print(list[:3])
>> ["jan", 1, "feb"]
```

Figure 5.3: Some examples of accessing elements in Python Lists

It should be noted that in the case of list slices, the index range used is inclusive on the left boundary and exclusive on the right. This means that the element that corresponds to the first index is returned whilst the element stored in the second index is not.

Due to all this flexibility, Python Lists were considered to be a good alternative to the naive String Appending applied on the original version of the script. Each **kmer** produced in the **kmerize** function instead of being appended to a huge string of **kmers** that are separated by a single whitespace character (**“ ”**) was added to a Python String List instead as a separate element. In the following subsections, two different applications of these scenario on the tool script will be discussed, performed and evaluated. The difference between these two approaches is the way they handle the appending of an element to the list and the joining of two Python Lists.

First Version of String List Optimisation

The first approach to the application of Python String Lists on the script provided that their join would be performed using the **plus (+) operator** that underlies the append of the former list to the other. This method is very similar to the Naive String Concatenation that was described earlier but in this case it involved the appending of two dynamic data structures instead of plain immutable strings.

An identical functionality to the one provided by the **plus (+) operator** is offered by the built-in **extend()** method. The former approach is preferred though, since in the bytecode level, the first technique involves only a single **INPLACE_ADD()** instruction whilst the second uses an extra function call.

The modifications applied to the script for the introduction of Python Lists can be shown below:

```
def kmerize(dna, k):
    thisStringList = [] # MODIFICATION
    for i in range(0, len(dna)+1-k, k):
        start = max(0, i)
        stop = min(len(dna), i + k) + 1
        thisStringList.append(dna[start:stop]) # MODIFICATION
    return thisStringList
```

Figure 5.4: The modifications that were made on the **kmerize()** function in order to accommodate the first approach to the application of String Lists.

```
def kmerizeAlignments(alignmentsIterator, k):
    kmerizedList = []
    for alignment in alignmentsIterator:
        kmerData = kmerize(alignment.query_sequence,k)
        kmerizedList = kmerizedList + kmerData # MODIFICATION
    return kmerizedList
```

Figure 5.5: The modifications that were made on the **kmerizeAlignments()** function in order to accommodate the first approach to the application of String Lists.

```
alignmentsIterator = thisBam.fetch(bedRecord['chr'], thisStart, thisEnd
 ↪ + 1)
trainingData = trainingData + kmerizeAlignments(alignmentsIterator,
 ↪ kmerSize) # MODIFICATION
target_labels.append(classesList.index(thisClass))
```

Figure 5.6: The modifications that were made on other sections of the script in order to accommodate the first approach to the application of String Lists.

The String Concatenation at the **kmerize()** function was replaced by a list append operation that added each **kmer** as a new string element to the list. Then, each list of **kmerData** that was created inside the **kmerizeAlignments()** function by iterating over the **alignmentsIterator** was appended to a larger list called **kmerizedList**. This process was repeated for every **BAM** file given as input. Each of these lists was appended to a universal list of all the **kmers** called **trainingData** using the **plus (+)** operator again.

The accuracy assessment of this technique that was described at the beginning of the chapter suggested that the **kmerization** process in this method produced the exact same results as the original script.

The implementation and assessment process described earlier was applied and the following results were collected and plotted:

Run	Execution Time	Run	Execution Time
1	57.439	11	57.554
2	57.419	12	57.346
3	57.473	13	57.658
4	57.461	14	57.474
5	57.575	15	57.444
6	57.439	16	57.481
7	57.826	17	57.462
8	57.598	18	57.400
9	57.399	19	57.376
10	57.335	20	57.378

Figure 5.7: The execution times (in seconds) for the 20 consecutive times for which the Deterministic Profiler was run on the script that adopted the first approach to the application of String Lists optimisation.

	Baseline System	String List - Version 1 System
Mean Value	21.7759	57.4769
Standard Deviation	0.1320	0.1174
Most representative performance	21.7830	57.4769
Run with the most representative performance	11	14

Figure 5.8: The aggregated results from the Deterministic Profiling of the script that adopted the first approach to the application of String Lists optimisation.

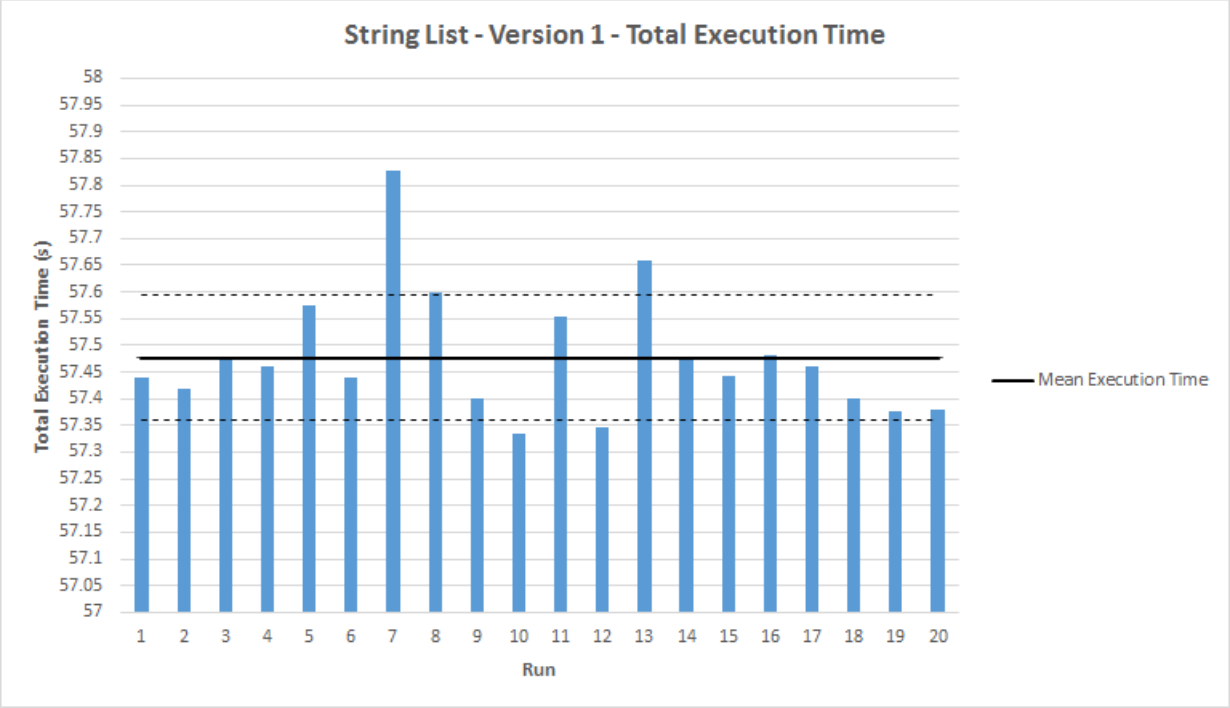

Figure 5.9: A Graph that shows the Total Execution Time (in seconds) of the System after the application of the first approach to the application of the String List optimisation for 20 Consecutive Runs.

The graph suggests that most of the runs had smaller execution time than the average; only **6** of them actually exceeded it. The rise in the **Mean Value** though can be explained by the fact that these **6** runs have significantly higher execution time than the others and so they managed to increase the **Mean Value** with their variance. From these, only **3** have exceeded the upper boundary that was set by the top line of **Standard Deviation** range.

```
Filename: bedNbamClassifier.py

Line # Mem usage Increment Line Contents
================================================
   113 347.230 MiB 0.000 MiB @profile
   114 def kmerize(dna, k):
   115 347.230 MiB 0.000 MiB thisStringList = []
   116 347.230 MiB 0.000 MiB for i in range(0, len(dna)+1-k, k):
   117 347.230 MiB 0.000 MiB start = max(0, i)
   118 347.230 MiB 0.000 MiB stop = min(len(dna), i + k) + 1
   119 347.230 MiB 0.000 MiB thisStringList.append(dna[start:stop])
   120 347.230 MiB 0.000 MiB return thisStringList
```

Figure 5.10: The section of the report generated by the **memory_profiler** for the **kmerize()** function of the System after the application of the first version of the Python List optimisation.

```
Filename: bedNbamClassifier.py

Line # Mem usage Increment Line Contents
================================================
   127 346.527 MiB 0.000 MiB @profile
   128 def kmerizeAlignments(alignmentsIterator, k):
   129 346.527 MiB 0.000 MiB kmerizedList = []
   130 347.230 MiB 0.703 MiB for alignment in alignmentsIterator:
   131 347.230 MiB 0.000 MiB kmerData = kmerize(alignment.query_sequence,k)
   132 347.230 MiB 0.000 MiB kmerizedList = kmerizedList + kmerData
   133 347.230 MiB 0.000 MiB return kmerizedList
```

Figure 5.11: The section of the report generated by the **memory_profiler** for the **kmerizeAlignments()** function of the System after the application of the first version of the Python List optimisation.

As it can be seen from the above results, the proposed modifications did not actually lead to any improvements on the execution of the script. On the contrary, they actually caused an extra delay which resulted to a total execution time which was almost **3** times larger than the execution time of the original script measured earlier. The same thing held for the memory usage since the results from the Memory Profiling showed that the amount of memory used by the modified script was also **3** times higher than the one of the original script. Therefore, it is safe to say these modification cannot be considered as an optimisation over the original script.

A possible explanation for the inefficiency of this method is the fact that the string list appending technique used is an operation that is analogous to the Concatenation of Strings but for Python Lists. Hence, the action of appending two lists using the **plus (+) operator** actually created a new list to insert the joined version of the two others. This imposed some negative implications in both the execution time of the script and its memory usage. When it comes to the cost of this operation when compared to the Naive String Appending technique, the former one probably used more time since it involved the initialisation and the memory allocation of whole a data structure rather than of a single string object.

Second Version of String List Optimisation

Since the reason why the first version of the Python List optimisation did not manage to improve the mean execution time of the script was the fact that the string appending using the **plus (+) operator** actually created a new list to insert the combined version of the two lists, it was necessary to find a way to append a Python list to another without creating a new list instance.

The second approach simply suggests the assignment of the second list to a list slice at the end of the first list. The list slice starts from the index which is equal to the length of the first list which basically means that the second list is assigned to a new position at the end of this list as it would be done if a new element was simply inserted to the first list. This approach seems to avoid all the overhead added to the execution of the program using the previous approach since it does not a create a new list from scratch.

An example that applies the described operation can be seen below:

```
l1 = [1,2]
l2 = [3,4]
l1[len(l1):] = l2

print(l1)
>> [1,2,3,4]
```

Figure 5.12: An example of appending a list to another without creating a new list

It is worth mentioning that this method is not the same as inserting the second list into a position at the end of the first list since that would store the entire list in a single position of the former list. Python would then recognise the second list as a single element added to the other list; a list stored along with some strings in a bigger list. This action is completely valid and acceptable in the context of Python. The difference between these two operations is illustrated below:

```
l1 = [1,2]
l2 = [3,4]
l3 = [5,6]
l1[len(l1):] = l3
l2[len(l2)] = l3

print(l1)
>> [1,2,5,6]

print(l2)
>> [3,4,[5,6]]
```

Figure 5.13: An example that illustrates the difference between inserting a list into another as a new element and inserting a list into another as a list slice

In order to introduce this approach to the script, the following modifications needed to take place:

```
def kmerize(dna, k):
    thisStringList = [] # MODIFICATION
    for i in range(0, len(dna)+1-k, k):
        start = max(0, i)
        stop = min(len(dna), i + k) + 1
        thisStringList.append(dna[start:stop]) # MODIFICATION
    return thisStringList # MODIFICATION
```

Figure 5.14: The modifications that were made on the **kmerize** method of the script in order to accommodate the second approach to the application of String Lists.

```
def kmerizeAlignments(alignmentsIterator, k):
    kmerizedList = [] # MODIFICATION
    for alignment in alignmentsIterator:
        kmerData = kmerize(alignment.query_sequence,k)
        kmerizedList[len(kmerizedList):] = kmerData # MODIFICATION
    return kmerizedList
```

Figure 5.15: The modifications that were made on the **kmerizeAlignments** method of the script in order to accommodate the second approach to the application of String Lists.

```
alignmentsIterator = thisBam.fetch(bedRecord['chr'], thisStart, thisEnd
 ↪  + 1)
trainingData[len(trainingData):] =
 ↪  kmerizeAlignments(alignmentsIterator, kmerSize) # MODIFICATION
target_labels.append(classesList.index(thisClass))
```

Figure 5.16: The modifications that were made on other sections of the script in order to accommodate the second approach to the application of String Lists.

As it can be noticed, the only thing that was different in the implementation of the second approach was the way the list appeding was performed.

The accuracy assessment of this technique proved that the **kmerization** process in this method produced the same results as the original script.

Again, the optimisation was evaluated using the usual method. The results are provided below in both textual and graphical form:

Run	Execution Time	Run	Execution Time
1	12.463	**11**	12.591
2	12.240	**12**	12.791
3	12.611	**13**	12.523
4	12.422	**14**	12.360
5	12.616	**15**	12.360
6	12.416	**16**	12.596
7	12.335	**17**	12.587
8	12.564	**18**	12.418
9	12.457	**19**	12.357
10	12.439	**20**	12.677

Figure 5.17: The execution times (in seconds) for the 20 consecutive times for which the Deterministic Profiler was run on the script that adopted the second approach to the application of String Lists optimisation.

	Baseline System	String List - Version 2 System
Mean Value	21.7759	12.4912
Standard Deviation	0.1320	0.1360
Most representative performance	21.7830	12.4912
Run with the most representative performance	11	1

Figure 5.18: The aggregated results from the Deterministic Profiling of the script that adopted the second approach to the application of String Lists optimisation.

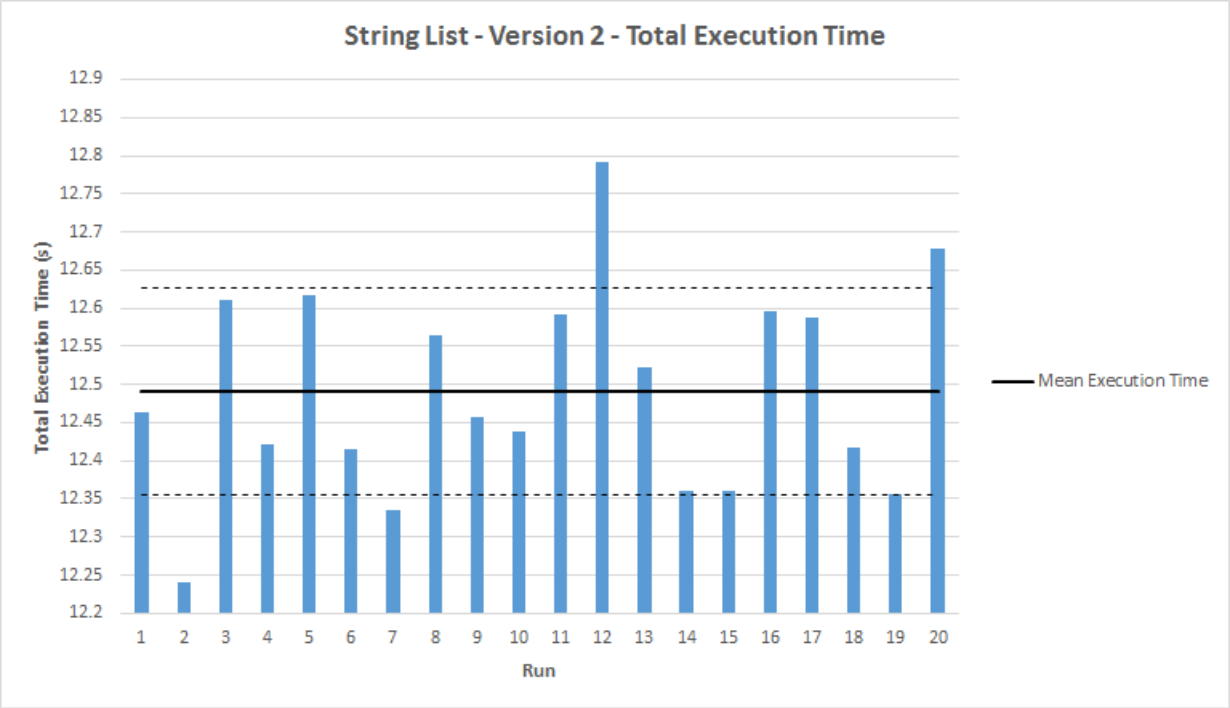

Figure 5.19: A Graph that shows the Total Execution Time (in seconds) of the System after the application of the second approach to the application of the String List for 20 Consecutive Runs.

The proposed optimisation has managed to reduce the **Mean Value** of the execution time to the **57%** of its original value. That means that it provided a significant improvement over the original system since it almost doubled the execution speed.

The process that was described at the beginning of this chapter was used to assess the significance of the previous observation. The **null hypothesis** that it needed to be disproved here was that replacing the string concatenation operations used in the script with this version of String Lists will not improve the performance of the script. On the contrary, the **alternative hypothesis** suggested that this change will actually make a positive impact on the performance of the script. A **Paired Samples T-Test** was performed and the calculated **p** value was equal to **$3x10^{-33}$**. This value is smaller than the constant **a = 0.05**, therefore the null hypothesis was **rejected** in favour of the alternative one. Therefore, the proposed optimisation improved the performance of the script and this impact was considered as statistically significant.

```
Filename: bedNbamClassifier.py

Line # Mem usage Increment Line Contents
================================================
   113 346.578 MiB 0.000 MiB @profile
   114 def kmerize(dna, k):
   115 346.578 MiB 0.000 MiB thisStringList = []
   116 346.578 MiB 0.000 MiB for i in range(0, len(dna)+1-k, k):
   117 346.578 MiB 0.000 MiB start = max(0, i)
   118 346.578 MiB 0.000 MiB stop = min(len(dna), i + k) + 1
   119 346.578 MiB 0.000 MiB thisStringList.append(dna[start:stop])
   120 346.578 MiB 0.000 MiB return thisStringList
```

Figure 5.20: The section of the report generated by the **memory_profiler** for the **kmerize()** function of the System after the application of the second version of the Python List optimisation.

```
Filename: bedNbamClassifier.py

Line # Mem usage Increment Line Contents
================================================
   127 345.012 MiB 0.000 MiB @profile
   128 def kmerizeAlignments(alignmentsIterator, k):
   129 345.012 MiB 0.000 MiB kmerizedList = []
   130 345.715 MiB 0.703 MiB for alignment in alignmentsIterator:
   131 345.715 MiB 0.000 MiB kmerData = kmerize(alignment.query_sequence,k)
   132 345.715 MiB 0.000 MiB kmerizedList[len(kmerizedList):] = kmerData
   133 345.715 MiB 0.000 MiB return kmerizedList
```

Figure 5.21: The section of the report generated by the **memory_profiler** for the **kmerizeAlignments()** function of the System after the application of the second version of the Python List optimisation.

Additionally, this approach to the use of String list was also more efficient in terms of memory usage from the first one since it managed to reduce the amount of memory used by almost **1MiB**, as it can be noticed in the results of Memory Profiling.

5.1.2 Mutable String Class

A few programming languages provide alternative types that act as mutable Strings. For example, Java provides *StringBuffer* [29] and *StringBuilder* [30] to overcome its String object immutability. Over the years, Python has offered its own alternatives to this issue.

The **Mutable String** Class was derived from the **User String** Class. It was built to redefine the String objects so that they would become mutable. This class was created for purely educational purposes so that it would act as an example for explaining the concept of inheritance in Python [40]. This was the reason why it was not suggested as a solution for actual programmatic tasks.

Even though it seemed to have mitigated the problem of String Immutability, the implementation of **MutableString()** class had a number of drawbacks associated with it. Instances of this class could not be used as keys for dictionaries since, by definition, dictionaries can accept only immutable types as their keys. In order to accomplish this, the **__hash__()** method was removed from their implementation so that any attempt to use this object as a key for a dictionary was rejected [40]. Additionally, some methods defined within this class behaved just like the corresponding methods applied on the normal Immutable Strings; they provided the return of new strings rather than an in-place update of the string as they were supposed to do.

As some related reading suggests, the Mutable String had no significant benefit to provide over the standard String even though one might think that it would. It did not improve the memory usage or the runtime performance at all. On the contrary, it was highly likely to reduce the performance of the system further [8].

Initially, the plan was to introduce the **MutableString()** on the script for experimental purposes. Unfortunately, this scenario needed to be abandoned due to the fact that the **MutableString** class has been deprecated since the release of the 2.6 version of Python and was removed completely in Python 3. The Origin Classification script uses Python 2.6 and therefore that made the application of this technique impossible [40].

Python 3 has a mutable String type to offer as well that is named **bytearray**. Sadly, it is not supported by the Python version that the script uses and therefore it could not be used for the purposes of this project.

5.1.3 Character Array

In Python, an **array** sequence type is used to represent an array of basic values. These values can be either integers, floating point numbers or characters. Its behaviour is very similar to a list but with a major difference: the objects stored in it should be of the same type. This type is declared when the object is created by passing a **type code** character as an argument to the array constructor.

The **array** type defines the **fromstring()** method which appends the characters from a given string to the character array. This functionality was be exploited in order to replace the Naive String Concatenation in the Origin Classification Script.

The modifications to be applied on the code are shown below:

```
from array import array # MODIFICATION
def kmerize(dna, k):
    thisCharArray = array('c')
    for i in range(0, len(dna)+1-k, k):
        start = max(0, i)
        stop = min(len(dna), i + k) + 1
        thisCharArray.fromstring(dna[start:stop]) # MODIFICATION
        thisCharArray.append('\n') # MODIFICATION
    return thisCharArray
```

Figure 5.22: The modifications that were made on the **kmerize()** function of the script in order to accommodate the application of Character Array.

```
def kmerizeAlignments(alignmentsIterator, k):
    kmerizedList = array('c')
    for alignment in alignmentsIterator:
        kmerData = kmerize(alignment.query_sequence,k)
        kmerizedList.extend(kmerData)
        kmerizedList.append('\n')
    return kmerizedList
```

Figure 5.23: The modifications that were made on the **kmerizeAlignments()** function of the script in order to accommodate the application of Character Array.

```
alignmentsIterator = thisBam.fetch(bedRecord['chr'], thisStart, thisEnd
 ↪ + 1)
trainingData[len(trainingData):] =
 ↪ kmerizeAlignments(alignmentsIterator, kmerSize).tostring().split()
 ↪ # MODIFICATION
target_labels.append(classesList.index(thisClass))
```

Figure 5.24: The modifications that were made on other sections of the script in order to accommodate the application of Character Array.

The String Concatenation at the **kmerize()** function was replaced by an array appending operation that took

every **kmer** as an argument and added each single character to the Character Array. Along with the characters of each **kmer**, an extra whitespace character was added in order to separate it from the next one. Then, each Character Array of **kmerData** that was created inside the **kmerizeAlignments()** function by iterating over the **alignmentsIterator** was added to a larger character array called **kmerizedList** using the **extend()** method. These process was repeated for every **BAM** file given as input. Each of these Character Arrays was converted into a string which was then split in order to produce a universal list of all the **kmers** called **trainingData**.

The accuracy of this method was assessed as well and it was proved to return the same output from the **kmerization** process as the baseline.

In order to assess the efficiency of the above modifications, the same process as the one used for the previous optimisations was used:

Run	Execution Time	Run	Execution Time
1	16.667	11	16.818
2	16.583	12	16.598
3	16.485	13	16.605
4	16.451	14	16.597
5	16.604	15	16.441
6	16.844	16	16.427
7	16.879	17	16.517
8	16.657	18	16.601
9	16.588	19	16.534
10	16.962	20	16.694

Figure 5.25: The execution times (in seconds) for the 20 consecutive times for which the Deterministic Profiler was run on the script that adopted the Character Array optimisation.

	Baseline System	Character Array System
Mean Value	21.7759	16.6276
Standard Deviation	0.1320	0.1486
Most representative performance	21.7830	16.6050
Run with the most representative performance	11	13

Figure 5.26: The aggregated results from the Deterministic Profiling of the script that adopted the Character Array optimisation.

By comparing the **Mean Values** of the baseline system and the system that uses the Character Array for String Concatenation, one can notice that the average execution time has been to reduced to almost **76%** of its original value. This was a fairly respectable improvement over the original system but it was not as large quantitatively as the second approach on the String List that was proposed in a previous subsection. This lack of efficiency in the Performance that was shown by the Character Array when it was compared to the String List can be explained by the different policy that it is followed by the Python List during the appending of new elements. As soon as all the allocated elements get used, the Python List allocates a few extra elements [57] and this speeds up the appending process.

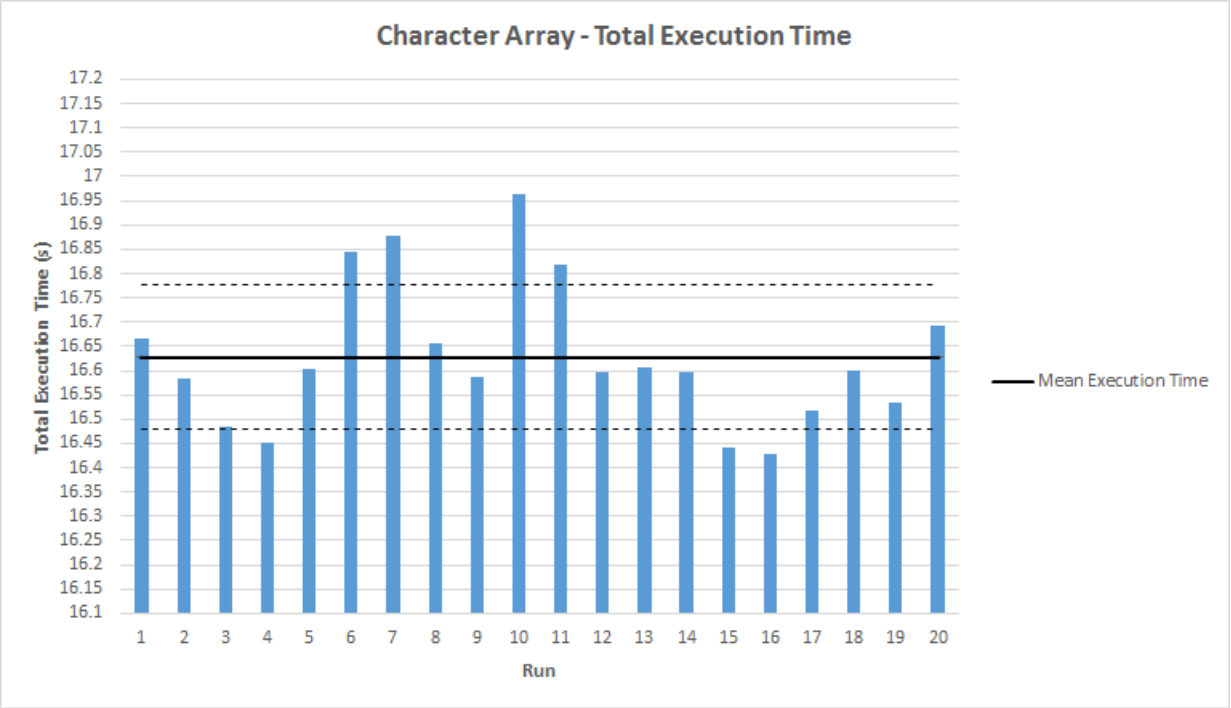

Figure 5.27: A Graph that shows the Total Execution Time (in seconds) of the System after the application of Character Array optimisation for 20 Consecutive Runs.

According to the graph, the distribution of the values of the execution time was slightly different this time. Even though most of the execution time values were located in the range denoted by the **Standard Deviation** boundaries, there are a few outlier runs that had either significantly larger execution time with respect to the **Mean Value** or significantly lower. However, this was not reflected on the **Mean Value** due to the fact the their different kind of variations seemed to counterbalance each other.

But before it is fully accepted, the usual statistical analysis process should have been repeated to prove its significance. The **null hypothesis** there was that the replacement of the plain String Concatenation with the Character Array optimisation did not improve the performance of the system and the **alternative hypothesis** was that it did. The **p-value** in this case was equal to $\mathbf{4.31x10^{-28}}$ which is smaller than the value of the constant **a**, therefore the null hypothesis was **rejected** and the alternative one was proved to be correct. Thus, the solution that provides the use of Character Arrays is acceptable as an optimisation over the original script.

```
Filename: bedNbamClassifier.py

Line # Mem usage Increment Line Contents
================================================
   114 345.262 MiB 0.000 MiB @profile
   115 def kmerize(dna, k):
   116 345.262 MiB 0.000 MiB thisCharArray = array('c')
   117 345.262 MiB 0.000 MiB for i in range(0, len(dna)+1-k, k):
   118 345.262 MiB 0.000 MiB start = max(0, i)
   119 345.262 MiB 0.000 MiB stop = min(len(dna), i + k) + 1
   120 345.262 MiB 0.000 MiB thisCharArray.fromstring(dna[start:stop])
   121 345.262 MiB 0.000 MiB thisCharArray.append('\n')
   122 345.262 MiB 0.000 MiB return thisCharArray
```

Figure 5.28: The section of the report generated by the **memory_profiler** for the **kmerize()** function of the System after the application of the Character Array optimisation.

```
Filename: bedNbamClassifier.py

Line # Mem usage Increment Line Contents
================================================
  129 345.637 MiB 0.000 MiB @profile
  130 def kmerizeAlignments(alignmentsIterator, k):
  131 345.637 MiB 0.000 MiB kmerizedList = array('c')
  132 345.637 MiB 0.000 MiB for alignment in alignmentsIterator:
  133 345.637 MiB 0.000 MiB kmerData = kmerize(alignment.query_sequence,k)
  134 345.637 MiB 0.000 MiB kmerizedList.extend(kmerData)
  135 345.637 MiB 0.000 MiB kmerizedList.append('\n')
  136 345.637 MiB 0.000 MiB return kmerizedList
```

Figure 5.29: The section of the report generated by the **memory_profiler** for the **kmerizeAlignments()** function of the System after the application of the Character Array optimisation.

The results from the Memory Profiling of the script suggested that this method made use of less memory than the two approaches on the use of String List. It reduced the total memory usage by **1MiB**. This is due to the fact that Python arrays use a memory allocation strategy which provides that their storage is done by allocating contiguous blocks of Memory. This strategy is more efficient memory-wise than the strategy used by the Python Lists which simple use pointers to different blocks where the contained objects are actually stored [57]. Also Python Lists are more memory-consuming because they allocate extra elements automatically, before there is an explicit need for them, and in case they are not eventually used, they just waste memory.

5.1.4 Pseudo File

As it was previously mentioned, Java Programming Language offers dynamic objects that act as a mutable versions of Strings. One of them is the **String Buffer**. Python provides an object type that is very similar to it named **cStringIO**. This module acts as a pseudo file in the sense that it reads from and writes to a so-called "Memory File" but without actually producing an output file. There is also a similar module called **StringIO** but the former one is preferred due to its speed and efficiency that makes it more suitable for use for heavier purposes.

It is worth saying that the contents of a String Buffer cannot be read directly but they would rather need to be extracted from the "Memory File" using its **getvalue()** method.

The modifications needed to be made on the script are the follows:

```
from cStringIO import StringIO # MODIFICATION
def kmerize(dna, k):
    thisStringIO = StringIO() # MODIFICATION
    for i in range(0, len(dna)+1-k, k):
        start = max(0, i)
        stop = min(len(dna), i + k) + 1
        thisStringIO.write(dna[start:stop]) # MODIFICATION
        thisStringIO.write('\n') # MODIFICATION
    return thisStringIO # MODIFICATION
```

Figure 5.30: The modifications that were made on the **kmerize** function the script in order to accommodate the application of Pseudo File.

```
def kmerizeAlignments(alignmentsIterator, k):
    kmerizedList = StringIO() # MODIFICATION
    for alignment in alignmentsIterator:
        kmerData = kmerize(alignment.query_sequence,k)
        kmerizedList.write(kmerData.getvalue()) # MODIFICATION
        kmerizedList.write('\n') # MODIFICATION
    return kmerizedList
```

Figure 5.31: The modifications that were made on the **kmerizeAlignments** function of the script in order to accommodate the application of Pseudo File.

```
alignmentsIterator = thisBam.fetch(bedRecord['chr'], thisStart, thisEnd
 ↪ + 1)
trainingData[len(trainingData):] =
 ↪ kmerizeAlignments(alignmentsIterator, kmerSize).getvalue().split()
 ↪ # MODIFICATION
target_labels.append(classesList.index(thisClass))
```

Figure 5.32: The modifications that were made on other sections of the script in order to accommodate the application of Pseudo File.

The accuracy of this method was also proved since it returned the same output from the **kmerization** process.

The Deterministic Profiling of the new version of the script returned the following results:

Run	Execution Time	Run	Execution Time
1	16.832	**11**	17.001
2	16.638	**12**	16.527
3	16.914	**13**	16.691
4	16.754	**14**	16.500
5	16.706	**15**	16.441
6	16.572	**16**	16.446
7	16.776	**17**	16.832
8	16.773	**18**	16.773
9	16.894	**19**	17.056
10	16.529	**20**	16.677

Figure 5.33: The execution times (in seconds) for the 20 consecutive times for which the Deterministic Profiler was run on the script that adopted the Pseudo File optimisation.

	Baseline System	Pseudo File System
Mean Value	21.7759	16.7166
Standard Deviation	0.1320	0.1779
Most representative performance	21.7830	16.706
Run with the most representative performance	11	5

Figure 5.34: The aggregated results from the Deterministic Profiling of the script that adopted the Pseudo File optimisation.

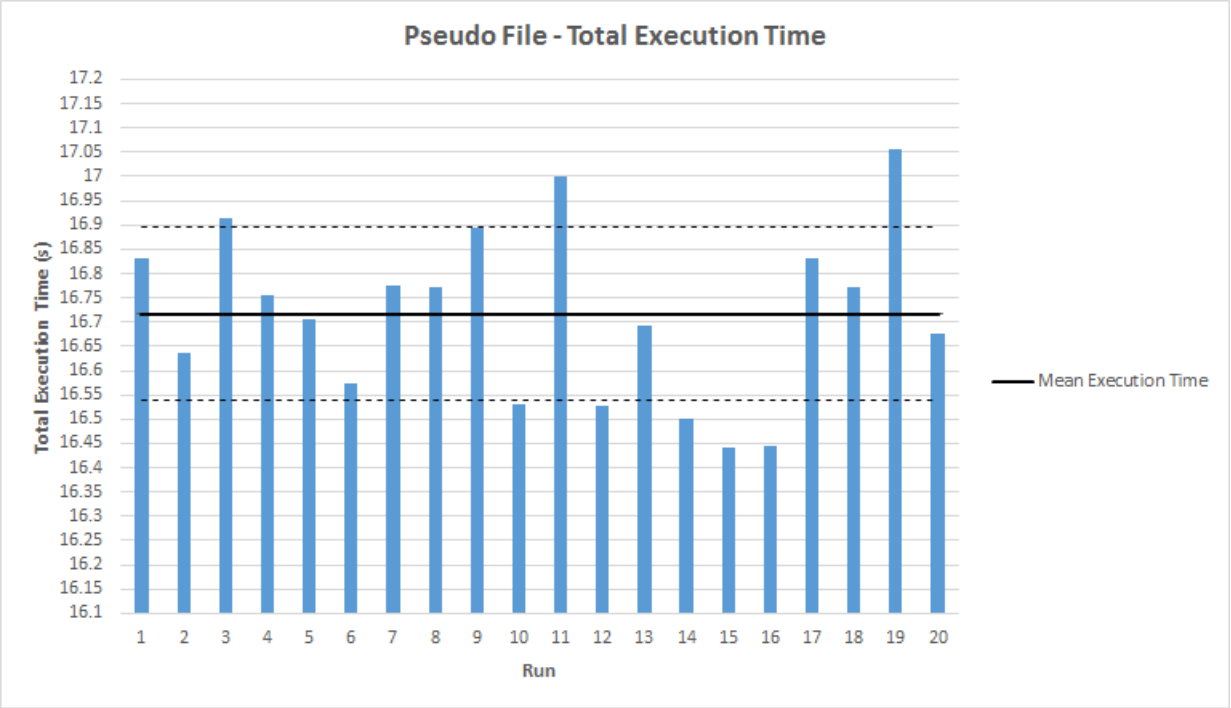

Figure 5.35: A Graph that shows the Total Execution Time (in seconds) of the System after the application of Pseudo File optimisation for 20 Consecutive Runs.

As it can be noticed by the graph, the Pseudo File technique resulted to only a single outlier that has a large absolute difference from the **Mean Value**. This may have led to the largest value of **Standard Deviation** so far and subsequently the greater area that was bounded within its upper and lower limits.

The use of this pseudo file approach was proved to be efficient since it improved the execution time of the script by reducing it to the **77%** of the corresponding value of the original system. It was an acceptable improvement but again it was not as large as the ones noticed before. This was due to the fact that **cStringIO** was not built for providing better performance over another operation; it was simply created to support the cases when an **API** requires files as input whilst a developer is intended to pass strings.

In order to fully accept these results, the usual statistical analysis process should have been conducted to prove their significance. The **null hypothesis** there was that the replacement of the plain String Concatenation with the Pseudo File approach did not improve the performance of the system and the **alternative hypothesis** was that it did. The **p-value** in this case was equal to $\mathbf{8.69x10^{-28}}$ which is smaller than the value of the constant **a**, therefore the null hypothesis was **rejected** and the alternative one was proved to be correct. Thus, the solution that provides the use of Pseudo File is acceptable as an optimisation over the original script.

```
Filename: bedNbamClassifier.py

Line # Mem usage Increment Line Contents
================================================
  114 347.062 MiB 0.000 MiB @profile
  115 def kmerize(dna, k):
  116 347.062 MiB 0.000 MiB thisStringIO = StringIO()
  117 347.062 MiB 0.000 MiB for i in range(0, len(dna)+1-k, k):
  118 347.062 MiB 0.000 MiB start = max(0, i)
  119 347.062 MiB 0.000 MiB stop = min(len(dna), i + k) + 1
  120 347.062 MiB 0.000 MiB thisStringIO.write(dna[start:stop])
  121 347.062 MiB 0.000 MiB thisStringIO.write('\n')
  122 347.062 MiB 0.000 MiB return thisStringIO
```

Figure 5.36: The section of the report generated by the **memory_profiler** for the **kmerize()** function of the System after the application of the Pseudo File optimisation.

```
Filename: bedNbamClassifier.py

Line # Mem usage Increment Line Contents
================================================
  129 345.609 MiB 0.000 MiB @profile
  130 def kmerizeAlignments(alignmentsIterator, k):
  131 345.609 MiB 0.000 MiB kmerizedList = StringIO()
  133 345.609 MiB 0.000 MiB for alignment in alignmentsIterator:
  134 345.609 MiB 0.000 MiB kmerData = kmerize(alignment.query_sequence,k)
  135 345.609 MiB 0.000 MiB kmerizedList.write(kmerData.getvalue())
  136 345.609 MiB 0.000 MiB kmerizedList.write('\n')
  137 345.609 MiB 0.000 MiB return kmerizedList
```

Figure 5.37: The section of the report generated by the **memory_profiler** for the **kmerizeAlignments()** function of the System after the application of the Pseudo File optimisation.

The Memory Profiling suggests that the Pseudo File approach had a memory performance that was close to the Average of the aforementioned methods. Unfortunately, it was still not as optimal as the memory usage of the Original script.

5.1.5 List Comprehensions

List Comprehensions provide a concise alternative to the Python List construction. In a nutshell, List Comprehensions is essentially a tool for transforming each member of a sequence or an iterable into another list by applying some specific operations on them.

Some examples can be seen in the figure below:

```
l1 = [x**2 for x in range(10)]
l2 = [x for x in range(1, 10)]


print(l1)
>> [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

print(l2)
>> [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Figure 5.38: Some examples of List Comprehensions that illustrate their functionality

In these examples, List Comprehensions apply a specific operation on each member of the range of numbers they are given in the **for** loop and return a list with the resulted values.

The List Comprehension technique was applied to the script and the modifications that have been made in order to accommodate it are shown below:

```
def kmerize(dna, k):
    return [dna[max(0, i):min(len(dna), i + k) + 1] for i in range(0,
     ↪ len(dna)+1-k, k)] # MODIFICATION
```

Figure 5.39: The modifications that were made on the **kmerize** function of the script in order to accommodate the application of List Comprehensions.

```
def kmerizeAlignments(alignmentsIterator, k):
        kmerizedList = [] # MODIFICATION
    return [''.join(kmerize(alignment.query_sequence,k)) for alignment
     ↪ in alignmentsIterator] # MODIFICATION
```

Figure 5.40: The modifications that were made on the **kmerizeAlignments** function of the script in order to accommodate the application of List Comprehensions.

```
alignmentsIterator = thisBam.fetch(bedRecord['chr'], thisStart, thisEnd
 ↪ + 1)
trainingData[len(trainingData):] =
 ↪ kmerizeAlignments(alignmentsIterator, kmerSize) # MODIFICATION
target_labels.append(classesList.index(thisClass))
```

Figure 5.41: The modifications that were made on other sections of the script in order to accommodate the application of List Comprehensions.

The correctness of this method was proved by applying the usual accuracy assessment. The output returned was identical to the one of the original system.

Run	Execution Time	Run	Execution Time
1	9.974	**11**	9.849
2	9.829	**12**	9.867
3	9.798	**13**	9.906
4	10.363	**14**	9.867
5	9.972	**15**	9.902
6	9.926	**16**	9.911
7	9.932	**17**	9.853
8	9.869	**18**	9.982
9	9.831	**19**	9.875
10	10.02	**20**	9.856

Figure 5.42: The execution times (in seconds) for the 20 consecutive times for which the Deterministic Profiler was run on the script that adopted the List Comprehensions optimisation.

	Baseline System	List Comprehensions System
Mean Value	21.7759	9.9191
Standard Deviation	0.1320	0.1195
Most representative performance	21.7830	9.926
Run with the most representative performance	11	6

Figure 5.43: The aggregated results from the Deterministic Profiling of the script that adopted the second approach to the application of List Comprehensions optimisation.

Figure 5.44: A Graph that shows the Total Execution Time (in seconds) of the System after the application of List Comprehensions optimisation for 20 Consecutive Runs.

The graph shows a pretty normal distribution of values with only a single outlier that was located outside the **Standard Deviation** boundaries. It can be assumed that in case this outlier did not existed or its variance from the rest of the values was smaller then probably the **Mean Value** would be even smaller.

As it can be seen by the above plot and results, the use of List Comprehensions brought some significant improvements on the performance of the script. The **Mean Value** of the Execution time was reduced to the **47%**

percent of the Mean Value of the baseline system and thus it can be deduced that the execution speed was more than the double of its original value.

In order to explain the above observation, an online research for relevant literature was conducted. One source suggested that the optimality that lies behind the use of List Comprehensions over the simple list creation through a for loop is due to the operations that happen in the lower, bytecode level. According to an experiment conducted by Leary, List Comprehensions perform better because they are not required to load the append attribute off of the list (**LOAD_ATTR** bytecode) and call it later as a function (**CALL_FUNCTION** bytecode) as it is happening with the simple method but it actually generates a special **LIST_APPEND** bytecode that enables the faster appending of the new elements onto the resulted list [22].

An final step before accepting the obvious superiority of the List Comprehensions was the repetition of the usual statistical analysis process for proving the significance of its results. The **null hypothesis** there was that the replacement of the plain String Concatenation with the List Comprehensions method did not improve the performance of the system and the **alternative hypothesis** was that it did. The **p-value** in this case was equal to **8.21×10^{-37}** which is smaller than the value of the constant **a**, therefore the null hypothesis was **rejected** and the alternative one was proved to be correct. Thus, the solution that provides the use of List Comprehensions is also statistically approved as an optimisation over the original script.

```
Filename: bedNbamClassifier.py

Line # Mem usage Increment Line Contents
================================================
  113 109.344 MiB 0.000 MiB @profile
  114 def kmerize(dna, k):
  115 109.344 MiB 0.000 MiB return [dna[max(0, i):min(len(dna), i + k) + 1] for i in range
      (0, len(dna)+1-k, k)]
```

Figure 5.45: The section of the report generated by the **memory_profiler** for the **kmerize()** function of the System after the application of the List Comprehensions optimisation.

```
Filename: bedNbamClassifier.py

Line # Mem usage Increment Line Contents
================================================
  125 109.191 MiB 0.000 MiB @profile
  126 def kmerizeAlignments(alignmentsIterator, k):
  127 109.191 MiB 0.000 MiB kmerizedList = []
  128 109.316 MiB 0.125 MiB return [''.join(kmerize(alignment.query_sequence,k)) for
      alignment in alignmentsIterator]
```

Figure 5.46: The section of the report generated by the **memory_profiler** for the **kmerizeAlignments()** function of the System after the application of the List Comprehensions optimisation.

The memory performance of this technique was also the most optimal amongst the different optimisations since it managed to use only around **4MiB** of more memory than the original system in contrast with the rest of the methods that allocated 3 times more memory than the baseline system.

5.1.6 Linked List

But apart from the alternatives to the String Concatenation that Python itself can provide to the developer, it is also possible to make use of user-defined data structures. The structure that will be discussed in this section is the Linked List. A Linked List is a dynamic data structure that can store elements in a linear way. Its elements are called nodes and they are generic containers of data. Each of these nodes is comprised of the value it stores and (at least) a reference to the next node. A Linked List can be either Singly or Doubly linked. The former variation provides only a reference to the next node whilst the latter provide an extra reference to the previous node that makes it connected in both directions.

One significant benefit from the use of a Linked List data structure is the fact that the capacity is completely dynamically allocated in the sense that it is not assigned an initial capacity but, it grows or shrink with respect to the nodes that are inserted to or removed from it, instead [1].

A significant drawback of the Linked List structure is the fact that the elements that are stored within this list cannot be accessed individually. Instead, it uses a sequential access model where when one needs to access a particular element within the list, he/she visit all the references, one by one, in order to reach the node of interest [1]. This is not a problem in the specific case since the system always iterates over all the elements of the structure without any need to access a single item individually.

The modifications made on the script can be shown below:

```
class Node:

    def __init__(self, data=None, next_node=None):
        self.data = data
        self.next_node = next_node

    def get_data(self):
        return self.data

    def get_next(self):
        return self.next_node

    def set_next(self, new_next):
        self.next_node = new_next
```

Figure 5.47: A Python Implementation of the **Node** class

```
class LinkedList:

    head = None
    size = 0

    def __init__(self, head=None, size=0):
        head = None
        size = 0

    def isEmpty(self):
        return self.size == 0

    def get_size(self):
        return self.size

    def insert(self, data):

        node = Node(data, None)

        if self.isEmpty():
            self.head = node
        else:
            current = self.head

            while current.get_next() is not None:
                current = current.get_next()

            current.set_next(node)

        self.size = self.size + 1

    def append(self, other):

        if self.isEmpty() and other.isEmpty():
            return
        elif self.isEmpty():
            self.head = other.head
            return self
        elif other.isEmpty():
            return self
        else:
            current = self.head

            while current.get_next() is not None:
                current = current.get_next()

            current.set_next(other.head)
```

Figure 5.48: A Python Implementation of the **LinkedList** class

```
class LinkedList:
    def tolist(self):
        l = []
        current = self.head

        while current is not None:
            l.append(current.get_data())
            current = current.get_next()

        return l
```

Figure 5.49: A Python Implementation of the **LinkedList** class (continued)

The modifications on the script that have been made in order to accommodate the Linked List data structure are shown below:

```
def kmerize(dna, k):
    thisLinkedList = LinkedList(None) # MODIFICATION
    for i in range(0, len(dna)+1-k, k):
        start = max(0, i)
        stop = min(len(dna), i + k) + 1
        thisLinkedList.insert(Node(dna[start:stop])) # MODIFICATION
    return thisLinkedList # MODIFICATION
```

Figure 5.50: The modifications that were made on the **kmerize** function of the script in order to accommodate the implementation of the Linked List.

```
def kmerizeAlignments(alignmentsIterator, k):
    kmerizedList = [] # MODIFICATION
    for alignment in alignmentsIterator:
        kmerData = kmerize(alignment.query_sequence,k)
        kmerizedList[len(kmerizedList):] = kmerData.tolist() #
         ↪ MODIFICATION
    return kmerizedList
```

Figure 5.51: The modifications that were made on the **kmerizeAlignments** function of the script in order to accommodate the implementation of the Linked List.

```
alignmentsIterator = thisBam.fetch(bedRecord['chr'], thisStart, thisEnd
 ↪ + 1)
trainingData[len(trainingData):] =
 ↪ kmerizeAlignments(alignmentsIterator, kmerSize) # MODIFICATION
target_labels.append(classesList.index(thisClass))
```

Figure 5.52: The modifications that were made on other sections of the script in order to accommodate the implementation of the Linked List.

This time, apart from modifying specific portions of the existing code, it was necessary to create two additional classes to serve the implementation of the Linked List data structure. These were the **Node** and the **Linked List** classes. The former constructs a **Node** instance by assigning the **data** and the **next_node** attributes to it and complementing it with **Accessor** and **Mutator** methods. The latter constructs the **Linked List** structure by assigning the **head** attribute to it and an initial **size** of zero and supplementing it with three methods **insert()**, **append()** and **tolist()** that support the operations of inserting an element to the Linked List, appending two Linked Lists and converting a Linked List into to a plain Python list. It is important to state that in order to make sure than the script will recognise the existence of the two new classes, it was required to append the code that defines them before the section of the code that creates instances of them [41].

The actual application of the new structure on the script can be seen on the modifications made on the existing code. In the **kmerize()** function, a new Linked List was created before the **for-each** loop. Each **kmer** that had been constructed within the loop was inserted to it using the **insert()** method defined for this class. Each Linked List instance created for a particular alignment iterator was appended to a larger Linked List called **kmerizedList** using the **append()** method. Later, this list was converted into a plain Python List using the **tolist()** method and appended as a list slice into the universal list of **kmers** for all the **BAM** files called **trainingData**.

The optimisation was assessed using the usual process:

Run	Execution Time	Run	Execution Time
1	97.303	**11**	97.958
2	98.207	**12**	97.775
3	97.579	**13**	98.906
4	97.844	**14**	97.891
5	97.620	**15**	97.393
6	97.367	**16**	99.071
7	97.757	**17**	97.232
8	97.621	**18**	97.237
9	98.389	**19**	97.748
10	97.436	**20**	96.967

Figure 5.53: The execution times (in seconds) for the 20 consecutive times for which the Deterministic Profiler was run on the script that adopted the Linked List optimisation.

	Baseline System	Linked List System
Mean Value	21.7759	97.7651
Standard Deviation	0.1320	0.5403
Most representative performance	21.7830	97.7570
Run with the most representative performance	11	7

Figure 5.54: The aggregated results from the Deterministic Profiling of the script that adopted the Linked List optimisation.

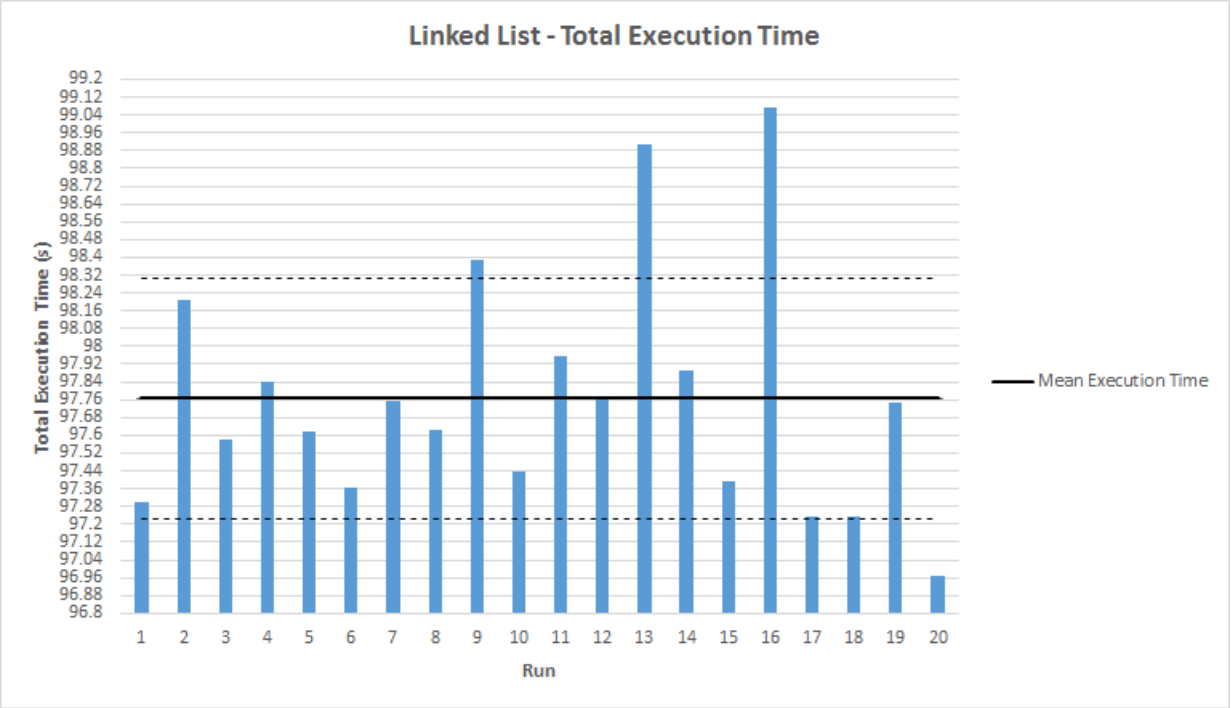

Figure 5.55: A Graph that shows the Total Execution Time (in seconds) of the System after the application of Linked List optimisation for 20 Consecutive Runs.

As the above results suggest, the implementation of a python version of a Linked List data structure did not improved the execution of the script. On the contrary, it can be noticed that it actually added some overhead over the execution of the script.

These results disproved the initial assumption that a Linked List, as a dynamic Data Structure, would benefit the performance of the tool. This behaviour can be explained by the fact that, even though it was possible to use this data structure in different sections of the script without any problems, it was necessary to convert it into an iterable object type before passing it as an argument to the **fit_transform()** method, according to the **sci-kit** learn Documentation [50]. Therefore, by defining an appropriate **tolist()** method within the **LinkedList** class, the Linked List can be converted into a Python List before being passed as an argument in the aforementioned method. Unfortunately, this process itself added a significant overhead on the execution of the script and this could have caused the extra delay. An additional reason why this implementation did not improved the performance of the script at all is the fact the the results from the accuracy assessment of this technique were not 100% positive and this may have also led to a degradation of the efficiency of the script.

```
Filename: bedNbamClassifier.py

Line # Mem usage Increment Line Contents
================================================
   188 63.473 MiB 0.000 MiB @profile
   189 def kmerize(dna, k):
   190 63.473 MiB 0.000 MiB thisLinkedList = LinkedList()
   191 63.473 MiB 0.000 MiB for i in range(0, len(dna)+1-k, k):
   192 63.473 MiB 0.000 MiB start = max(0, i)
   193 63.473 MiB 0.000 MiB stop = min(len(dna), i + k) + 1
   194 63.473 MiB 0.000 MiB thisLinkedList.insert(dna[start:stop])
   195 63.473 MiB 0.000 MiB return thisLinkedList
```

Figure 5.56: The section of the report generated by the **memory_profiler** for the **kmerize()** function of the System after the application of the Linked List optimisation.

```
Filename: bedNbamClassifier.py

Line # Mem usage Increment Line Contents
================================================
   202 63.508 MiB 0.000 MiB @profile
   203 def kmerizeAlignments(alignmentsIterator, k):
   204 63.508 MiB 0.000 MiB kmerizedList = LinkedList()
   205 63.508 MiB 0.000 MiB for alignment in alignmentsIterator:
   206 63.508 MiB 0.000 MiB kmerData = kmerize(alignment.query_sequence,k)
   207 63.508 MiB 0.000 MiB kmerizedList.append(kmerData)
   208 63.508 MiB 0.000 MiB return kmerizedList
```

Figure 5.57: The section of the report generated by the **memory_profiler** for the **kmerizeAlignments()** function of the System after the application of the Linked List optimisation.

It is worth mentioning the remarkable improvement that the application of the Linked List data structure has provided to the memory usage of the script. It managed to reduce the use of memory to the half of the original one. This can be easily explained by taking the nature of the data structure into consideration. Up until the point where elements are actually added to the Linked List, no memory is allocated to it. This means that this data structure only takes up the amount of memory it necessarily needs to store its elements; no more, no less. This can be certainly considered to be the most optimal memory allocation strategy since it does not waste any space, no matter the present or the absence of any elements to be stored in the Linked List.

Chapter 6

Conclusions

The project itself as well as the experimental procedure followed throughout its duration was source for a large number of conclusions for the behaviour of the script itself as well as the process of optimisation and the scientific method in general. In the following sections, these conclusions and the personal reflection of the author upon the undertaken project will be provided and discussed.

6.1 Conclusions

Firstly, as far as the specific script is concerned, the most optimal solution to String Immutability was proved to be the use of **List Comprehensions**. This optimisation provided a significant improvement over the performance of the original script. To be more specific, it managed to double the execution speed which is remarkable for a single optimisation. In addition, some runner-up optimisations were identified that increased the efficiency of the script but not in this extend. These were the **second approach to the application of Python String List**, the use of **Character Array** and the application of **Pseudo File** method.

On the contrary, some techniques such as the **first approach to the application of Python String List** and the implementation of **Linked List** not only did they not improve the performance of the script but also added a significant delay to its execution and increased the memory consumption. This was surprising due to the fact that they were, initially, considered to be more optimal than the original system.

Furthermore, another concept that was stressed from the results was the computational importance of the study of String Concatenation. It is extremely costly as an operation and especially when it is used to solve problems in the larger scale it can be destructive for the efficiency of a software component. A concrete proof of this claim is the significant overhead that it added to the Origin Classification tool. Since it was the dominant operation of the script, the fact that it used an inefficient way to be performed caused enormous delays in the execution of the whole script. The observed improvement on its performance after the application of the most optimal alternative was amazing. Therefore, this led to the general conclusion which stated that pure String Concatenation should be avoided to the greatest extend possible since it degrades the performance of the script without providing any significant benefits over the other alternatives that produce the same result.

6.2 Reflection

The whole experimental process followed for the purposes of this project as well as the project itself has led to some serious consideration of some aspects of the software optimisation and the experimental approach of a scientific problem.

The process of Optimisation is by definition a complex problem. There are no general guidelines or guaranteed solutions that always work. Everything depends on the nature and the context of the problem to which the optimisation will be applied to. Different solutions can be proved optimal for different applications of the same problem. Therefore, one should always experiment with various approaches in order to determine the most appropriate solution.

Additionally, the Optimisation procedure is possible to provide some surprising results that are contradictory to what it was expected. For instance, a modification that according to the programmer's assumption would certainly improve the performance of a script can be proved to actually downgrade it or the opposite. However, this could provide food for thought since it requires a better insight on the lower level of execution that would help in gaining useful knowledge regarding the smaller picture of how a certain operation works.

From a Scientific perspective, this project cultivated the idea that new knowledge not only does come from proving the correctness of a hypothesis but also by disproving it. Forming a clear and correct idea on how certain things work is an iterative process; it starts from some initial assumptions and results in the integration of the whole picture. The intermediate steps of this procedure bring small snippets of knowledge that are used in the following steps to solve the puzzle of the initial question that was the seed for the plantation of this process.

Finally, in terms of personal development, this project offered the chance to work on a real-life application of Computing Science in Biology and Medicine by working on a product that was built to support the Scientific Research. More specifically, this project enabled the author to deal with Genetic Sequencing, an emerging and interesting topic in the aforementioned fields, and get a first-hand experience from its practical applications by attending meetings of the Wellcome Trust Centre of Molecular Parasitology Bioinformatics team with the purpose of discussing the operation of the script with its members. This was a pretty interesting experience since it provided the opportunity for constructive interaction and cooperation between individuals from different academic backgrounds (Computing Science and Biology) for considering different approaches that could be taken for the solving the same problem.

Overall, the project was very challenging and interesting and it provided knowledge from different aspects and academic disciplines.

Chapter 7

Future Work

As this iteration of the project came to an end, a number of suggestions will be provided as seeds for study for the people that may undertake the project in the future. The suggestions will have two scopes: short-term proposals and long-term proposals. The difference between them is that the former can be applied on the tool immediately whilst the latter require reformation of the code.

Firstly, it would be recommended to take up some investigations that originated from results from this iteration of the project. For instance, it would be extremely useful to examine whether an alternative implementation of the Linked List would be optimal for the performance of the script as it was proved to be in terms of memory usage. Additionally, an additional optimisation iteration focused only on the improvement of the memory usage of the script would certainly lead to some useful alterations.

As a supplement to the above recommendations, an investigation of how an appropriate implementation of a Trie Data Structure could be possibly used to ease the process of storing **kmers** and how it will cooperate with the **CountVectorizer** object from the **sciki-learn** library in order to perform the counting of the occurrences of each **kmer** in the input file(s). The Trie is capable of storing each **kmer** efficiently in a tree-like manner so that the **kmers** that appear more frequently in the input would be stored closer to the root node of this tree in order to be traversed more quickly.

The first suggestion for new work provides the application of an object-oriented programmatistic style on the script. That would involve the change of the script structure so that it will become more modular. This means that the functionality of the script would be split into classes and functions. The aforementioned change would help any future profilings of the script since some profilers such as the **memory_profiler** can only assess individual methods or functions rather than the whole script. That would also make the debugging easier since the errors are more easily observable in smaller blocks of code and it would improve the the portability of the tool in general.

Another suggestion is to try the application of multithreading on the script. Python provides a collection of modules to support threading in its scripts. The **threading** module that is based on the lower-level **thread** module can be used for the manipulation of threads within the script [38]. The application of threads would enforce the introduction of concurrency within the Classification tool by taking advantage of the different processing cores of the machine that it runs on. For example, a **Master** thread could be assigned the delegation of each **BAM** file to a different **Worker** thread to enable their processing in parallel. It should be kept in mind though that all the data structures used within the script should be thread safe to avoid any undesirable results caused by the concurrent access of threads on the same data at the same time and therefore, a careful research should be conducted before in order to perform all the appropriate actions needed to ensure it. For example, as far as the pandas data structures (**Series** and **Dataframe**) are concerned, their documentation suggests that "*As of pandas 0.11, pandas is not 100% thread safe*" by definition and it recommends the use of **locks** to avoid any threading issues [31].

Moreover, since the project deals with the processing of significantly large amounts of data, a cluster computing programmatic paradigm could be used to support the tool. The **Apache Hadoop** framework or the **Apache Spark** could be introduced so that they would provide a Big Data infrastructure to the system. The latter one is more preferred since it is more prominent in the Big Data area at the moment whilst the former is considered to be deprecated.

Amongst the longer term suggestions is the migration to another programming language to exploit the different advantages that they provide. For example, if the use of **Apache Hadoop** is seriously considered, it is worth converting the code from **Python** to **Java** since the latter is the native language of the framework and it would possibly make its use easier. The same holds for **Apache Spark** and **Scala**. Furthermore, one might consider migrating to a lower-level language such as C to be benefited from the, by definition, faster execution of the script.

The conversion of the script to a different language is difficult though. Apart from the effort of actually translating the script from one language to another which can be considered as trivial, there is also the significant overhead for figuring out how to preserve the functionality provided by the various external libraries such as the **sci-kit learn**. Some of them are either not easily accessible from other languages or not available at all and that would require some extra research for identifying any possible alternatives which can take a significant amount of time that may not worth the trade-off.

Bibliography

[1] V.S. Adamchik. Carnegie Mellon University - Computer Science - 121 - Linked Lists. https://www.cs.cmu.edu/~adamchik/15-121/lectures/Linked%20Lists/linked%20lists.html. Accessed: 2016.

[2] AllSeq. Ion Torrent. http://allseq.com/knowledge-bank/ion-torrent/. Accessed: 2016.

[3] Amazon Web Services. Amazon DynamoDB. https://aws.amazon.com/dynamodb/. Accessed: 2016.

[4] Amazon Web Services. Amazon EC2 - Virtual Server Hosting. https://aws.amazon.com/ec2/. Accessed: 2016.

[5] Amazon Web Services. Amazon EMR. https://aws.amazon.com/elasticmapreduce/. Accessed: 2016.

[6] Amazon Web Services. Amazon S3. https://aws.amazon.com/s3/. Accessed: 2016.

[7] Amazon Web Services. Amazon Web Services. https://aws.amazon.com/. Accessed: 2016.

[8] D.M. Beazley. *Python Essential Reference*. Sams Publishing, 2006.

[9] S.M. Brown. Sequencing-by-Synthesis: Explaining the Illumina Sequencing Technology. http://bitesizebio.com/13546/sequencing-by-synthesis-explaining-the-illumina-sequencing-technology/. Accessed: 2016.

[10] BusinessDictionary.com. What is stochastic? definition and meaning. http://www.businessdictionary.com/definition/stochastic.html. Accessed: 2016.

[11] E. Canfield. Sanger Method for DNA Sequencing. http://www.bio.davidson.edu/Bio111/seq.html. Accessed: 2016.

[12] GENOME Unlocking Life's Code. Timeline of the human genome. https://unlockinglifescode.org/timeline. Accessed: 2016.

[13] Cython Developers. Cython - C-Extensions for python. http://cython.org/. Accessed: 2016.

[14] EMBL-EBI. Ion Torrent: Proton / PGM sequencing. https://www.ebi.ac.uk/training/online/course/ebi-next-generation-sequencing-practical-course/what-next-generation-dna-sequencing/ion-torre. Accessed: 2016.

[15] EMBL-EBI. What is Next-Generation DNA Sequencing. https://www.ebi.ac.uk/training/online/course/ebi-next-generation-sequencing-practical-course/what-you-will-learn/what-next-generation-dna-. Accessed: 2016.

[16] Encyclopaedia Britannica. Denaturation. http://www.britannica.com/science/denaturation. Accessed: 2016.

[17] Encyclopaedia Britannica. DNA sequencing - First-generation sequencing technology. http://www.britannica.com/science/DNA-sequencing. Accessed: 2016.

[18] S.M. Huse, J.A. Huber, H.G. Morrison, M.L. Sogin, and D.M. Welch. Accuracy and quality of massively parallel DNA pyrosequencing. *Genome Biology*, 8(7), 2007.

[19] illumina. Sequencing by Synthesis (SBS) Technology. http://www.illumina.com/technology/next-generation-sequencing/sequencing-technology.html. Accessed: 2016.

[20] D. Jurafsky and J.H. Martin. *Speech and Language Processing*. 2014.

[21] Lab CE. Acceptable Standard Deviation (SD). https://www.labce.com/spg49741_acceptable_standard_deviation_sd.aspx. Accessed: 2016.

[22] C. Leary. Efficiency of list comprehensions. http://blog.cdleary.com/2010/04/efficiency-of-list-comprehensions/. Accessed: 2016.

[23] matplotlib Development Team. matplotlib: python plotting. http://matplotlib.org/. Accessed: 2016.

[24] J.D. Meier, C. Farre, P. Bansode, and S. Barber. Explained: Types of Performance Testing. http://perftesting.codeplex.com/wikipage?title=Explained%3a%20Types%20of%20Performance%20Testing&referringTitle=Performance%20Testing. Accessed: 2016.

[25] National Human Genome Research Institute. The Human Genome Project Completion: Frequently Asked Questions. http://www.genome.gov/11006943. Accessed: 2016.

[26] National Institute of Health Research Portfolio Online Reporting Tools (RePORT). Human Genome Project. https://report.nih.gov/nihfactsheets/ViewFactSheet.aspx?csid=45. Accessed: 2016.

[27] NumPy Developers. NumPy. http://www.numpy.org/. Accessed: 2016.

[28] Oracle. Java Documentation - The Java Tutorials - Immutable Objects. https://docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html. Accessed: 2016.

[29] Oracle. Java Platform, Standard Edition 7 - API Specification - Class StringBuffer. https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuffer.html. Accessed: 2016.

[30] Oracle. Java Platform, Standard Edition 7 - API Specification - Class StringBuilder. https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html. Accessed: 2016.

[31] Pandas Development Team. Caveats and Gotchas - Thread Safety. http://pandas.pydata.org/pandas-docs/stable/gotchas.html#thread-safety. Accessed: 2016.

[32] B.Y. Park, B.C. Lee, K.H. Jung, M.H. Jung, B.L. Park, Y.G. Chai, and I.G. Choi. Epigenetic changes of serotonin transporter in the patients with alcohol dependence: methylation of an serotonin transporter promoter CpG island. *Psychiatry Investigation*, 8:130–133, 2011.

[33] PyData. pandas. http://pandas.pydata.org/. Accessed: 2016.

[34] Pysam Developers. Pysam. https://code.google.com/archive/p/pysam/. Accessed: 2016.

[35] Pysam Developers. pysam - an interface for reading and writing SAM files. http://pysam.readthedocs.org/en/latest/api.html. Accessed: 2016.

[36] Python for Beginners. String Concatenation and Formatting. http://www.pythonforbeginners.com/concatenation/string-concatenation-and-formatting-in-python. Accessed: 2016.

[37] Python Software Foundation. 10.4 what is deterministic profiling? https://docs.python.org/2.4/lib/node453.html. Accessed: 2016.

[38] Python Software Foundation. 16.2. threading — Higher-level threading interface. https://docs.python.org/2/library/threading.html. Accessed: 2016.

[39] Python Software Foundation. 5.6. Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange. https://docs.python.org/2/library/stdtypes.html#typesseq. Accessed: 2016.

[40] Python Software Foundation. 8.14. UserString — Class wrapper for string objects. http://www.tutorialspoint.com/python/python_lists.htm. Accessed: 2016.

[41] Python Software Foundation. 9.3.1. Class Definition Syntax. https://docs.python.org/2/tutorial/classes.html#class-definition-syntax. Accessed: 2016.

[42] Python Software Foundation. memory_profiler 0.41. https://pypi.python.org/pypi/memory_profiler. Accessed: 2016.

[43] Python Software Foundation. Python 2.7.11 Documentation 26.4. The Python Profilers. https://docs.python.org/2/library/profile.html. Accessed: 2016.

[44] QIAGEN. Pyrosequencing Technology and Platform Overview - Principle of Pyrosequencing. https://www.qiagen.com/gb/resources/technologies/pyrosequencing-resource-center/technology-overview/#principle. Accessed: 2016.

[45] G. Reda. Intro to pandas data structures. http://www.gregreda.com/2013/10/26/intro-to-pandas-data-structures/. Accessed: 2016.

[46] J.B. Reece, L.A. Urry, M.L. Cain, S.A. Wasserman, P.V. Minorsky, and R.B. Jackson. *Campbell Biology (9th Edition)*. Benjamin Cummings, 9 edition, 2009.

[47] M. Ronaghi. Pyrosequencing Sheds Light on DNA Sequencing. *Genome Biology*, 11:3–11, 2011.

[48] Samtools Developers. Cram format specification (version 3.0). http://samtools.github.io/hts-specs/CRAMv3.pdf. Accessed: 2016.

[49] scikit-learn Developers. scikit-learn - Machine Learning in Python. http://scikit-learn.org/. Accessed: 2016.

[50] scikit-learn developers. sklearn.feature_extraction.text.countvectorizer. http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html#sklearn.feature_extraction.text.CountVectorizer.fit_transform. Accessed: 2016.

[51] SciPy.org. SciPy. http://www.scipy.org/. Accessed: 2016.

[52] Scitable by nature Education. primer - Definition. http://www.nature.com/scitable/definition/primer-305. Accessed: 2016.

[53] M.S. Shell. *An introduction to Numpy and Scipy*. University of Santa Barbara, 2014.

[54] J.F. Siqueira, A.F. Fouad, and I.N. Rôças. Pyrosequencing as a tool for better understanding of human microbiomes. *Journal of Oral Microbiology*, 4, 2012.

[55] SmartBear. What is Load Testing? http://smartbear.com/learn/performance-testing/what-is-load-testing/. Accessed: 2016.

[56] StackExchange Biology. What does 5' and 3' mean in dna and rna strands? http://biology.stackexchange.com/questions/15082/what-does-5-and-3-mean-in-dna-and-rna-strands. Accessed: 2016.

[57] stackoverflow. Python List vs. Array - when to use? http://stackoverflow.com/questions/176011/python-list-vs-array-when-to-use. Accessed: 2016.

[58] SymPy Development Team. SymPy. http://www.sympy.org/en/index.html. Accessed: 2016.

[59] IPython Development Team. Jupyter and the future of IPython - IPython. http://ipython.org/. Accessed: 2016.

[60] TechTarget WhatIs.com. Semiconductor - Definition. http://whatis.techtarget.com/definition/semiconductor. Accessed: 2016.

[61] The Free Dictionary by Farlex. Medical dictionary - Biotinylation. http://medical-dictionary.thefreedictionary.com/biotin. Accessed: 2016.

[62] The SAM/BAM Format Specification Working Group. Sequence Alignment/Map Format Specification. https://samtools.github.io/hts-specs/SAMv1.pdf. Accessed: 2016.

[63] Thermo Scientific™ Pierce™ Protein Methods Library. Biotinylation. https://www.thermofisher.com/uk/en/home/life-science/protein-biology/protein-biology-learning-center/protein-biology-resource-library/pierce-protein-methods/biotinylation.html. Accessed: 2016.

[64] tutorialspoint. Python Lists. http://www.tutorialspoint.com/python/python_lists.htm. Accessed: 2016.

[65] University of California Open Computer Facility. DNA Sequencing. https://www.ocf.berkeley.edu/~edy/genome/sequencing.html. Accessed: 2016.

[66] University of Utah Health Sciences. PCR howpublished = http://learn.genetics.utah.edu/content/labs/pcr/, note = Accessed: 2016,.

[67] Virtual Amrita Laboratories Universalizing Education. Polyacrylamide Gel Electrophoresis. http://vlab.amrita.edu/?sub=3&brch=186&sim=319&cnt=1. Accessed: 2016.

[68] Wellcome Trust. DNA sequencing - the Sanger method. http://www.wellcome.ac.uk/Education-resources/Education-and-learning/Resources/Animation/WTDV026689.htm. Accessed: 2016.

[69] Yale University Molecular Biophysics & Biochemistry. A History of Genome Sequencing. http://bioinfo.mbb.yale.edu/course/projects/final-4/. Accessed: 2016.

[70] YourGenome. What is the Illumina method of DNA sequencing? http://www.yourgenome.org/facts/what-is-the-illumina-method-of-dna-sequencing. Accessed: 2016.

Chapter 8

Appendix

Deterministic Profiling - Cumulative Time - String Appending

```
Dumping vectorizer...done
Dumping transformer...done
Dumping vectors...done
Dumping target_labels and classes...done
Dumping classifier...done
Self accuracy: 100.00 %
       24630610 function calls (24374600 primitive calls) in 21.783 seconds

  Ordered by: cumulative time

  ncalls tottime percall cumtime percall filename:lineno(function)
    2/1 0.017 0.009 21.794 21.794 bedNbamClassifier.py:2(<module>)
    130 4.702 0.036 14.464 0.111 bedNbamClassifier.py:124(kmerizeAlignments)
  195729 4.847 0.000 8.119 0.000 bedNbamClassifier.py:113(kmerize)
      1 0.003 0.003 5.521 5.521 text.py:792(fit_transform)
      1 1.767 1.767 5.438 5.438 text.py:737(_count_vocab)
    130 0.002 0.000 2.006 0.015 text.py:237(<lambda>)
    130 0.000 0.000 1.738 0.013 text.py:213(<lambda>)
    141 1.737 0.012 1.737 0.012 {method 'findall' of '_sre.SRE_Pattern' objects}
  195859 0.173 0.000 1.377 0.000 calignmentfile.pyx:1683(__next__)
      7 0.000 0.000 1.275 0.182 numpy_pickle.py:351(dump)
      7 0.000 0.000 1.274 0.182 pickle.py:220(dump)
 116577/7 0.127 0.000 1.274 0.182 numpy_pickle.py:255(save)
 116577/7 0.293 0.000 1.274 0.182 pickle.py:269(save)
  19383/4 0.060 0.000 1.273 0.318 pickle.py:345(save_reduce)
    15/4 0.000 0.000 1.273 0.318 pickle.py:640(save_dict)
    15/4 0.025 0.002 1.273 0.318 pickle.py:655(_batch_setitems)
```

Figure 8.1: Report from the Deterministic Profiling of the most Representative Run of the Original Script sorted by the Cumulative Time

Deterministic Profiling - Number of Calls - String Appending

```
Dumping vectorizer...done
Dumping transformer...done
Dumping vectors...done
Dumping target_labels and classes...done
Dumping classifier...done
Self accuracy: 100.00 %
       24630610 function calls (24374600 primitive calls) in 21.600 seconds

   Ordered by: call count

   ncalls tottime percall cumtime percall filename:lineno(function)
5585942/5585336 0.822 0.000 0.823 0.000 {len}
  5199869 1.149 0.000 1.149 0.000 {min}
  5188297 1.177 0.000 1.178 0.000 {max}
  5188104 1.109 0.000 1.109 0.000 {method 'append' of 'array.array' objects}
   291246 0.048 0.000 0.048 0.000 {id}
   216769 0.054 0.000 0.054 0.000 {method 'get' of 'dict' objects}
   196860 0.128 0.000 0.128 0.000 {range}
   195859 0.178 0.000 1.390 0.000 calignmentfile.pyx:1683(__next__)
   195859 1.109 0.000 1.109 0.000 calignmentfile.pyx:1675(cnext)
   195729 0.093 0.000 0.219 0.000 calignedsegment.pyx:981(__get__)
   195729 4.846 0.000 8.078 0.000 bedNbamClassifier.py:113(kmerize)
   195729 0.103 0.000 0.103 0.000 calignedsegment.pyx:494(makeAlignedSegment)
   195729 0.125 0.000 0.125 0.000 calignedsegment.pyx:444(getSequenceInRange)
   195729 0.052 0.000 0.052 0.000 calignedsegment.pyx:645(__dealloc__)
   194324 0.083 0.000 0.083 0.000 {method 'write' of 'file' objects}
 116577/7 0.293 0.000 1.279 0.183 pickle.py:269(save)
```

Figure 8.2: Report from the Deterministic Profiling of the most Representative Run of the Original Script sorted by the Number of Calls

Deterministic Profiling - Cumulative Time - Python List - First Approach

```
Dumping vectorizer...done
Dumping transformer...done
Dumping vectors...done
Dumping target_labels and classes...done
Dumping classifier...done
Self accuracy: 100.00 %
       22856575 function calls (22851257 primitive calls) in 57.474 seconds

   Ordered by: cumulative time

   ncalls tottime percall cumtime percall filename:lineno(function)
     2/1 5.492 2.746 57.485 57.485 bedNbamClassifier.py:2(<module>)
     130 39.568 0.304 51.488 0.396 bedNbamClassifier.py:127(kmerizeAlignments)
   195729 5.897 0.000 10.186 0.000 bedNbamClassifier.py:113(kmerize)
   195859 0.197 0.000 1.438 0.000 calignmentfile.pyx:1683(__next__)
  5188297 1.213 0.000 1.213 0.000 {max}
  5199869 1.162 0.000 1.162 0.000 {min}
   195859 1.124 0.000 1.124 0.000 calignmentfile.pyx:1675(cnext)
  5259266 0.960 0.000 0.960 0.000 {method 'append' of 'list' objects}
5431651/5431045 0.826 0.000 0.826 0.000 {len}
      5 0.004 0.001 0.284 0.057 __init__.py:3(<module>)
   195729 0.096 0.000 0.241 0.000 calignedsegment.pyx:981(__get__)
   196860 0.153 0.000 0.153 0.000 {range}
   195729 0.145 0.000 0.145 0.000 calignedsegment.pyx:444(getSequenceInRange)
     18 0.007 0.000 0.130 0.007 __init__.py:1(<module>)
      3 0.002 0.001 0.118 0.039 __init__.py:4(<module>)
      3 0.001 0.000 0.118 0.039 __init__.py:5(<module>)
```

Figure 8.3: Report from the Deterministic Profiling of the first approach of the Python List optimisation sorted by the Cumulative Time

Deterministic Profiling - Cumulative Time - Python List - Second Approach

```
Dumping vectorizer...done
Dumping transformer...done
Dumping vectors...done
Dumping target_labels and classes...done
Dumping classifier...done
Self accuracy: 100.00 %
       23052422 function calls (23047104 primitive calls) in 12.463 seconds

   Ordered by: cumulative time

   ncalls tottime percall cumtime percall filename:lineno(function)
     2/1 0.132 0.066 12.474 12.474 bedNbamClassifier.py:2(<module>)
     130 0.365 0.003 11.839 0.091 bedNbamClassifier.py:127(kmerizeAlignments)
   195729 5.649 0.000 9.866 0.000 bedNbamClassifier.py:113(kmerize)
   195859 0.151 0.000 1.324 0.000 calignmentfile.pyx:1683(__next__)
  5188297 1.212 0.000 1.212 0.000 {max}
  5199869 1.170 0.000 1.170 0.000 {min}
   195859 1.095 0.000 1.095 0.000 calignmentfile.pyx:1675(cnext)
  5259266 0.956 0.000 0.956 0.000 {method 'append' of 'list' objects}
5627510/5626904 0.846 0.000 0.846 0.000 {len}
       5 0.004 0.001 0.286 0.057 __init__.py:3(<module>)
   195729 0.089 0.000 0.200 0.000 calignedsegment.pyx:981(__get__)
      18 0.007 0.000 0.131 0.007 __init__.py:1(<module>)
       3 0.002 0.001 0.119 0.040 __init__.py:4(<module>)
       3 0.001 0.000 0.119 0.040 __init__.py:5(<module>)
       1 0.000 0.000 0.116 0.116 text.py:13(<module>)
       1 0.000 0.000 0.114 0.114 imputation.py:4(<module>)
```

Figure 8.4: Report from the Deterministic Profiling of the second approach of the Python List optimisation sorted by the Cumulative Time

Deterministic Profiling - Cumulative Time - Character Array

```
Dumping vectorizer...done
Dumping transformer...done
Dumping vectors...done
Dumping target_labels and classes...done
Dumping classifier...done
Self accuracy: 100.00 %
       28436384 function calls (28431066 primitive calls) in 16.605 seconds

   Ordered by: cumulative time

   ncalls tottime percall cumtime percall filename:lineno(function)
     2/1 0.124 0.062 16.616 16.616 bedNbamClassifier.py:2(<module>)
     130 0.398 0.003 15.749 0.121 bedNbamClassifier.py:129(kmerizeAlignments)
   195729 7.280 0.000 13.595 0.000 bedNbamClassifier.py:114(kmerize)
  5383963 1.658 0.000 1.658 0.000 {method 'append' of 'array.array' objects}
  5187973 1.388 0.000 1.388 0.000 {method 'fromstring' of 'array.array' objects}
   195859 0.154 0.000 1.336 0.000 calignmentfile.pyx:1683(__next__)
  5188297 1.227 0.000 1.227 0.000 {max}
  5199869 1.154 0.000 1.154 0.000 {min}
   195859 1.094 0.000 1.094 0.000 calignmentfile.pyx:1675(cnext)
5431781/5431175 0.865 0.000 0.865 0.000 {len}
       5 0.004 0.001 0.284 0.057 __init__.py:3(<module>)
    1752 0.233 0.000 0.233 0.000 {method 'split' of 'str' objects}
   195729 0.089 0.000 0.206 0.000 calignedsegment.pyx:981(__get__)
      18 0.007 0.000 0.131 0.007 __init__.py:1(<module>)
       3 0.002 0.001 0.118 0.039 __init__.py:4(<module>)
   195729 0.118 0.000 0.118 0.000 calignedsegment.pyx:444(getSequenceInRange)
```

Figure 8.5: Report from the Deterministic Profiling of the Character Array optimisation sorted by the Cumulative Time

Deterministic Profiling - Cumulative Time - Pseudo File

```
Dumping vectorizer...done
Dumping transformer...done
Dumping vectors...done
Dumping target_labels and classes...done
Dumping classifier...done
Self accuracy: 100.00 %
       28827972 function calls (28822654 primitive calls) in 16.706 seconds

   Ordered by: cumulative time

   ncalls tottime percall cumtime percall filename:lineno(function)
     2/1 0.123 0.062 16.717 16.717 bedNbamClassifier.py:2(<module>)
     130 0.448 0.003 15.852 0.122 bedNbamClassifier.py:129(kmerizeAlignments)
   195729 7.216 0.000 13.663 0.000 bedNbamClassifier.py:114(kmerize)
 10767404 3.122 0.000 3.122 0.000 {method 'write' of 'cStringIO.StringO' objects}
   195859 0.151 0.000 1.322 0.000 calignmentfile.pyx:1683(__next__)
  5188297 1.231 0.000 1.231 0.000 {max}
  5199869 1.183 0.000 1.183 0.000 {min}
   195859 1.093 0.000 1.093 0.000 calignmentfile.pyx:1675(cnext)
5431781/5431175 0.902 0.000 0.902 0.000 {len}
      5 0.004 0.001 0.286 0.057 __init__.py:3(<module>)
    1752 0.234 0.000 0.234 0.000 {method 'split' of 'str' objects}
   195729 0.088 0.000 0.199 0.000 calignedsegment.pyx:981(__get__)
     18 0.007 0.000 0.132 0.007 __init__.py:1(<module>)
      3 0.002 0.001 0.119 0.040 __init__.py:4(<module>)
      3 0.001 0.000 0.119 0.040 __init__.py:5(<module>)
      1 0.000 0.000 0.116 0.116 text.py:13(<module>)
```

Figure 8.6: Report from the Deterministic Profiling of the Pseudo File optimisation sorted by the Cumulative Time

Deterministic Profiling - Cumulative Time - List Comprehensions

```
Dumping vectorizer...done
Dumping transformer...done
Dumping vectors...done
Dumping target_labels and classes...done
Dumping classifier...done
Self accuracy: 100.00 %
       17867497 function calls (17861750 primitive calls) in 9.926 seconds

   Ordered by: cumulative time

   ncalls tottime percall cumtime percall filename:lineno(function)
     2/1 0.017 0.008 9.937 9.937 bedNbamClassifier.py:2(<module>)
     130 0.367 0.003 9.433 0.073 bedNbamClassifier.py:125(kmerizeAlignments)
   195729 4.161 0.000 7.383 0.000 bedNbamClassifier.py:113(kmerize)
   195859 0.148 0.000 1.311 0.000 calignmentfile.pyx:1683(__next__)
  5188297 1.173 0.000 1.173 0.000 {max}
  5199869 1.150 0.000 1.150 0.000 {min}
   195859 1.092 0.000 1.092 0.000 calignmentfile.pyx:1675(cnext)
5432045/5431439 0.821 0.000 0.822 0.000 {len}
       5 0.004 0.001 0.280 0.056 __init__.py:3(<module>)
   195729 0.087 0.000 0.196 0.000 calignedsegment.pyx:981(__get__)
202051/202042 0.136 0.000 0.136 0.000 {method 'join' of 'str' objects}
      18 0.007 0.000 0.128 0.007 __init__.py:1(<module>)
       3 0.002 0.001 0.117 0.039 __init__.py:4(<module>)
       3 0.001 0.000 0.117 0.039 __init__.py:5(<module>)
       1 0.000 0.000 0.114 0.114 text.py:13(<module>)
       1 0.000 0.000 0.112 0.112 imputation.py:4(<module>)
       1 0.000 0.000 0.112 0.112 imputation.py:4(<module>)
```

Figure 8.7: Report from the Deterministic Profiling of the List Comprehensions optimisation sorted by the Cumulative Time

Deterministic Profiling - Cumulative Time - Linked List

```
Dumping vectorizer...done
Dumping transformer...done
Dumping vectors...done
Dumping target_labels and classes...done
Dumping classifier...done
Self accuracy: 100.00 %
       208280456 function calls (208274540 primitive calls) in 97.757 seconds

   Ordered by: cumulative time

   ncalls tottime percall cumtime percall filename:lineno(function)
     2/1 0.007 0.003 97.768 97.768 bedNbamClassifier.py:2(<module>)
     130 0.436 0.003 97.248 0.748 bedNbamClassifier.py:202(kmerizeAlignments)
   195729 6.539 0.000 94.441 0.000 bedNbamClassifier.py:188(kmerize)
  5187973 52.946 0.000 84.435 0.000 bedNbamClassifier.py:58(insert)
169065008 27.958 0.000 27.958 0.000 bedNbamClassifier.py:36(get_next)
   195859 0.168 0.000 1.368 0.000 calignmentfile.pyx:1683(__next__)
  5187973 1.314 0.000 1.314 0.000 bedNbamClassifier.py:29(__init__)
  5775160 1.298 0.000 1.298 0.000 bedNbamClassifier.py:52(isEmpty)
  5188297 1.255 0.000 1.255 0.000 {max}
  5199869 1.210 0.000 1.210 0.000 {min}
   195859 1.111 0.000 1.111 0.000 calignmentfile.pyx:1675(cnext)
  4992244 1.062 0.000 1.062 0.000 bedNbamClassifier.py:39(set_next)
5432149/5431543 0.854 0.000 0.855 0.000 {len}
   195729 0.586 0.000 0.729 0.000 bedNbamClassifier.py:74(append)
       5 0.004 0.001 0.287 0.057 __init__.py:3(<module>)
   195729 0.094 0.000 0.222 0.000 calignedsegment.pyx:981(__get__)
```

Figure 8.8: Report from the Deterministic Profiling of the Linked List optimisation sorted by the Cumulative Time