System for turnkey analysis of semi-automated genome annotations

Marc Asenjo Ponce de León

Director: Michael M. Hoffman

Supervisor: Mickaël Mendez

Princess Margaret Cancer Centre, Toronto

Tutor: Maria José Serna Iglesias

Computer Science Department

Bachelor Degree in Informatics Engineering

Computing Specialization

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya

July 3rd, 2018

Abstract

English

In the bioinformatics field there is a very important concept called a segmentation, a representation of the genome with differentiated segments being specified for patterns found on some given input data. Biological meaning needs to be assigned to these segmentations by manually running sets of analyses which may need external data, and then visualizing the results. While a lot of tools to do so exist, none of them automate the whole process of analysis and visualization. Furthermore, all automated visualizations available lack the capacity of making it easier for the researcher to correlate the results obtained from different analysis.

The objective of this project is to solve many of these issues by developing a software capable of doing many of these tasks automatically. It does so by downloading data, running analysis and summarizing all results in one single visualization that is easy to read so that researchers can easily conclude biological hypothesis from it. The tool developed, called Segzoo, is open source Python software available to install and execute on a Linux machine very easily.

Català

En el camp de la bioinformàtica existeix un concepte anomenat segmentació que es refereix a una representació del genoma per segments que presenten un mateix patró trobat en una sèrie de senyals de dades. A aquestes segmentacions, però, cal assignar-los un sentit biològic executant una sèrie d'anàlisis que poden necessitar dades externes, i finalment visualitzant els resultats obtinguts. Encara que moltes eines existeixen per facilitar aquest procés, no n'existeix cap que automatitzi tot l'anàlisi i la visualització. A més, totes les visualitzacions creades automàticament que existeixen no permeten a l'investigador correlacionar fàcilment els resultats obtinguts en diferents anàlisis.

L'objectiu d'aquest projecte és solucionar aquests problemes plantejats desenvolupant un software capaç de fer totes les tasques automàticament. Ho fa descarregant dades externes, executant anàlisis i agrupant tots els resultats en una sola visualització que és fàcil d'interpretar per tal que els investigadors puguin extreure'n hipòtesis. L'eina desenvolupada, anomenada Segzoo, és codi lliure en Python que pot fàcilment ser instal · lat i executat en una màquina amb Linux.

Castellano

En el campo de la bioinformática existe el concepto de la segmentación, una representación del genoma separado en segmentos que presentan un cierto patrón encontrado en una serie de señales de datos de entrada. A estas segmentaciones hace falta asignarles un sentido biológico ejecutando una serie de análisis que pueden necesitar datos externos, y finalmente visualizando los resultados obtenidos. Aunque existen muchas herramientas que facilitan este proceso, ninguna automatiza todo el análisis y la visualización. Además, todas las visualizaciones creadas automáticamente que existen no permiten al investigador correlacionar fácilmente los resultados obtenidos en distintos análisis.

El objetivo de este proyecto es solucionar estos problemas planteados desarrollando un software capaz de hacer todas las tascas automáticamente. Lo consigue descargando datos, ejecutando análisis y agrupando los resultados en una sola visualización que es fácil de interpretar para que los investigadores puedan formular hipótesis. La herramienta desarrollada, llamada Segzoo, es código libre en Python que puede instalarse y ser ejecutado fácilmente en una máquina con Linux.

Acknowledgement

I would first like to thank my thesis director Michael M. Hoffman of the Princess Margaret Cancer Centre at University Health Network in Toronto for giving me the opportunity to develop this project in his lab and acting as its director. He consistently helped in setting goals for the project and gave very useful input on the most important decisions to take, while also allowing it to be my own work. He steered me in the right direction whenever I needed it but always gave me the last word on what I wanted to do with the project.

I would also like to thank the project's supervisor, Mickaël Mendez of the Princess Margaret Cancer Centre at University Health Network in Toronto. He always kept close to the development of the project and offered help whenever I needed it. He guided me in both organizing and developing the project, specially during the beginning stages of my stay where I had to learn the most about the scope I would be working with. Being mostly new to the bioinformatics scene, he eased me into the basics and provided useful references to check out. What has been accomplished would never have been possible without him, so thank you so much.

Next I would also like to thank my thesis tutor, Maria José Serna Iglesias of the Computer Science Department at UPC. She was always there to help me with the development of this thesis and provided a lot of useful feedback.

I would also like to acknowledge the whole *Hoffman Lab* that welcomed me and created an environment for me to comfortably work and develop the project. Additionally, I would like to thank the developers and the community I got in contact with and efficiently helped me out whenever I needed.

Finally, I must express my very profound gratitude to my family and friends for providing me with unfailing support throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Contents

List of Figures

List of Tables

1 Context

1.1 Introduction

Computational biology and bioinformatics are two fields of study that find themselves still very much in development and are constantly making progress towards understanding all kinds of biological data such as genomic data. For scientists to make findings in these fields a set of computer tools are necessary to process, analyze, summarize and compare the data. Typically, when talking about biological data, the main problem that needs to be faced is its size, which can lead to enormous amounts of time spent managing it.

The size of the data also means that it is very difficult to analyze manually. It is because of this that all analysis are mostly made by a computer, but it is a challenge to represent the results in a way that is readable and understandable by the researcher. In a way, the tools used need to work as translators so that the results of the analysis can be useful to the research being done.

In this project all these challenges will need to be faced while developing a tool for the bioinformatics community. The data to be analyzed and visualized in this case will be a representation of patterns found in a genome called a segmentation, which can be obtained with the software named Segway explained in detail on following sections. This will be done in the *Hoffman Lab* [1], The Princess Margaret Cancer Research Tower (PMCRT) in Toronto, Canada, under the direction of Dr. Hoffman.

1.2 Objectives

The main objective of this project is to develop a tool that takes care of a big portion of the work that currently a bioinformatics engineer must do by himself. This tool will have to download data, run analysis and plot the results obtained in a compacted visualization. This plot must be easily readable and allow the researcher to analyze the obtained results.

The tool to be developed will compete with a lot of current tools that allow to run this kind of analysis, as we'll see in section 2 of this document. However, this project's objective is to solve some of the issues currently present with available options as well as further improve them. As a consequence, the following characteristics must be present in the tool:

- The analysis must be fully automated and the tool easy to use. Most of the available tools require the scientists to make comparisons or download data by themselves. The objective of the tool to be developed is to resemble a turnkey system with some potential customization options for the user.
- The final visualization must be a compact and easy-to-read plot that allows for cross-result analysis. There are tools available that perform single analysis, but typically it turns out difficult to compare these analysis' results with each

other. In other words, this tool must do a good job at summarizing all the results in a way that they can be co-related.

While these are the initial main objectives, the main focus of this project is to create a tool that proves useful for the bioinformatics community, and so the specified objectives may vary during its development and even more goals may be added.

1.3 Segway

To understand the finality and necessity of this project it is indispensable to first explain Segway [2, 3]. It consists in a software that generates a genome segmentation from input data signals. In the following subsections this description will be explained in more detail by first taking a look at an overview of how it works and why it is necessary, followed by an example showing how Segway operates.

1.3.1 Overview

Nowadays there's a lot of genome-wide data sets representing a variety of properties (see the ENCODE Project [4]) that have been produced by researchers. Consequently, the current representation of the genome is much more than just a sequence of nucleotides.

A segmentation provides a conceptually simple approach to finding patterns in genomic data, finding segment boundaries while assigning labels to them. The result is a partitioned genome into non-overlapping contiguous segments with assignments to a finite set of labels so that regions sharig the same segment label have in common certain properties in observed data [2].

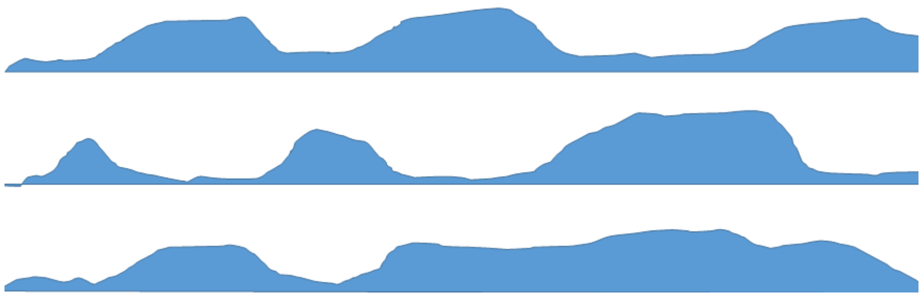
Segway is a software that uses a dynamic Bayesian network (DBN) [5] method to discover patterns in genomic signal datasets, and then transforms these multiple datasets into a simple segmentation, labeling the best pattern at every position in the genome. This turns out to be a very useful tool for bioinformatics scientists when trying to analyze genomic datasets, because it allows for meaningful and simple visualizations of results, from which it becomes easier to assign biological meaning to the data. It is able to detect phenomena associated with low signal in a single dataset that are revealed to be significant when we jointly consider multiple datasets. Other segmentation software tools exist [6], but they generally fail at doing the job as good as Segway does either because of performance issues or because of the quality of the results.

Segway uses a format to store the genomic data it works with called Genomedata [7]. This format allows for quick random access to the data, which can be of very big proportions, and also an efficient storage of multiple tracks of numeric data anchored to a genome, which is exactly the kind of data Segway will be working with. Both Segway and Genomedata have been highly contributed to by Dr. Hoffman.

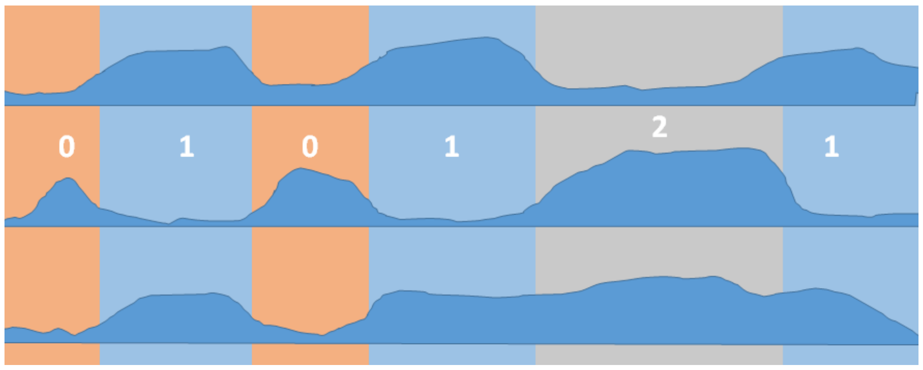
Segway uses the The Graphical Models Toolkit (GMTK) [8] to do expectation-maximization training [9] and Viterbi decoding [10]. The result is a set of learned parameters that will be used in this project, which are called the GMTK parameters. We will now see a brief description of how Segway works.

1.3.2 Operation

The input for Segway is a series of signals that extend throughout all the genome or those parts of it that are of interest, and can represent many different types of data. One of the most typical inputs are histone modifications [11] obtained by ChIP-seq [12]. These signals are called tracks. An example of how 3 tracks can look like is in Figure 1a. In a Segway run, the user specifies a number of labels, for example 10, and then the software will try to identify this number of most-seen patterns, and assign them to the segments generated.



(a) *Signals that serve as input in a Segway run, called tracks*



(b) *Segmentation obtained with Segway*

Figure 1: *Example of a segmentation obtained from three input tracks and three labels*

Segway works as a Neural Network, and so it must go through two procedures: training and annotating. The training procedure is where the label's identity is defined. In Figure 1b we see that label number 0 is associated with low levels of both the first and the third tracks, while it presents high values for the second track. This is what that label's identity is, which actually is represented in the set of parameters called the GMTK parameters. The training is usually performed using a small portion of the full length of the tracks.

After each label is assigned to a set of parameters, the annotating procedure is next. This is what will give the final segmentation, so it is necessary for it to run on all the genome that is of interest. What this procedure does is both identify the limits for the segments and determine the label to annotate them with, so that a specified likelihood is maximized. The results from this are the ones in Figure 1b, which is a case for only three labels (0, 1 and 2). These results can afterwards be analyzed using different tools, some of which are going to be explained in this document.

1.4 Stakeholders

The stakeholders of a project are those collectives or individual people that have a specific role in it, either from the developing or the receiving end of it. These two sides are generally always going to be represented in a project that is based on the development of any kind of tool that can be useful to a collective. This project is no exception in that regard, and so these are the stakeholders that can be differentiated.

1.4.1 Developer

The main developer of the desired tool. He is the responsible for making most of the decisions regarding details of the project like, for example, choosing which software to take advantage of in the development. He is also the one that will need to conduct all needed research in the field of study to make sure the development of the tool advances properly. As the name suggests, he will be the one to develop the tool while listening to the suggestions and feedback of both the director and the supervisor. He needs to make sure the scheduling suggested is followed, and that all the objectives are attained in the end. I am the one who is going to take this role in this project.

1.4.2 Director

The role of the director of the project is to ensure the developer has everything he needs to achieve the objectives specified. He needs to follow the project's progress closely and supply advice and help when necessary. He's responsible of taking the most important decisions in regards to the project. The director of this project is Michael M. Hoffman.

1.4.3 Supervisor

The supervisor has a similar role than the director in this case but will work more closely to the developer. He is the first person the developer should address to for any kind of problems he is facing with the development of the tool. He can have a big voice in decisions of importance but in many cases the opinion of the director is also necessary. The supervisor of this project will be Mickaël Mendez, PhD student in the *Hoffman Lab*.

1.4.4 Community

The computational biology community. This includes the rest of the *Hoffman Lab*, which is likely to be interested in this tool once it has been developed, It also includes all the possible users all around the world that could be interested in the analysis that will be made more accessible thanks to the tool developed in this project. However, in this huge, heterogeneous collective we can differentiate two different kinds of stakeholders:

- **Users**

  As it has been already mentioned, the users of the resulting tool are a very important stakeholder. Not only is this collective going to use the tool, but it will also provide a much valuable resource: feedback. The user is the target collective, and as such it is of interest to apply all the suggestions that can come up from it, to improve the tool significantly. The potential users for this project are most probably the current users of Segway or other similar segmentation methods.

- **Developers**

  From this community, aside from the users, we can distinguish the collective of the developers. This project will be open source, which will allow for anyone to not only install and use the tool, but to check how it's been developed too. The community that looks into these details will be able to give a different kind of feedback to the project, more technical than the more "utility oriented" feedback from the users. In this collective they can also be considered the developers and maintainers of all the tools and utilities that will be taken advantage of in the development of this project.

2 State-of-the-art

For this project's purpose in particular it is very important to keep in mind the actual state-of-the-art. We want to take a step forward in automating analysis and visualizations, but using existing tools is probably the best solution for most of the analysis. Furthermore, having an idea of how the current visualizations look like may proof very helpful in the building of this tool, because it generally is a good idea to visualize things in a way that researchers are already familiar with. In the following sections we will first take an overall look at the currently existing methods to conduct research on segmentations, before taking a closer look at the two tools that are used the most in the *Hoffman Lab* and will primarily serve as reference for this project.

2.1 Existing methods

While the primary existing tool that is going to be used for the project is Segtools, explained in subsection 2.2, there are many more tools that provide useful analyses. The University of California, Santa Cruz (UCSC) Genome Browser allows for researchers to explore the relationships between a segmentation and known annotations but, as mentioned in subsection 2.3, this has some limitations when considering the scale of a complete genome.

The Galaxy platform [14] and BEDTools [15] are tools that provide useful large-scale automated analysis, which is the kind of analysis we are interested in for this project. However, these methods miss some comparisons and don't plot visualizations which are critical to understanding the results obtained from the segmentation. Other utilities like SAMTools[16], which works with the different file format SAM, help in visualizing and post-processing DNA sequence read alignments.

A series of publications are relevant to the background of this project. Both 'rapid and reproducible chromatin state StateHub-StatePaintR: evaluation for custom genome annotation [version 1; referees: 1 approved with reservations]'[17] and 'Discovery and Characterization of Chromatin States for Systematic Annotation of the Human Genome' [13] are good examples on how analysis are done currently. Specifically, 'Discovery and Characterization of Chromatin States for Systematic Annotation of the Human Genome' is a clear example of many of the different visualization possibilities for representing results and, in particular, the visualization in Figure 2 is taken into consideration as an example of what wants to be accomplished in this project. The reason why this figure is of interest is because of its ability to convey a huge amount of information in one single plot. While this plot has been computed manually or in a semi-automated manner, the objective is to obtain something similar to this in a completely automated way.

Finally, the publication 'Reproducible genomics analysis pipelines with GNU Guix'[18] describes PiGx [19], a platform that hosts many reproducible bioinformatics pipelines that present some similarities with what we want to accomplish with this

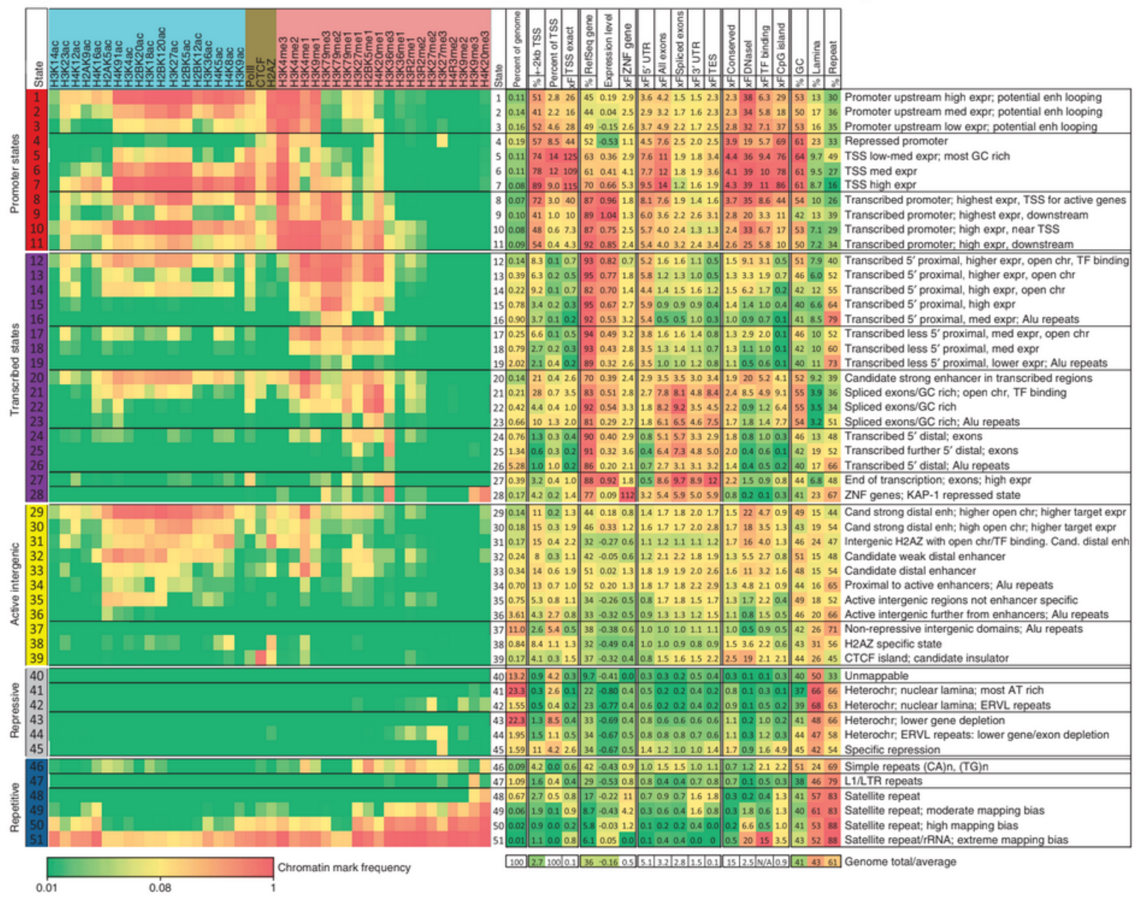


Figure 2: *"Chromatinstate definition and functional interpretation". [13] Visualization for each label (y-axis) of the parameters assigned to each track (left) and a set of attributes of interest (right)*

project. This article puts special emphasis on the reproducibility of thee pipelines. Typically bioinformatics tools tend to present a lot of dependencies and these can make it difficult to make an analysis automatic. While they use GNU Guix packages to take care of these problems, it will be necessary for the development of this project to consider possible alternatives to solve this issue.

2.2 Segtools

Also contributed to by Michael M. Hoffman, Segtools [20, 21] is a software toolkit that facilitates the exploratory analysis of genomic segmentations. It is designed to provide segmentation-centric summary statistics and visualizations, in a manner that is scalable and easy to use [20]. This tool can run a series of analysis on the obtained segmentation and obtain visualizations that allow the researcher to interpret the results and assign biological meaning to them. All these utilities are available both through command line commands and a Python API. In the following subsections we describe the most relevant tools to this project.

2.2.1 GMTK parameters

The GMTK parameters are learned by Segway during the training phase and are afterwards used to segment and annotate the genome, obtaining the final segmentation. These parameters represent the values for each of the input tracks that are related to each one of the labels.

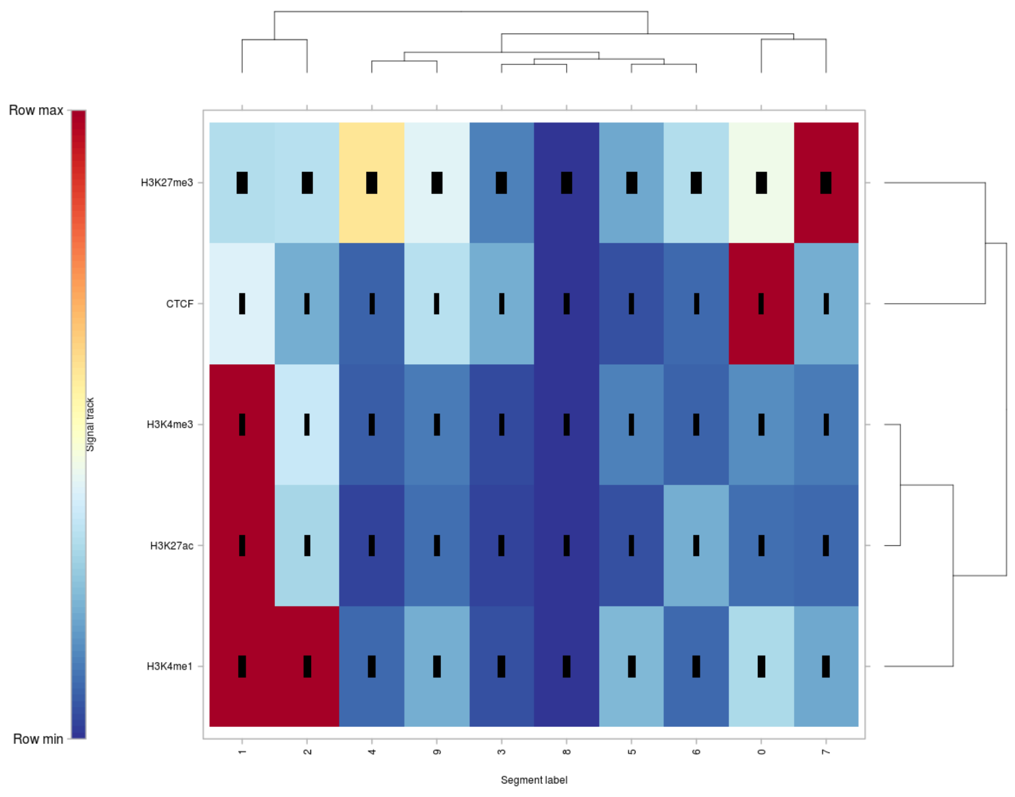


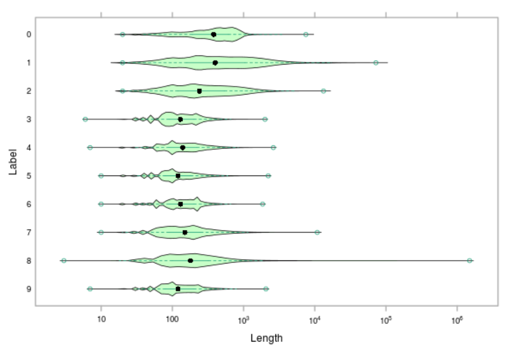
Figure 3: *Heatmap of the GMTK parameters obtained in training. The color represents the mean value and the size of the black square, the standard deviation*

For example, in the heatmap of Figure 3 we can see that label 1 is associated with high values of H3K4me3, H3K27ac and H3K4me1 (represented in red) while it presents low values for H3K27me3 and CTCF. This means that a region in the input that represents these levels for each of the 5 tracks mentioned will likely be annotated as label 1. So we can conclude that, in a way, the GMTK parameters define what each of the labels of the segmentation represent.

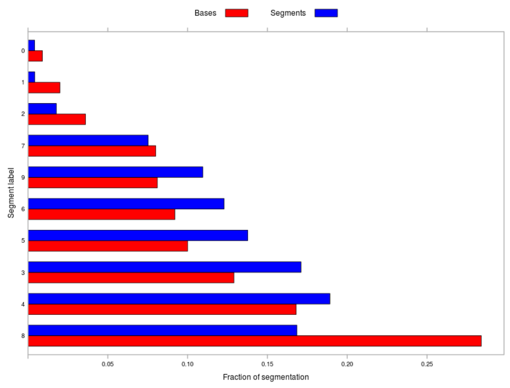
This tool also outputs information about the transitions between label segments. A heatmap is plotted showing how often a label's segments comes after another label's segment. This can be useful to deduce biological meaning of a label from another's.

2.2.2 Length distribution

This tool allows to better understand how the segmentation results look like. It specifies statistic information regarding the length and the number of segments for each label. From executing this tool two visualizations are obtained, as seen in Figure 4.



(a) The length distribution of the segments obtained



(b) Representation of the segment sizes by label

Figure 4: *Visualizations resulting from the tool length distribution*

In Figure 4a we observe the distribution for the lengths of the segments of each label. In this case we can see that the mean value for each label, represented with a black dot, is of the same order of magnitude, but higher for label 1. Note that the x-axis has a logarithmic scale. This information can be useful to see whether each label is associated to longer or smaller segments.

In Figure 4b we can see the percentage of total segments and of total base pairs that each of the labels cover. This is useful to better understand what each label represents by knowing how prevalent they are in the segmentation. In this case we can see that label 8 takes up most bases, because of some long segments that we can see in Figure 4a too. Combining this information with the one in Figure 3, we can deduct that label 8 represents areas without input signal for all the tracks, or areas with very low signal, and that the studied genome presents very large segments of this kind.

2.2.3 Aggregation

This is one of the most versatile tools in Segtools. The command looks at the aggregate occurrence of segment labels around and within a set of annotations. The necessary inputs for this command are the segmentation obtained with Segway and an annotation file, that specifies annotations for some regions of the genome. One clear example is that of genes. The annotation file must contain information on where genes and exons are found in the genome. If this is the case, then the aggregation is done using an idealized gene model of 3 exons and 3 introns, plus the *flanking* regions, which consist in a number of base pairs before and after the gene. All these

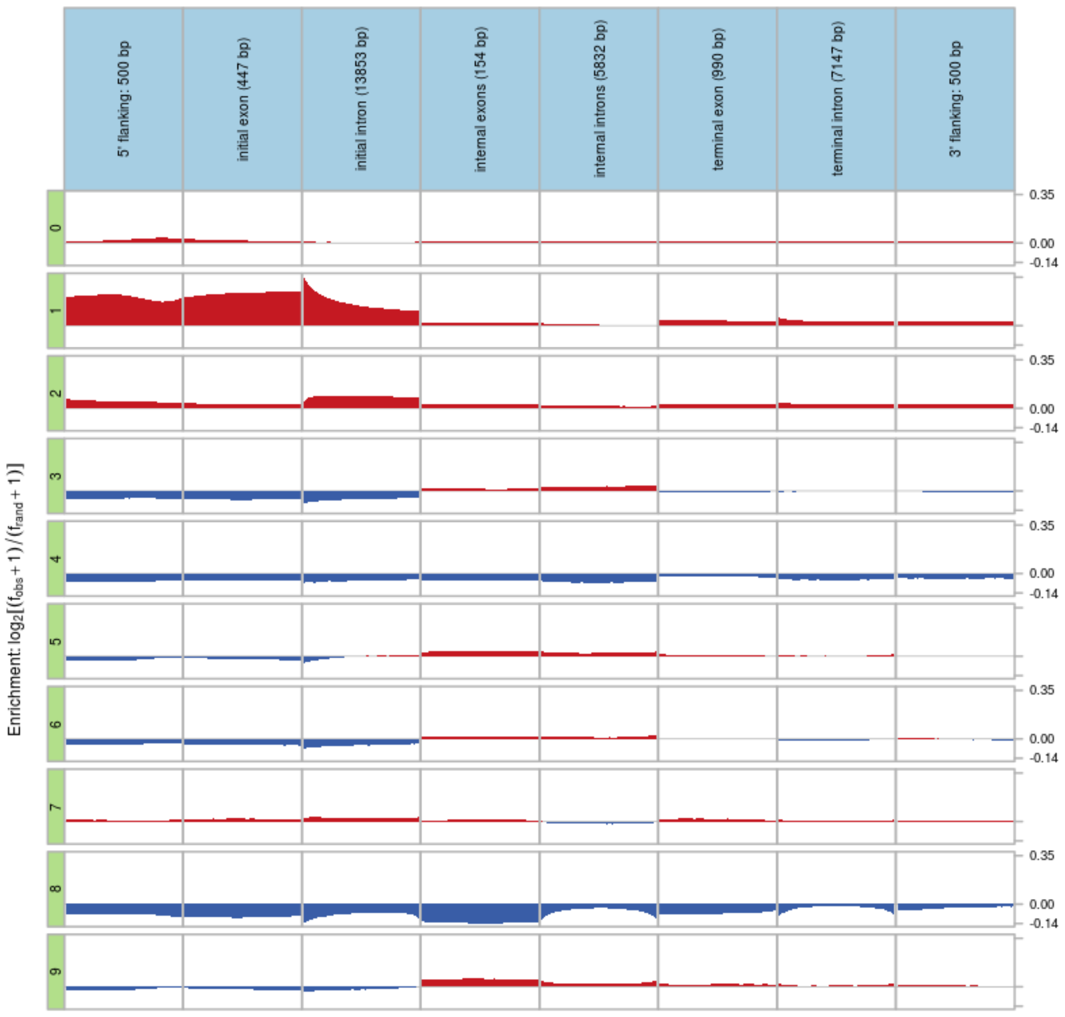


Figure 5: *Graphs of the enrichment of each labels's segments with the different features of an idealized gene*

regions are called *components*. The results resemble those in Figure 5. For each label and component a graph of the enrichment is shown.

This tool is very useful when trying to assign biological meaning to each label. When using it in the *gene mode* that has been described, some interesting conclusions can be drawn, like that a certain label is only found in the initial components of a gene, like the case of label 1 in Figure 5. It can also be observed whether the label's segments are found more frequently in exons or in introns. While this is the functionality that is more relevant for this project, as it has mentioned before this tool is very versatile. The annotation used can consist on many different things that can prove interesting for the analysis.

2.2.4 Feature distance

This tool takes a segmentation and one or more annotation files and prints the distance from each segment to the nearest annotation entry, called a feature, in each file [21]. The way this command works is similar to the aggregation, because it also compares the segmentation with another annotation. In this case, the output is a distance to the feature, which could be a gene for example. If this distance is zero, it means that the segment is overlapped with one of the features. A histogram of these distances is shown per label in a simple visualization when executing this tool.

2.2.5 Other tools

Other tools exist, but they either are less used overall or not relevant for this project right now. An extra utility is given in Segtools, which consists in generating an HTML file with all the different results from all the already executed tools. These results are presented one after another in different sections, after some basic information regarding the segmentation used. This summary has its limitations, as it is not easily readable and the results from the different tools cannot be easily compared with each other.

Segtools does not present any way to automatically generate all the analysis results. They all need to be run one by one and, furthermore, all necessary data must be downloaded manually beforehand. Also, as mentioned previously, there's no way of visualizing all the results at once in a way that makes them easy to correlate. These are all issues this project will be trying to solve. As stated in 'Exploratory analysis of genomic segmentations with Segtools', "Segtools analyses are easy to perform, script, and incorporate into existing analysis pipelines, making them useful for both manual and automated exploration". Due to our interest in running automated exploration, the use of Segtools for our project can prove useful.

2.3 UCSC Genome Browser

While Segtools is very useful in obtaining summarizing analysis of different sorts, it may be sometimes useful to look at specific positions of the genome for the segmentation and compare it to other annotations, just like *segtools aggregation* does. If this kind of analysis is desired, the UCSC Genome Browser [22, 23] provides it. This tool consists on a web interface that allows the user to upload the obtained results from the segmentation and that has access to different datasets of annotations [24].

The results obtained from using this tool tend to look like in Figure 6. The first 5 signals shown are the tracks used as input for the Segway run, that in this case are histone modifications signals obtained by ChIP-seq. Following them, the segmentation is shown, with every segment from a different label being colored differently. From this point, all sorts of annotations can be added to the visualization, like genes, exons or transcript sites. This allows the user to manually move throughout the genome and compare the segmentation and the input tracks to all sorts of information,

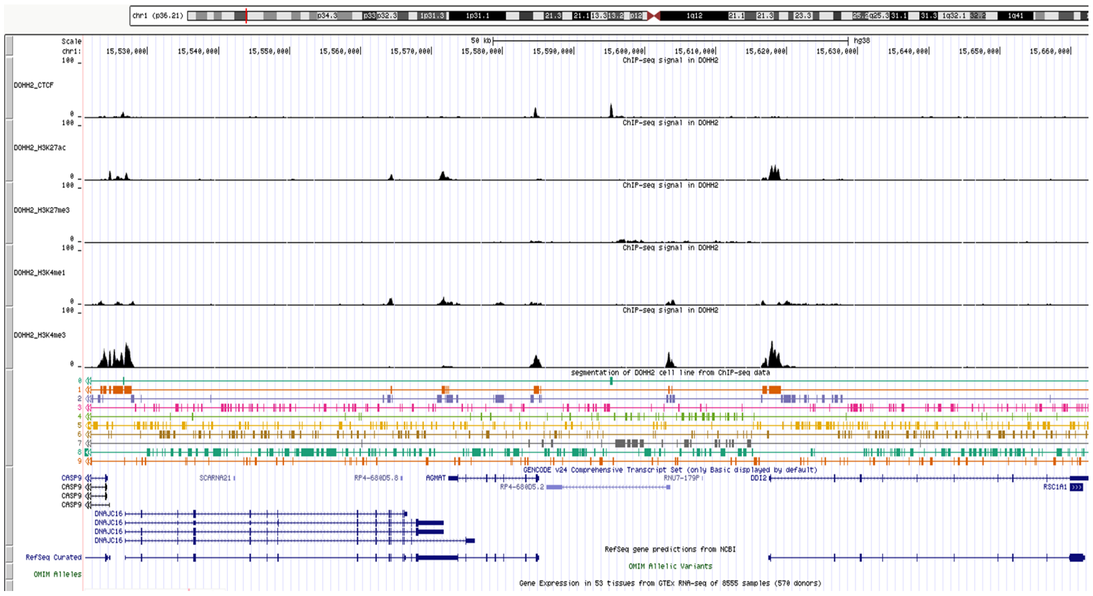

chr1 (p36.21)
Scale
chr1:
50 kb
hg38
ChIP-seq signal in DOHH2
DOHH2_CTCF
DOHH2_H3K27ac
DOHH2_H3K27me3
DOHH2_H3K4me1
DOHH2_H3K4me3
segmentation of DOHH2 cell line from ChIP-seq data
GENCODE v24 Comprehensive Transcript Set (only Basic displayed by default)
CASP9
DNAJC16
SCARNA21
RP4-680D5.8
AGMAT
RP4-680D5.2
RNU7-179P
DDI2
RSC1A1
RefSeq gene predictions from NCBI
RefSeq Curated
OMIM Allelic Variants
OMIM Alleles
Gene Expression in 53 tissues from GTEx RNA-seq of 8555 samples (570 donors)

Figure 6: *Snippet of the results obtained from the Genome Browser*

being able to come up with deductions like a label being present mostly at the start of the genes (promotor regions).

The limitations in this case are different from those of Segtools. The UCSC Genome Browser already possesses all necessary data for the comparison, however this one must be done manually by the user, and that way only a certain portion of the genome can be analyzed at once. This can be useful in specific cases where a very specific kind of comparison must be made and in a certain area of the genome. Nevertheless, the tool we are looking to develop has as an objective to summarize as much as possible the results in order to make it easy for the user to extract conclusions from it.

3 Scope and methodology

3.1 Scope of the project

One challenge in creating any software toolkit is to define the scope of the project, treading a line between solving many problems and solving a few of problems well. As mentioned in the objectives in subsection 1.2, the finality of this project is to design and implement Python software in a Linux environment to further automate exploratory analysis of semi-automated segmentations. Segtools can perform many of the individual comparisons and analysis needed but identifying, downloading and pre-processing appropriate independent annotations is currently left up to a user. Not only this, but currently there's no good automated way to visualize all the obtained results in a way that they can be correlated. This tool to be developed will need to perform three main tasks, explained in the following sections, in a totally automated way. Figure 7 is a representation of the workflow to be executed.

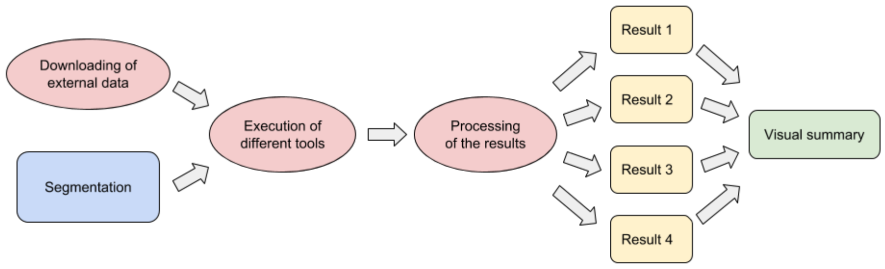

Downloading of external data
Segmentation
Execution of different tools
Processing of the results
Result 1
Result 2
Result 3
Result 4
Visual summary

Figure 7: *Worfklow that the developed tool will need to go through. Squares represent data (files) while circles represent actions (executed scripts or code)*

3.1.1 Downloading

The first step to run the analysis is to download all necessary data for it. For example, the software will download external annotations such as gene models and datasets on gene expression, mutation prevalence, and conservation. All these data must be obtained from a known, trustworthy source in an automated way so that if new versions of the data are uploaded, those are downloaded instead by the software. We don't want any hard-coded information like links or paths in the code so that all possible changes in the database don't affect the tool, or that at least they represent a minimum change needed in it.

Once the source to get this data from is decided, there's one more thing to care about. which is the location where this data is going to be downloaded to, and the possible processing it may need. While this tool will ideally be used multiple times by a researcher using different segmentations as an input, most of the data downloaded will be valid for more than one segmentation. For example, a given human genome assembly, which contains all the genome's base pairs of nucleotides, will be useful for all the segmentations that are using that version of the human genome. Consequently, most of these data will need to be stored in a shared space where all the runs of the software can find it so that it's not downloaded all over again.

3.1.2 Processing and analysis

The next step for the software is to use the downloaded data and the given segmentation to produce the results for the wanted analysis. To do that, the software will run other known software like Segtools to compare annotations and datasets to a researcher's semi-automated genome annotation of interest and to obtain other sorts of results that we want to be shown in the final plot like, for example, the GMTK parameters. Some of the data will probably need to be processed before this step to match the inputs necessary for each software used.

After running all the required analysis, the result will be a set of outputs. These outputs, however, may or may not be what we need, and so will require a post-processing. The data will be computed into tables of the shape that is of interest for the final visualization. All these data will be stored in a local path from where the tool has been executed, so that it doesn't interfere with other analysis that need to be run in the future.

The software that is going to be executed will primarily be Segtools because of the fact that it can already produce many of the analysis that can be of interest. To take care of all the processing of the data Python libraries and utilities will be used.

3.1.3 Plotting of a visualization

The final step for this tool is to compute a summary of all the results obtained. This may actually be the most important step, because the way this visualization is presented will determine whether or not this tool can be useful for the scientists for more than just running a bunch of analysis automatically. One of the main objectives from this project is to make this final result from the workflow an easy-to-read visualization that makes it easy to correlate different results.

It will be necessary to decide which information is displayed and how, maybe allowing for customization from the user to some extend. There are many ways to plot a visualization of this sort but, as stated before, it is best to follow already established standards to make it easier for scientists to interpret the results. Examples like the one of Figure 2 will be heavily considered when taking decisions as they are a representation of what is established right now as a good visualization of results of this kind.

3.1.4 Availability

To obtain the feedback desired from the community this tool will need to be available online. There are many ways to share a project like this, so a decision will have to be made on the approach to take. The tool will be developed using Pyhton, so the initial objective is for it to be a Python Package and available for installation online.

Moreover, a documentation will need to be produced for the tool. The extent of it will be decided during the development of the project, but the guidelines that will be followed are those from 'Creating great documentation for bioinformatics software' [25].

3.2 Methodology

While the objectives of the project are very clear, there are many things that are still to be decided during the development of the project. Knowing this and that the amount of time available for it is very limited, the best approach for the project is to follow an agile methodology. Agile methodologies provide a lot of flexibility to a project and allow to obtain good results in low amounts of time. While not all the main characteristics from agile development are applicable to the project we will now see those that do.

3.2.1 Characteristics

In this project there will be a weekly meeting with the director to constantly update the current goals. This iterative approach makes it much easier to keep in mind the current state of the project. By doing this it will be possible to closely follow the schedule proposed and apply changes to it immediately if necessary. However, small updates to the goals will be continuously made throughout the week when they are mandatory.

To support the method described every decision, update to the goals or progress will be documented into a diary-like document called the *Lab Notebook*. With one entry per week, all progresses and obstacles encountered will be written, as well as including links to scripts and images that are related. At the end of each entry, a special *Next* section will be included in which the current goals being worked on are specified. With the help of the notebook it will be easy to keep track of all the progress and the goals achieved.

Related to the idea of having a short development cycle is the concept of iterative programming. This concept refers to the method of developing a program by creating a first functional version of it before trying to add all future functionality. By doing so, one can make sure that the first version is working before moving on to adding more things to it. This is much safer and easier than trying to develop the final version from scratch and, because of the fact that debugging it proves to be much easier, it can even be quicker. Not only the goals will be set in an iterative way but the development will be too.

Finally, the concept of client feedback from agile methodologies is also applicable. As mentioned in subsection 1.4 the feedback from both the users and developers will be crucial once a first version of the tool is available online. In this case, moreover, this feedback will also be reflected in the director and the supervisor's feedback because of them being part of said community, which will make this concept even more present in this project.

3.2.2 Tools to be used

The main programming language for this project will be Python, with occasionally some Shell scripting. Going beyond that, it will be useful to take advantage of existing tools and libraries for most part of the development of the tool.

The first one is Snakemake [26, 27], a Python tool to easily develop workflows. It uses a Python-based language which is easy to read, and most importantly, takes care of all the file dependencies in a workflow. By specifying inputs and outputs for different rules, and a code to run when executing the rule, Snakemake takes care of all the rest. When requesting a file, it will automatically calculate which files are missing for it to be created and the way to obtain them, recursively. This way, if the workflow is specified correctly, only the necessary files will be created and so only the necessary code will be run. Furthermore, Snakemake takes care of parallelization and optimization of the workflow. It is a known tool in the bioinformatics community where it is widely used, which ensures that support for it won't be dropping any time soon. It can be executed from the command-line but it also presents a Python API. A second option that was considered to develop the workflow is BPipe [28], which does a similar job making it easy to identify errors and restart jobs and in making the pipeline more robust. However, it is based on shell scripts which makes it less adequate for this project.

The second tool is Go Get Data (GGD) [29], which has an Anaconda [30] channel that houses conda [31] recipes for genomic data. GGD will be what we use to obtain the data we want to download in the first stage of the workflow. Automated tests ensure data quality and fidelity to the specified genome and build, which refers to the version of the genome used (hg19, hg38...). It is possible to add new recipes to the channel, which essentially is like adding data to a database. When wanting to download a specific file you can just execute a conda shell command that will do so. This makes it easy to automate the download while having some variables like the genome build used, which is what is of interest for this project. Another possibility was considered for this functionality too, GenomePy [32]. It supports both command-line instructions and a Python API, and is very easy to use. However, the source from which the data is retrieved is not as trustworthy and is limited.

Aside from these two fundamental tools, various Python libraries will be needed when dealing with the data. Pandas [33], Seaborn [34] and Matplotlib [35] can prove useful for both data manipulation and the visualization aspects of the project.

As monitoring tools, BitBucket [36] will be used with Mercurial [37] for the development of the main tool, under the *hoffmanlab* team. For other purposes, GitHub [38] and Git [39] will be used for the same reasons. Both Mercurial and Git are revision-control tools that allow the developer to document all changes to a project in a short term way and keep those changes tracked to be able to revert them if needed. BitBucket and GitHub provide remote repositories with integrated issue tracking systems and more utilities to make it easy for the user to track the project's progress.

3.3 Possible obstacles and solutions

For the methodology to be completely defined it is important for it to cover all possible obstacles found. In the best case, a solution must always be defined which either prevents or acts on the issues found along the project. In the following subsections different obstacles will be described along with the solution suggested for each of them.

3.3.1 Scheduling

Some months may look like a lot of time, but without good scheduling the project will never achieve its objectives. The weekly meetings with the director will substantially help in ensuring that the projects keeps the set schedule or that, if needed, the correct changes are applied to it in case of something unexpected affecting the project. Furthermore, the closer supervision of the supervisor will ensure that the weekly objectives are achieved.

3.3.2 Bugs

This project's main objective is the development of a tool, which also means developing software. As with all software, bugs will be appearing during the process. The problem of finding this bugs will be solved by running a series of tests each time a new version of the software is finished, which will consist basically in running the software completely or partially and check the results obtained, comparing them to the theoretical results. However, most of the bugs encountered will probably cause the software to crash instead of giving wrong results. To correctly identify these bugs a set of debugging tools will be used, like the addition of breakpoints and watchpoints to the code.

3.3.3 Optimization

While this is not the most important characteristic for this project, the software must be optimized. because of the large size of the data it will be working with, the difference between the software being optimized or not could mean a very significant difference in execution time and/or space occupied. By running constant tests and always having in mind the need of this optimization it should be possible to, little by little, change the code into an optimized version of it.

3.3.4 Compatibilities

Many different software will be executed by the developed tool so there will be a lot of dependencies. These dependencies will bring further dependencies in a recursive way, which may cause trouble regarding all the software being compatible. The most basic and probably the most important compatibility issue that may be found because of the development in Python will be the difference between Python 2 and Python 3. To deal with these issues, a clear list of dependencies must be kept updated and, if possible, the tool should be made compatible with all of them. If this is not the case, some dependencies may have to be dropped and other solutions may have to be found.

3.3.5 Third party issues

As mentioned, a lot of third party software will be needed for this project. While most of the tools used are well-known and supported by a large community, some issues may be found within them. All these issues that are not related specifically to the software of this project will be considered as third party issues. To deal with them, the first thing to do if possible is to contact the responsible for that software to try and fix the problems. If it is of maximum importance, the pertinent changes may be done directly by me. If none of these are possible, it may be necessary to find workarounds to the issue or even drop the need of that software.

4 Project planning

Having a good planning is essential to any project. It facilitates always knowing which is the current state of the project compared to the objectives stated, which at the same tame allows for a more efficient repartition of time for each task to do. In this section we will see what is the schedule planned for this project, with a description of all the tasks to accomplish and an analysis of the time needed for them. Alternatives will be studied in case issues are found in the way of the development.

4.1 General guidelines

The staring date for this project was 14th of February, while it's deadline is the 21th of June. The estimated length of the project is about 4 months, considering 8 hours of work a day 5 days a week, but the start and end dates are those of the stay in PMCRT in Toronto, Canada. Consequently, this is the time period in which the project length can move and vary.

While the initial estimated duration is of 4 months, the fact that agile methodologies techniques are used means that this duration is subject to change due to the constant updates on objectives for the project. Furthermore, the possible obstacles found in the way, even if surpassed, may mean a delay in the finalization of the project. An important concept to have in mind from agile methodologies is that the final state of the project may change because of the time constraint. This means that if, due to different factors, the initial objectives of the project can not be achieved, it may be decided to work on finalizing and perfecting some of the aspects of the project instead of new ones.

4.2 Task description

This project can be divided into 9 different tasks that will need to be accomplished for the final objectives to be fully achieved. A description for each of them is provided in this section so that the reach of the project can be better understood.

4.2.1 Define tools to use

The first thing to consider for this project is what tools will be primarily used to develop the desired tool. As mentioned in the methodology, there are many sections of this project that require the use of external tools to facilitate the development and also for optimization reasons. A tool that allows for easy development of a workflow is needed, as it is complex to deal with a file system from scratch. Another tool for the downloading section of the tool is needed to ensure the automation and the veracity of the downloaded files. Other tools and libraries may be needed to take care of other sections of the project, like dealing with big amounts of data, processing it and visualizing it. For this task the only resources needed are a computer and internet, like with the rest of tasks described, and the developer as a human resource to look for the best solutions.

4.2.2 Test tools and first workflow

Once the tools that will be used are defined, Segtools [26] and GGD [29] in this case, the first step is to build a simple workflow to test both of them. The workflow will be written using Snakemake, while one of the rules that it specifies will be downloading a file using GGD. For this first version of the workflow a single analysis will run. Said analysis will be segtools-aggregation (seen in subsection 2.2), because it requires an external annotation file that needs to be downloaded, and it is one of the most useful tools in Segtools and will likely be present in the final visualization for this project. With this workflow we want to make sure all the software we want to use works correctly and that the first results obtained are also correct. The human resources are for the developer to develop the workflow and the director and supervisor to help with decisions.

4.2.3 Add analysis

When a first version has been tested and debugged, we can add more functionality to the workflow. It will be in this stage that more analysis, developed by other Segtools tools or external software, will be added to the workflow. It will be necessary to decide which information we want to obtain next that may be useful for the final visualization. The addition of more analysis will likely require the addition of more external data which will be downloaded using GGD. No other resources are needed than those specified in the last task.

4.2.4 Process results

Once a good amount of data is already being obtained by the workflow, the next step is to process it to convert it into readable data that can be used directly by the visualization. Some standardization of the results will be necessary, so it will have to be decided how all these data is finally stored for the visualization to be created from it. A first approach is to create, for each of the analysis, a table that summarizes the results. In most of the cases these results will have the labels either on the y-axis or the x-axis, so the selection between these two will also need to be standardized. To complete this stage some other tools will be needed, for example Pandas [33], a Python library to easily manage data. No other resources are needed.

4.2.5 First visualization

Finally, after making sure all the data is being obtained automatically and correctly when running the tool, a first visualization can be developed with said data. The contents of this initial plot will depend on the decisions taken regarding the analysis to be included in the workflow. It is important to state that this stage does not intend to obtain a final version of the visualization so, to stick with the schedule, it is not necessary to expend too much time trying to optimize it yet. In order to do this new tools may be needed, like Seaborn [34], a Python library to create visualizations and plots that also uses Pandas. The same human resources apply here too.

4.2.6 Upload Python package

After all the completed stages a first version of the tool will already be usable. Once this point is reached, the first priority is to make it available online for download and installation. The first step to achieve this will be to create a Python package that contains the developed workflow and scripts and runs them when executing some command, for example using the command-line interface. Once this package has been tested locally and works fine, it will then be uploaded to PyPi [40], a repository of Python packages that allows users to install them by running the command `pip install package-name`. The human resources needed here are for the developer to do what has been described.

4.2.7 Bioconda upload

Bioconda [41] is a channel from Anaconda [30] where bioinformatics recipes are stored. Uploading the tool here has several advantages. This will add more visibility to the tool, while also helping with the requirements handling of the tool, making it even easier for the user to install and use the tool. Once this stage is completed, it will be possible to download the tool by using `conda install -c bioconda package-name` on the command-line. No other resources are needed than those for the Python packaging.

4.2.8 Finalize workflow and visualization

When the tool is available already to install online easily and feedback can already be obtained, it is the time to do some final tweaks to the workflow if needed and implement all the useful feedback obtained. Regarding the visualization, now it is the moment to complete it and make sure that it accomplishes all the stated objectives in subsection 1.2. Because of this, the amount of time needed for this stage could vary largely depending on the feedback obtained and the state the tool was left at before starting the packaging and uploading stages. The human resources needed are the same as in all the development stages.

4.2.9 Optimize code

Finally, the pipeline created will need to be optimized both in execution time and disk space occupied. That said, the optimization must have been taken into account during all the development, and so the only things left to do in this stage should be minor tweaks to the tool. While it would be ideal to have this stage, this is the only one which is not necessary to accomplish the objectives for this project, and thus it can be left out if necessary by the agile methodology used. The human resource in this case is the developer's work.

4.3 Estimated time

In Table 1 an estimation of the number of hours dedicated to each task is shown. The total amount of days available for the development of the tasks described is 90, which adds up to 720 h available. We can observe that the estimated total time for the project is lower so it is feasible.

Task	Estimated duration (h)
Define tools to use	30
Test tools and first workflow	100
Add analysis	30
Process results	50
First visualization	100
Upload Python package	150
Bioconda upload	50
Finalize workflow and visualization	100
Optimize code	50
Final stage	30
Total	690

Table 1: *Estimated time needed for each task described*

4.4 Gantt chart

Figure 8 shows the planning of the different tasks of the project in a Gantt chart. Aside from the tasks mentioned before, the writing of the final report and the final presentation have been added to the graph.

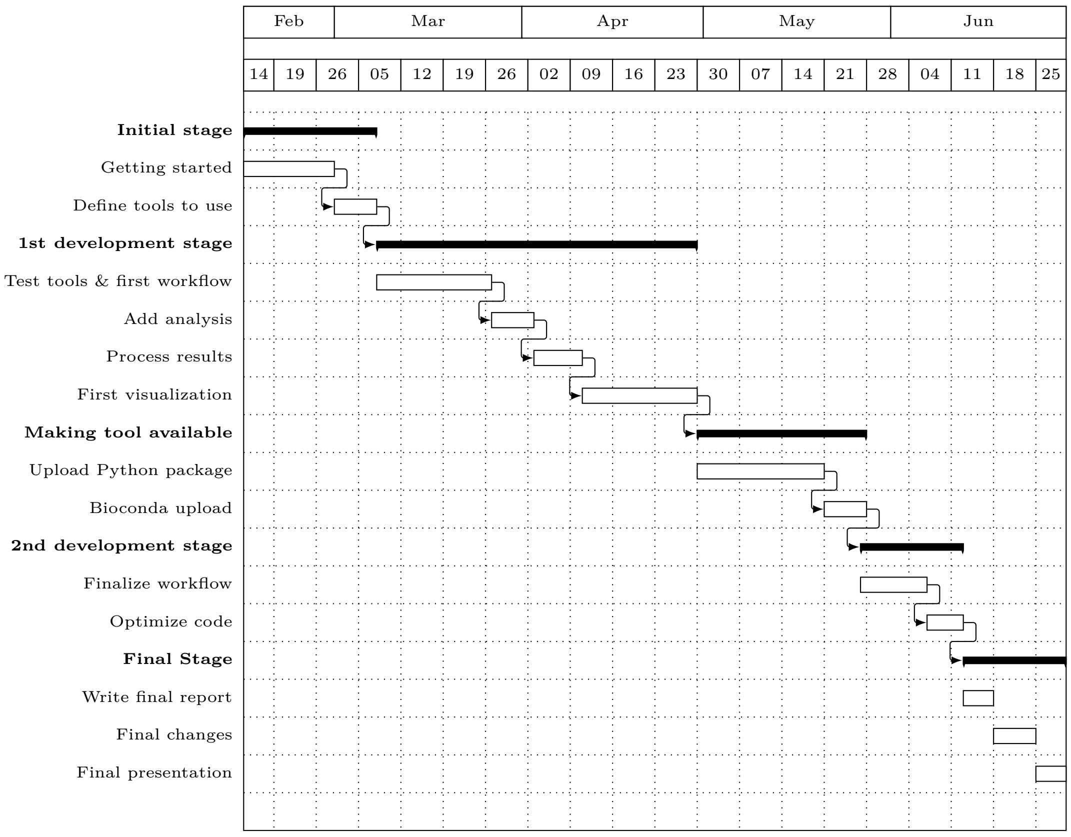


Figure 8: *Initial Gantt chart of the project*

4.5 Alternatives and action plan

There are already two factors that take care of potential changes in the scheduling. The first one is the agile methodology applied, which should allow for quick and efficient decision making and change of the goals. The second one is the estimated times used, which also include an extra margin for possible unexpected obstacles that may appear. However, it is necessary to have a more accurate planning in place for each of the possible obstacles so that the response to it is faster and more efficient whenever they appear.

Most of the basic issues and how to handle them have been specified in subsection 3.3. All these obstacles should be possible to overcome during the time margin applied to each task's estimated time. However, the compatibility issues and the third party issues can turn out to be more difficult to solve than previously stated. Dropping the dependencies may not always be an option, and so it may be necessary at some point to invest a big amount of time in solving this kind of issues. If this is the case, the final optimization stage, as it has been mentioned before, can be skipped as it is not necessary for the objectives of this project to be fulfilled.

Another possible issue that may be encountered is the time needed for the packaging and upload of the tool. This stage comes with a bit of learning with it and, furthermore, the speed of it may ultimately depend on third parties. This could be caused by two main reasons: the first one, that uploading to Bioconda means getting a repository pull request accepted, which is done by the Bioconda team; and second, these uploads are only viable if all the dependencies are right, which ties to the issue mentioned before. If this was the case, the order of the planning proposed could be a little bit modified. While it's optimal to have the tool available online as fast as it works properly, the second development stage could be started before. This way, once the upload can be done the current version in development could be uploaded, and then the development could continue further.

With all this, it is important to keep in mind the agile methodology concepts to make sure that the main objectives are achieved in the end, while maybe having to prioritize them over other things occasionally.

5 Budget and sustainability

5.1 Budget

Although no money will be explicitly invested into the development of this project, the resources used must also be taken into account, and so a budget is necessary for this project's definition. In this section different sorts of budgets will be studied to obtain a final budget that will need to be kept at the end of the project.

5.1.1 Direct budget

In Table 2 and Table 3 the direct budgets mentioned in subsection 4.2 are estimated. For software and hardware budgets the useful life of the product is taken into account, and so is the amortization of the product during the duration of the project. Notice all the software used is open source and therefore it presents a null price.

Product	**Price (€)**	**Units**	**Useful life**	**Amortization (€)**
Hardware				
HP Personal Laptop	1.200	1	5 years	85
DELL PRECISSION T3610	1000	1	5 years	71
DELL LCD monitor	200	1	6 years	12
Software				
Python	0	1	–	0
Conda	0	1	–	0
CentOS	0	1	–	0
Mercurial	0	1	–	0
Bitbucket	0	1	–	0
PyCharm Community Edition	0	1	–	0
LaTeX	0	1	–	0
TeXstudio	0	1	–	0
Total	2.400			168

Table 2: *Hardware and software budgets. Useful life estimates from* cnet *[42]*

For the human resources presented in Table 3, three roles needed for the project are used as reference. These three roles will be taken partially in some way by the stakeholders for this project, mainly the developer. The total hours used for this project, which are 720, are divided into these roles.

Role	**Hours**	**€/hour**	**Salary (€)**
Project Manager	100	38,8	3.880
Software Developer	400	20,2	8.080
Software Tester	220	20	4.400
Total	720		16.360 €

Table 3: *Human resources budget, salaries from* payscale *[43], specifying the search to the city of Toronto*

The attribution of these budgets to the tasks of the project can be considered proportional the hours estimated for them. This can be assumed because both three roles defined will have a similar weight in all the tasks, because all will need to be managed and directed, developed and tested.

5.1.2 Other budgets

No unexpected budgets must be specified for this project, because its total duration will not change. Due to the possibility of adding more analysis to the pipeline and plots to the visualization, even if the project ends soon it can be continued until the total duration specified for it has been accomplished. If unexpected obstacles are found, as it has been mentioned before, the goals of the project will be achieves all the same and the total duration will not be modified. Other causes for unexpected budget could be the need of new software or hardware, which will be topped at 500 €. Because of the nature of the project, which is software based, no depreciation costs will take place.

However, we can specify some indirect costs to the project, which can be found in Table 4.

Product	**Price (€)**	**Units**	**Cost (€)**
Electricity	0,12 /kWh	550 kWh	66
Internet	40 /month	4,25 months	170
Total			236

Table 4: *Indirect costs*

No benefits will be obtained from these project, because the tool to be developed will be open source software available to anyone.

5.1.3 Total budget and monitoring

Taking into account all previously specified budgets, we can see the final results in Table 5. An extra 5% has been added to the budget to cover any factors that may not have been considered.

Concept	**Estimated Cost (€)**
Hardware and Software	168
Human resources	16.360
Unexpected costs	500
Indirect costs	236
Subtotal	17.264 €
Contingency (5%)	863 €
Total	**18.127**

Table 5: *Total project costs*

To make sure the final budget is kept within this estimated one the direct and indirect budgets must be accomplished, as expected, and the unexpected costs must be kept within the given limit. While the 5% of contingency has been added to the budget, it is important to keep in mind that this amount is only expected to be needed as an emergency, so ideally all the budgets must keep within their original values. To keep track of all this, all possible extra costs will need to be documented and kept track of, so that they can be added to the final budget in the end.

5.2 Sustainability

In this section we will take a look at the sustainability effect of this project, based on the sustainability matrix shown in Table 6 and taking into account the online survey found in `goo.gl/kWLMLE`. The results obtained show that the project takes into account many of the sustainability issues it can have an effect on.

	PPP	**Useful life**	**Risks**
Environmental	Design consumption	Ecological footprint	Environmental risks
	9/10	19/20	-1/-20
Economical	Bill	Viability plan	Economical risks
	9/10	19/20	-2/-20
Social	Personal impact	Social Impact	Social risks
	9/10	16/20	0/-20
Sustainability range	27/30	54/60	-3/-60
	78/90		

Table 6: *Sustainability matrix for the project*

5.2.1 Environmental

This project is based on the development of a tool so, at first, it would seem that no environmental effect will take place. However, there is one form of environmental effect that is present, which is the electricity consumption needed. During the development phases of the project the electricity consumption will be moderate, as not much demanding software will be run. Once updated to the internet, the installation and use of the tool should also demand a very low amount of resources. This is the reason why the consumption is considered to be low, as well as the ecological footprint. No major environmental risks are presented.

5.2.2 Economical

The specifics regarding the economical effect can be found in subsection 5.1. While some resources will be used, the estimated budget is not very high. Furthermore, the tool will allow in the future to reduce costs in segmentation analysis, both in time and resource consumption. For these reasons the economical section of Table 6 is also stated as very positive. The plan suggested in this project ensures the viability of it, and the design and use of tools makes it so the useful life of the project is very high.

5.2.3 Social

As it has already been mentioned before, this project has the aim to have an important social impact in the bioinformatics community. The goal of the project is to make the scientist's life easier by automating a lot of the work that needs to be done manually right now. This aspect of the project makes it so it has a positive impact, allowing the user to spend less time in running analysis and concluding results much faster. The tool will be available online for anyone to use, so it has a lot of reach too. No risks are presented whatsoever by this project on the social category.

Finally, from a personal perspective, this project will help me learn a lot of things about the process of the development of a tool. This will include a variety of skills to learn and have for the project to succeed. It is also an enrichment by the biological aspect of it, which will allow me to understand the work of a bioinformatic and learn a lot from it.

6 Applicable regulation

In this section all regulations applicable to the project's environment will be studied. These can be separated into those related to the policies stated by the organization where the development will take place, and those related to the distribution of the software developed.

6.1 UHN Policies

Since this project will be developed under the University Health Network (UHN), all their policies may apply to it. A variety of policies can be found in the following link [44]: `http://www.uhn.ca/corporate/AboutUHN/Governance_Leadership/Policies`. Because of the general focus for UHN being the research using patient data, most of the policies revolve around how to manage this sort of information. Other policies include the ruling on communicating confidential information to the police or to the media, as well as more general guidelines on how to use technological devices and resources in UHN or how to ensure the safety and healthiness of all the employees. While all these policies will be applicable during the stay in the *Hoffman Lab*, those won't be fully explained in this section because they don't directly apply to the result of this project, which is the tool we want to develop.

On the other hand, in the *University Health Network Policy & Procedure Manual* [45] and more specifically in the *Administrative – Intellectual Property Protection & Commercialization* the Intellectual Property (IP) Policy is defined. This project will be developing software, which is included in the intellectual property definition and, consequently, the specified policy should apply to this project. It is said that it applies to all UHN personnel and to all IP developed by them, including research works and institutional works. Furthermore, it is specified that even if the IP is developed by individuals affiliated to other medical or educational organizations it will still be governed by the policy, which is the case of this project with Universitat Politècnica de Catalunya (UPC). As a result, the following policy can and will be applied to the IP developed during this project.

Getting to the content of the mentioned policy, it states that "UHN solely owns all IP developed by UHN personnel, and any IP otherwise arising through the use of UHN resources, with the exception of traditional academic works, which remain the property of the author(s)" [45]. Following this statement, the definition used for the mentioned *traditional academic works* is the following: "Any work created for a scholarly purpose, including scholarly papers, books, book chapters, abstracts, presentations, whether or not published and whether or not distributed by any means, including print or electronic media" [45]. As a consequence, all software developed will be property of UHN while all written documents and presentations and such will remain my property. However, this doesn't interfere with the fact that the tool will be made available as open source software.

6.2 GNU General Public License

In order to distribute open source software a license to which the distribution is compliant is needed. For this project the license that will be used is the GNU General Public License version 2 (GPLv2) [46]. This license, used by many free software like Segway or Segtools, guarantees the freedom of the user to share and change free software.

In the context of free software, the word free is referring to freedom and not price. GPLv2 makes sure that the licensee has the freedom to distribute copies of the free software and charge for it if he wishes, that the software is accessible and that it can be changed or incorporated even partially into other free software projects. All the conditions specified apply not only to the original creator of the free software, but to all those that have any kind of contact with it, either by using it or by doing any of the actions specified as legal by the license. Of course, in between all the freedom it allows, some conditions must apply to protect these rights for other users. These conditions include that any changes done to the original software must be documented in some way stating the date of the changes, and that any incorporation or change must maintain this same license so that the original free software never loses all these rights.

As stated by its name, this license is thought for software open to the public without any specific restrictions, which is what we are looking for in this project and so the appliance of this license is appropriate. The full license can be found in the following link: `https://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html`.

7 Design and development

During the course of this project the initial planning defined in section 4 has been followed as much as possible. However, because of the methodology used and obstacles being found, this planning has seen alterations. We will now see the full development of the project, stage by stage, with all the changes to the original planning explained. Thanks to the *Lab Notebook* that was kept updated at all times, all the exact dates in which stages were completed can be compared to the original planning.

7.1 Initial stage

During the initial stage things like the objectives, the methodology and the scope must be defined. It is its finality to get everything ready for the development of the project to begin, having a fixed schedule to follow if possible. This whole initial stage was completed on schedule without major hold-backs before moving to the first development stage next.

7.1.1 Getting started

The start date for this project was the 14th of February of 2018. During this starting stage the necessary steps where taken to begin the internship in the University Health Network (UHN). From creating an account to setting up the computer where I would be working, all these preliminary tasks where taken care of during this stage. This includes preparing the space where the *Lab Notebook* is created, necessary for the methodology described in subsection 3.2, and also preparing the work space directory following the guidelines in *A quick guide to organizing computational biology projects* [47].

During and after setting everything up it was necessary to get familiar with some of the concepts I would be working with during the project. This includes gathering information, reading papers and trying out some software like Segway and Segtools. All this is of high importance so that it is possible to better define the objectives of the project as well as execute its development correctly. In addition all the state-of-the-art was researched, summarized in section 2. As mentioned before, this step is specially important for the likes of this project because of having the objective to create a tool that can be useful for the researchers.

7.1.2 Resources used

Once all the objectives have been well defined, it is necessary to specify which resources are going to be taken advantage of. While many of the options and decisions were mentioned in subsection 3.2, we will now see some more details on these resources.

Python

The first thing to decide when beginning to develop any kind of software is the programming language(s) that is going to be primarily used. In this case, Python [48] is the one chosen. While this decision was taken from the very beginning, it is because of very good reasons that this is the language chosen for most bioinformatics applications.

Python is a language that allows for quick programming and effective integration of systems. It is a programming language that is very easy to both write and read, and has a lot of different applications. It counts with a big variety of libraries that allow for its versatility in many programming areas. Furthermore, it has a packaging utility that makes it very easy to distribute Python tools, as we will see. Specially in the bioinformatics community, Python is a very popular language because of all the characteristics mentioned. Although it is not the best for optimization reasons, most of the code can be written in Python. The big amount of support for it means that there is a lot of already developed Python tools and libraries open source. Because of this and the language's nature, big amounts of code can be reused which, combined with the generally excellent documentations for Python modules online, makes it very easy to develop a new tool using this language.

As mentioned, a big reason why Python is the language chosen for this project is its popularity in the bioinformatics community. Not only this, but both Segtools and Segway are also primarily Python software. As we have the intention to use Segtools for some analysis it may prove useful to use the same language, specially since Segtools has a Python API as an addition to the command line tool. Most of the other resources that will be used during the development of this project will be related to Python so, logically, this was the first decision needed to be made.

Snakemake

In subsection 3.2 we saw that Snakemake [26] is the option that proved to be best fitted for this project's pipeline building. As stated in the documentation, Snakemake's workflow management system is a tool to create reproducible and scalable data analyses. These characteristics are specially important in bioinformatics software, which is what this tool is primarily used for.

We saw that, being a Python tool, the language it uses to write workflows is very similar to that of Python. This is illustrated in Figure 9. This language is based on *rules*, which specify how to obtain certain output files from certain input files. In Figure 9 we can see how they are specified. First, the keyword "rule" is used followed by its name, and then the inputs and outputs can be specified by their file names separately. To specify how to obtain the outputs there are many keywords available. The most notable two of them are *shell* and *run*, which are followed respectively by command line calls or Python code.

Next we will see the concept of variables in Snakemake. First of all, many of the values specified under keywords like *input* or *output* are considered like variables that are accessible by calling the structure *{keyword}*, like it is used in the *shell* call

```
rule targets:
  input:
    "plots/dataset1.pdf",
    "plots/dataset2.pdf"

rule plot:
  input:
    "raw/{dataset}.csv"
  output:
    "plots/{dataset}.pdf"
  shell:
    "somecommand{input}{output}"
```

Figure 9: *Example of a workflow using Snakemake, from its documentation [26]*

in the rule ”plot” in Figure 9. Calling the structure is equivalent to adding all its contents, but just one element can be chosen by assigning names to them or using a list-like indexation. Aside from these, there is one concept called *wildcards*. These are variables that don’t have a specific value, so they act as placeholders. They don’t need to be initiated, and they are called by using the similar notation *{wildcard}*. These are useful to make a rule be able to run on many different input files that follow a pattern in their path.

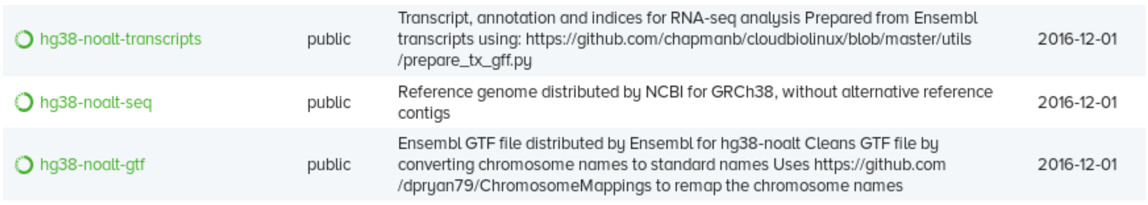
To execute a workflow, which is stored in a file called Snakefile, a rule can be specified or, if not, the rule that is found on top of the file will be the one targeted. Consequently, all the rules needed to obtain the inputs for the target to be executed will also be run. Going back to the workflow shown in Figure 9, if the workflow is executed the rule ”target” will try to be run, and as a consequence the two input files will be needed. To obtain them, the rule ”plot” can be run two times, with *{dataset}* = dataset1 and *{dataset}* = dataset2 respectively. Snakemake knows this by comparing the needed input files to all the outputs obtainable and seeing if, for some value of the *wildcards*, they match. In this case, if ”raw/dataset1.csv” and ”raw/dataset2.csv” are found, then they will both be used to obtain the needed intermediate files, and the rule ”targets” will be completed, as no action or outputs are specified. If these two files are not found, the pipeline will exit with an error, as there is no rule currently specifying how to obtain them.

As mentioned before, this ”rule” management of the workflow makes it easily scalable, with the use of *wildcards* specially. This makes it so an iterable approach to the development of the tool is the perfect solution, which is what was decided in subsection 3.2. A first version of a workflow that obtains some results can be created because the future addition of more analysis is facilitated by Snakemake.

Finally, there are many more keywords that can be specified in a rule. Some examples of them are *params* which allows to declare variables accessible from the *shell* or *run* scopes, *threads* which specifies the maximum number of cores to be used for the execution of the rule, or *script* which allows for the execution of a separate script with Python code instead of having it directly in the Snakefile using the *run* keyword. Some of these may be needed for the development of the tool.

Go Get Data

As also stated in subsection 3.2, the tool that was selected to take care of downloading the necessary data to run the analysis wanted was Go Get Data (GGD) [29]. It consists on a platform that houses conda [31] recipes for genomic data. The whole list of packages it contains can be found either on their GitHub or in their Anaconda [30] channel, *ggd-alpha* [49]. The fact that all these packages are stored in the Anaconda cloud means that the data can be downloaded by running the command `conda install -c ggd-alpha package_name` on the Terminal. The "package_name" is the identifier for the recipe, the leftmost section seen in Figure 10. What this will do is run the recipe specified in that package which, in this case, will download a genomic data file in your computer and sometimes process it.



hg38-noalt-transcripts	public	Transcript, annotation and indices for RNA-seq analysis Prepared from Ensembl transcripts using: https://github.com/chapmanb/cloudbiolinux/blob/master/utils /prepare_tx_gff.py	2016-12-01
hg38-noalt-seq	public	Reference genome distributed by NCBI for GRCh38, without alternative reference contigs	2016-12-01
hg38-noalt-gtf	public	Ensembl GTF file distributed by Ensembl for hg38-noalt Cleans GTF file by converting chromosome names to standard names Uses https://github.com /dpryan79/ChromosomeMappings to remap the chromosome names	2016-12-01

Figure 10: *Snippet of packages in the ggd-alpha channel hosted in Anaconda [49]*

For this project in particular it will be interesting to know where exactly the files are downloaded. It is necessary to know it beforehand because of how Snakemake workflows work. For this kind of things, GGD has a convention for stating where the files will be downloaded, which is in the path `$PREFIX/$species/$build/$recipe/`. The `$PREFIX` refers to the path to the anaconda environment that is used when calling the command mentioned above. If no specific environment has been created, then the path is the one where anaconda has been installed. Additionally, this `$PREFIX` variable includes the path `/share/ggd/`. The rest of the variables are pretty self-explanatory. `$species` is the species for which the data comes from, for example "Homo_sapiens". `$build` is referring to the version of the genome for which the data has been computed. In the case of the human genome, the two most known builds are "hg38" and "hg19", where "hg" stands for Human Genome. Finally, `$recipe` is the specific name of the recipe used. In the case of the recipes we see in Figure 10, their build is "hg38-noalt" and the recipe names are those that follow in the package names: "transcripts", "seq" and "gtf".

The most important benefit from using this tool is the fact that it is completely open source, which means that anyone can add new recipes which download new data in case they don't yet exist. This can be done by creating a pull-request in their GitHub repository. Once some tests are automatically run to ensure the quality and fidelity to the information specified for the recipe, the new recipe is added and automatically uploaded to the Anaconda cloud, making it available for everyone to download. This tool can be very helpful because usually it is very difficult to find reliable genomic data on the internet and, this way, it becomes very easy. We can use GGD without worrying so much about the data we want being there or not, because of the fact that we can add it easily to the database.

Although it presents a lot of benefits, there are downsides to this tool. The most important one is that, as seen in Figure 10, most of the recipes are very old and, therefore, they usually get old versions of the data. Furthermore, the support for it had been dropped a bit before this project's start and, even if it has now been retaken, some of the conventions are not always followed by all the recipes. These issues, however, should prove easy to solve by keeping in contact with the developers in charge of it.

PyPi / Bioconda

Revolving the third stage of this project, making the tool available, both PyPi [40] and Bioconda [41] are the options decided as objective. They are not incompatible options as they serve different purposes and usually one depends on the other.

PyPi is a distributing system for Python packages that is widely popular and, usually, is the place to go to find specific packages. More than an option, it is almost mandatory to have the tool developed uploaded to this system for it to be easily distributed. Once a package is on PyPi it becomes accessible via the command `pip install package_name` to be installed automatically. While this is the more popular and logical option to use, there is another alternative to this for a Python software to be distributed. This alternative consists on having the software in a public repository hosted by either Bitbucket or GitHub and have the software sorted in a way that can be easily executed. The instructions given to the user would be to clone the repository to their local computer and run one or two commands to execute the program directly from there. However, this option treats poorly one of the main issues with bioinformatics software, which is the big amount of dependencies. However, PyPi not only makes it easier to install the package but it takes care of many dependencies for the user and also allows for different versions of the software to be uploaded and installed, chosen by the user.

Bioconda is an Anaconda channel and so it works very similarly to the "ggd-alpha" channel that was just described. It contains all sorts of bioinformatics recipes which, in this case, install software instead of downloading data. Even if PyPi already does a good job at taking care of all the dependencies, some software is not available there and not always are the dependencies well defined. However, by using conda all possible dependency issues are taken care of, always. This is ensured by applying automated tests to all new recipes which simulate the installation of the package in an empty environment and tests if it works. An empty environment is a simulation of a system without any prior installations. As a result, the tests fail if some dependency is left up or if there are any errors in the package. Additionally, for a new conda recipe to be accepted there are a set of conditions and guidelines that must be followed, like the inclusion of some information about the project. Finally, the way to install a package that is on anaconda is by using the command `conda install -c bioconda package_name` if needed to specify the channel.

As a conclusion, bioconda works as a better version of PyPi specially for the users who want to use the tool. This comes with the cost of having to get through the trouble of adding the recipe to the *bioconda-recipes* GitHub [50] in a similar way to how GGD does it, and following the guidelines and respecting the stated conditions. Because of all this and that the conda upload can take advantage of a PyPi package already having been created, the first priority for this project will be to create the PyPi package and then move on to the Bioconda release, as was mentioned in the planning in section 4.

7.2 First development stage

Once all necessary decisions have been taken, the first development stage can be started. Like it has been mentioned both in the methodology and the planning sections, the objective will be to first have a functional version of a pipeline and then continuously add more things onto it. Furthermore, we will have an iterative approach to the development, which means that even when creating the first version of the pipeline, this will be done in several stages with proper testing for each one. The first step will be to get started with using the tools specified for the development, Snakemake and GGD.

In terms of the planning for the project, this stage was developed in schedule, as we will see, but in a different manner than planned. The fact that agile methodologies are being used means that the objectives are constantly being changed. This stage was initially thought out to be developed one task after the other. However, after implementing some analysis it was thought best to process them before adding more analysis, so that the procedure is more iterative and easy to test. This is why the four main tasks from this stage were at times developed at the same time and not sequentially. The budget estimated was only changed by the fact that it depends on the time spent, so the final budget needed can be considered to be much lower. As we will see, however, this time will be invested into other stages of the project so it doesn't have an effect on the final budget.

7.2.1 Testing tools

To get started with Snakemake the tutorial available in their documentation [26] was used. This tutorial guides the user through the Snakemake installation, in this case in a conda environment with Python 3, and the development of a simple pipeline which contains 6 simple rules very similar to the ones seen in Figure 9. Most of the characteristics and utilities explained from Snakemake like the main keywords and the *wildcards* are introduced here. All the data used for the tutorial can be easily downloaded following its instructions, so it is not necessary for the user to have anything in advance. After testing that the results from the tutorial were the ones expected, the true development of the tool can be started.

For the first version it was of interest to already download some data from GGD to test both tools. Two specific data files were chosen to be downloaded. The first one is the full human genome sequence, which basically specifies one nucleotide for each position of the genome, and can be used to obtain some data like the GC content, inspired by Figure 2. The GC content is basically the percentage of G and C contained by all the segments annotated with the same label, compared to the total number of nucleotides contained. The second data file chosen to download is the annotation containing information about genes, exons and transcripts in the human genome. This information is necessary to run analysis that compare the obtained results to other annotations like Segtools' *aggregation* or *feature distance*.

GGD hosts genomic data from different species and different builds. The segmentation used for testing during the development of the tool is the one obtained after completing Segway's protocol, which was used as a tutorial during the initial stage. This segmentation corresponds to a human genome (species = Homo sapiens), and specifically to the build *hg38*, and is stored in the usual format called Browser Extensible Data (BED) [51]. As a consequence, to start testing this pipeline it interests us to obtain all this data for the same given species and build. Luckily, GGD already contains both files we want that met these specifications. The specific packages are called *hg38-gtf* and *hg38-sequence*. The name of the annotation refers to the file format used to store the data, the General Transfer Format (GTF) format [52] which is similar to BED. Understanding how this format works will be of high importance for this project, as will be seen later. The sequence is stored in a format called FASTA [53], which is format specifically used to store genomic sequence data.

A first pipeline with only the two downloads being executed was created using one rule for each download and a general rule which specifies the targets needed. This rule is generally called "rule all", and typically only presents input files. If the rule is found on top of the file, when calling Snakemake through the command line by just calling `snakemake` this rule will be executed and therefore all the rules necessary to obtain said inputs will be run. After testing this pipeline using the command line, it was noticed that the path were one of the files was downloaded didn't exactly match the specified convention seen before. As this is just a first version, for now the different name is hardcoded into the code, as well as the species and build used, although all of them stored into variables present on top of the Snakefile to facilitate future changes to them.

7.2.2 First analyses

Once the tools have been tested the next step is to add analyses to the pipeline. Starting by those analyses that take advantage of the data already being downloaded by the workflow, the most important information to obtain from a segmentation will be created. As we have seen, this will be done by executing existing software like Segtools' different tools and afterwards processing the outputs. From this point onwards the development of the tool will be done using a Bitbucket repository where the code will be stored for anyone to access using the url `https://bitbucket.org/hoffmanlab/segzoo` [54].

Aggregation

As described in the initial planning, the first analysis we want to incorporate into the pipeline is Segtools' aggregation because it needs the GTF file previously downloaded and it is an analysis very commonly done in bioinformatics that should be in the final visualization in some way. As a result, a new rule was created that needed both the segmentation and the GTF file as inputs and had a single output file. All Segtools commands have an argument when using the command line called ”–outdir” which lets the user define the output directory for the results. The final files' names, however, cannot be modified so these must be directly put into the code when specifying the output paths. Other arguments must be taken into account when doing the call to *segtools-aggregation* using the keyword ”shell”. It must be specified that the analysis consists on an aggregation to a gene annotation so that the components that are used for the aggregation are those of an idealized gene. A series of other arguments can be added to make the execution faster: ”–noplot” to skip all the creation of visualizations, as we are only interested in the numerical results and, furthermore, the option ”–quick” allows to run the aggregation on only one chromosome instead of all of them, making the analysis a lot faster and the results erroneous, so this will only be useful for testing the pipeline.

```
#ANALYZING RULES

#Segtools Aggregation parameters obtention, general and by gene_biotype
rule run_segtools_aggregation_general:
    input:
        preprocessed_segmentation,
        gtf=os.path.join(BUILDPATH, recipeGTF, fileGTF)
    output:
        os.path.join(DATA, ”aggregation”, ”general”, ”feature_aggregation.tab”)
    shell:
        ”segtools−aggregation −−mode=gene −−quick −−noplot −−clobber −−outdir {DATA}/aggregation/general
              {preprocessed_segmentation} {input.gtf}”

rule run_segtools_aggregation_gene_biotype:
    input:
        preprocessed_segmentation,
        gtf=os.path.join(DATA, ”gene_biotype_gtfs”, ”general”, ”{biotype}”, fileGTF)
    output:
        os.path.join(DATA, ”aggregation”, ”gene_biotype”, ”{biotype}”, ”feature_aggregation.tab”)
    shell:
        ”segtools−aggregation −−mode=gene −−quick −−noplot −−clobber −−outdir
              {DATA}/aggregation/gene_biotype/{wildcards.biotype} {preprocessed_segmentation} {input.gtf}”
```

Figure 11: *Snippet of the initial workflow developed, showing the rules running the aggregation*

The analysis mentioned is going to be called ”general aggregation” from now on, which is the aggregation on all genes found in the gene annotation from the GTF file. This is because an extra rule was added to the pipeline a bit later which executes the aggregation on only a fraction of the genes. This will be explained in more detail later on, but a partition of the GTF file will be done depending on each gene's *biotype* [55], a property specified on the annotation that tells more about the gene's function. Using each of the GTF files resulting from this partition, a new aggregation analysis will be run using Snakemake's *wildcards* so that only one extra rule must be specified. The two rules, which can be compared in Figure 11, only differ on the paths assigned to the inputs and the outputs and the fact that one uses a *wildcard* for it and the other doesn't. Although these extra analyses are run using different gene annotation the results still share the same format than the general aggregation, so they can all be processed afterwards using the same script, but will need a new rule for it anyways, similar to the one in Figure 11.

Nucleotides

Although this was not in the original planning, it was decided to add another analysis to the first version of the pipeline. This analysis is the one which is going to use the other downloaded file, the genome sequence, and is going to get information about the nucleotides found in each label's segments. From these results other values can be obtained, like the GC content we already talked about.

To run this analysis there is more than one viable option of software to use. Segtools has a tool that can obtain these results from a segmentation, but it needs a genomedata [7] file for the sequence. Although it would be possible to convert other files like the FASTA we can obtain from GGD to the genomedata format, this would be expensive both in time and resources. For this reason another option was chosen instead: BEDTools [15]. This software, that was mentioned in section 2, also provides some tools similar to those from Segtools. One of its downsides is the fact that no plots are created from the results but, as mentioned, this is not something we need when running the analysis in the pipeline. More specifically, the software needed is the tool ”bedtools nuc”.

No more arguments can be provided to the tool other than the change of the directory where the resulting file will be created. This file contains for each segment in the BED file some of its information, including the label assigned, and a series of other values, including the number of each different nucleotide found in the segment. This information also includes the percent of GC and of AT. However, even if right now we are only interested in the GC content to add it to the final visualization, because it is an easy metric to recalculate we are more interested in the number of each nucleotides found as it is more generic information. The inclusion of this analysis was tested together with the aggregation.

Length distribution

Once the pipeline was more developed, with some analyses' results being already processed, other interesting analyses were added to it too. The first one was the length distribution. Similarly to the aggregation, Segtools provides a tool to obtain a set of metrics related to the length distribution of the segments and it was chosen to be used in the workflow. The implementation into the pipeline for this analysis resembles the others, adding a new rule that in this case only needs the segmentation as an input and outputs the results in the desired directory. Furthermore, the arguments added to the call are almost the same than in the case of the aggregation, although in this case there is no ”–quick” option.

label	num.segs	mean.len	median.len	stdev.len	num.bp	frac.bp
all	14424537	202.052	140.000	1369.058	2914508857	1.000
0	59801	414.964	380.000	270.765	24815249	0.009
1	60798	961.382	400.000	2160.199	58450099	0.020
2	255948	415.311	240.000	469.170	106298140	0.036
3	2467217	152.959	130.000	96.371	377383130	0.129
4	2728482	179.740	140.000	134.553	490417019	0.168
5	1985340	147.355	120.000	107.925	292549347	0.100
6	1771985	150.698	130.000	90.095	267034694	0.092
7	1085564	214.758	150.000	231.079	233133298	0.080
8	2429674	340.998	180.000	3295.653	828513796	0.284
9	1579728	149.338	120.000	99.608	235914085	0.081

Table 7: *Results obtained running segtools-length-distribution on the testing segmentation*

The results obtained from the analysis on the segmentation we are using for testing are the ones found in Table 7. A file called ”length_distribution.tab” is created which specifies the length for each segment analyzed. Another file is created with the name ”segment_sizes.tab” which is a tab-delimited file with the information on the table. As we can see, most of the columns are metrics than can already be very useful for the analysis of a segmentation: the number of segments for each label, the mean, median and standard deviation values for the segment lengths and the number of base pairs (nucleotide) that each label represents, both raw and in fraction. In addition to this, the representation for the results is clear and very easily manageable. For all these reasons, this table will be left like this without any post-processing, which is the reason why we are seeing it now and not in a later section. Furthermore, this format will be taken as the convention used to store all results. This way, they will al require the minimum amount of changes to be used for the visualization.

Feature distance

This analysis, also explained in section 2 together with its tool within Segtools, shares a lot of similarities to the aggregation analysis. They both need an external annotation to compare the segmentation to, but in different ways. While the aggregation gets the count of coincidences between a segment's label and a component in an annotation, feature distance finds out exactly that, the distance for each segment to the closest feature. In the case of the gene annotation in the GTF file we have, this analysis will output the distance to the nearest gene for each segment, given that the file has been filtered so that only the genes stay, taking out other present annotations like exons. This information can be of use if trying to understand the likeliness of a label being found in a gene.

The implementation of this analysis was done following the example of the aggregation analysis. Not only do these analyses share the GTF file as an input and similar arguments, but they both are interesting to be executed not only on a general analysis but also by gene biotype. As mentioned while explaining the aggregation analysis, the GTF file will be divided into smaller ones that only contain the genes of one specific biotype. This means that an extra rule will need to be created which uses *wildcards* to specify the biotype for which it is being run, like seen in Figure 11. All results are stored in a specific directory similarly to the other analysis.

Learned parameters

Finally, once all the analyses we have been seeing were implemented in the pipeline and the visualization was the next step, one last thing was decided to be added. This last analysis consists on the GMTK parameters explained in section 2, which identify what each label found by the segmentation means in terms of the input tracks used in the Segway run. The Segtools tool was also chosen for this task, to avoid having to write a custom script which parses the file where these parameters are stored, which would basically re-write the code that already exists. Furthermore, Segtools is already a heavy dependency for this pipeline, so there are no possible bad consequences for using it here too.

One single rule more was needed to implement this analysis and, as usual, adding the output file to the ”rule all” on top of the Snakefile so that it must obtained when executing the pipeline. A new input is needed to execute this rule, which is the ”params.params” file obtained by the training of the Segway network that needs to be parsed. The output obtained is a CSV file which contains all the mean values for the learned parameters, but not the variances, which could also be an interesting value to add to the visualization. This CSV is very similar to the results of the length distribution, seen in Table 7, presenting the labels in the rows and the names for all the tracks as the columns. As mentioned, this format is the one chosen as convention to store the results to be used directly by the visualization and, because of the minimal difference between both CSV and tab-delimited formats, this output will also be exempted from any post-processing.

When all wanted analyses for the first functional version of the pipeline had been added, various tests were run using the ”rule all” explained earlier and executing the pipeline in its entirety to check that all dependencies are specified correctly.

7.2.3 Processing results

Once the analyses are run on the pipeline it is sometimes necessary to process them to obtain the results that we want to add to the visualization. All outputs obtained by the different software used tend to be tab-delimited files, which is what we want in the end, but they don't always contain the data that we want yet or they do but in a different format than the one we want. The fact that all of them are tab-delimited and table-like makes it so they can all be processed using the same software, in this case a Python library mentioned before called Pandas [33]. Pandas is an open source library that provides high-performance easy-to-use data structures and data analysis tools for Python. Specifically in the 2D space, which is the one we will be treating with, it uses a data structure called a data frame, which is similar to a matrix with names assigned to each row, column and axis and allows for all sorts of operations to be done efficiently. Furthermore, a new data frame can be created easily by specifying a file with the characteristics we just mentioned.

The first two analyses to get processed were the aggregation and the nucleotide outputs, as they were the ones present in the first version of the pipeline. To do so, separate scripts were created that took the outputs from the analysis as inputs and stored the processed results in a different file. These scripts were written with the help of Jupyter [56] and the kernel IPython [57], which allow a Python working environment where commands can be executed independently or as a script to be able to interactively see the results obtained from them. Specially as we are working with table-shaped data with Pandas, the Jupyter notebook allows to constantly know how the results of a call look like so that it can be corrected instantly if needed. These scripts can later be called from Snakemake with the keyword *script*. By doing so, the scripts developed can access data from the Snakefile like the inputs or outputs of the rule they are in.

```
import pandas as pd
import numpy as np

#creates a data frame with segtools' output, computes the means of the values for each gene area and label,
#and writes it in a tab−delimited file

#[0:8] is the interval of groups to take if we only want to analyze the splicing
#[9:] would be for the translation
#it's been encapsuled in a function called run that can be called from Snakefle by importing this script

df = pd.read_table(snakemake.input[0], skiprows=1, header=0)
means = df.drop(columns=['offset', 'group']).groupby("component", sort=False).mean()[0:8].T
means.to_csv(path_or_buf=snakemake.output[0], sep='\t')
```

Figure 12: *Script that creates the aggregation results from the output from Segtools*

label	5' fl.	ini.ex	ini.in	int.ex	int.in	ter.ex	ter.in	3' fl
0	1.394	0.68	0.04	3.0	3.3	0.64	0.04	0.0
1	148.344	142.76	112.62	385.64	307.84	116.04	82.1	115.972
2	4.582	4.88	6.06	40.4	53.44	1.0	1.3	2.3
3	19.934	23.8	10.26	149.72	100.1	27.48	14.0	27.046
4	14.802	16.12	6.26	23.24	18.92	20.92	7.94	18.15
5	12.458	14.68	6.76	146.08	116.2	25.52	9.42	16.226
6	10.086	10.36	4.82	92.2	83.84	15.28	8.96	21.326
7	1.05	2.68	0.08	0.72	0.52	2.08	0.24	1.468
8	400.132	379.92	256.2	905.88	760.52	382.08	280.24	409.848
9	2.218	1.12	0.9	27.12	29.32	1.96	1.76	2.664

Table 8: *Results after processing the output from segtools-aggregation*

We will begin by seeing the processing of the aggregation results, which was done with the script in Figure 12. The initial data that is read in the first line of the script consists on a series of rows which contain the following information: an identifier of the specific position in the idealized gene to which it refers, composed by the component and a value that specifies the location within the component, and the count of segments for each label that were overlapped with that specific position of all the genes in the annotation. We are interested in having the labels as rows, not as columns, and compacting the information in some way. The probably most simple approach is the one selected as solution, which is to compute the mean value of counts for each component, which basically means turning each little graph that we could see in Figure 5 into a single value. All this is what is being taken care of by the second line of the script. Pandas commands always return the result so that all different changes to the data can be chained up together like this. Finally, the resulting table is stored in the output file defined by the Snakefile's rule using the same tab-delimited format. The table obtained for the general aggregation results can bee seen in Table 8. The component names have been changed to make the table more readable in this document, but the actual names are left unchanged from those specified by segtools-aggregation (Figure 5).

```
import pandas as pd
import numpy as np

#creates a data frame with bedtools' output, adds up the parameters for all segments of the same label,
#and writes it in a tab−delimited file
#two rows are added to the table: GC_content and AT_content
#it's been encapsuled in a function called run that can be called from Snakefle by importing this script

df = pd.read_table(snakemake.input[0], header=0, usecols=(["4_usercol", "12_num_A", "13_num_C", "14_num_G",
      "15_num_T", "18_seq_len"]))
df.columns = ["label", "num_A", "num_C", "num_G", "num_T", "length"]
sums = df.groupby("label").sum()
sums["GC_content"] = sums["num_G"].add(sums["num_C"]).div(sums["length"])
sums["AT_content"] = sums["num_A"].add(sums["num_T"]).div(sums["length"])
sums.to_csv(path_or_buf=snakemake.output[0], sep='\t')
```

Figure 13: *Script that creates the nucleotide results from the output from BEDTools*

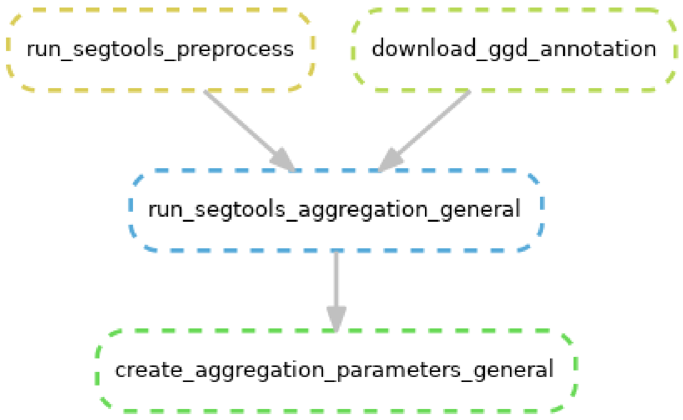
label	num_A	num_C	num_G	num_T	length	GC	AT
0	6545354	5848750	5859093	6562052	24815249	0.47	0.53
1	15240520	13898513	14001424	15308941	58450099	0.48	0.52
2	29904528	23169957	23238596	29985049	106298140	0.44	0.56
3	114543877	74023064	74069662	114746343	377383130	0.39	0.61
4	143764885	101351280	101377881	143922958	490417019	0.41	0.59
5	86476384	59650033	59716273	86706641	292549347	0.41	0.59
6	82170169	51243241	51303897	82317328	267034694	0.38	0.62
7	66229064	50335029	50304158	66265031	233133298	0.43	0.57
8	245658975	167188067	168698227	246949748	828513796	0.41	0.59
9	70008840	47875801	47900868	70128475	235914085	0.41	0.59

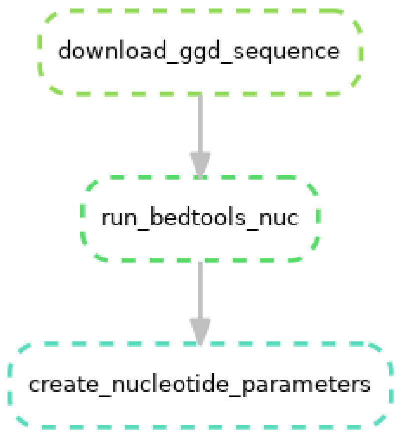
Table 9: *Results after processing the output from "bedtools nuc"*

Next is the nucleotide processing. As mentioned before, the nucleotide output is a list of segments with the nucleotides counts for each of them. The first line of the script in Figure 13 reads the data we are interested in from the original output, which afterwards has the names changed. Even though the intention is for only the

GC content to be used in the final visualization, all the nucleotide values will be stored too in the processed table for possible future interest. This said, the results for all segments from the same label are added up by grouping them up, and then the new columns ”GC content” and ”AT content” are created using the other values from each row. Again, the results are stored in the file stated by the Snakefile rule from which this script is called. The results for our segmentation are the ones visible in Table 9. Pandas once again makes all these different operations really easy to program and efficient.

As it may be noticed, all tables created resemble Table 7, which was taken as model for the convention to use. Because of this, both length distribution and GMTK parameters’ results will be left without further processing. The only analysis remaining to process, therefore, is the feature distance. However, due to not having any clear way to summarize its results into a table similar to these obtained, it was deemed more important to move on to other stages of the pipeline for this first functional version, and leave the feature distance analysis un-used for now. Therefore, the resulting files from the feature distance analysis were taken off from the target rule of the Snakefile from now on, as well as changing the unprocessed outputs to the processed results as new targets.


run_segtools_preprocess
download_ggd_annotation
run_segtools_aggregation_general
create_aggregation_parameters_general
(a) *Diagram of rules for the aggregation results*



(b) *Diagram of rules for the nucleotide results*

Figure 14: *Diagrams of the rule dependencies for the results processed*

The resulting pipeline from these analyses being run and processed already presents some interesting chain of dependencies. In Figure 14 we can observe both independent rule graphs for the aggregation and nucleotide analysis. These graphs show the rule dependencies that are created automatically because of the inputs and outputs specified. In both cases, the graph has a depth of 3, starting with the downloading of the necessary data, followed by the execution of the analysis and finally processing the results. The other initial rule dependency presented in Figure 14a will be explained in the following section.

7.2.4 Preprocessing rules

For all analysis to run there are a set of input files needed. In this case, it is of interest to preprocess these files before running the analyses on them. As we will see in this section, both the GTF file and the segmentation can be processed in some way that interests us.

GTF division

Although doing general analyses like aggregation and feature distance can provide interesting information about the segmentation studied, it is generally better to be more precise on the analysis. This can be done by using annotation files that contain more specific data. In this case, both the aggregation and feature distance are interesting to be run on genes separated by gene biotype. These gene biotypes identify the function that gene has, and the most common are the protein coding genes and the ones that generate lincRNA. Because of this, added to the general analysis run for both aggregation and feature distance, it is of interest to add to the pipeline the analysis per gene biotype. To do this it is necessary to divide the GTF gene annotation file into single gene biotype GTF files. Furthermore, we need to take into account another important factor. The aggregation needs as an input a GTF that specifies at least genes and exons, so that the differentiation between the intron and exon components can be made. However, feature distance needs only the genes so we will need to create two different files for each gene biotype, one with all annotations and the other one with just genes. This is done with a separate script, called from the Snakefile in the same way the processing scripts were called.

The script developed uses pybedtools [58], which is a Python library that wraps and extends BEDTools and offers feature-level manipulations from within Python. It proves useful when having to manipulate feature data like the one in our GTF file. The script runs through the GTF lines, called intervals, using one of pybedtools' classes to do so efficiently and stores the intervals into dynamic memory in form of two dictionaries. These are stored in a list under the dictionary's key according to their gene biotype, always in one dictionary and the second one only if it consists on a gene. Finally the files are created from the dictionaries' data using the class ”BedTool” provided by pybedtools.

This operation needs to be added to the workflow by creating a new rule that has the GTF file as input. However, in this case we have an unknown amount of output files, as the amount of gene biotypes and their names cannot be known beforehand. This is handled by using the ”dynamic files” functionality in Snakemake, which allows to specify a pattern of a path that an unknown amount of files can match. Similarly, the *dynamic* keyword can be used when specifying the targets of the pipeline, so that an analysis is run for every file matching the pattern specified. As an example, `dynamic(data/gene_biotype_gtfs/general/{biotype}/hg38.gtf)` is the call used to specify the outputs from this rule that relate to the GTF files containing all annotations. Finally, new rules were added like explained in the ”First analyses” section to run the analysis using the different GTF files created for both aggregation and feature distance.

Segmentation preprocessing

When testing the pipeline with all analyses implemented, including those by gene biotype, it was noticed that it was slow, specially the feature distance analyses. The main reason for this was that it was run 40 times, which is the number of biotypes found in the GTF file. At this stage, it was considered the possibility of running the analysis only on the most important gene biotypes, which usually are the ones most present across the genes. Because of this, a new file is created during the GTF separation script which informs about the number of annotations and of genes for each biotype, which is stored in a new directory called ”logs”.

Upon closer inspection, however, it was noticed that most of the time spent on the analyses was used to parse the segmentation which is in BED format. A useful tool in Segtools called ”preprocess” parses a segmentation one time and creates an intermediate file that takes seconds instead of 3 to 5 minutes to parse. This processing was added to the pipeline too, and all the inputs referring to the segmentation file were changed to this new preprocessed file, which is in a ”Pickle” format.

Unfortunately, BEDTools does not allow this format as an input, so this optimization can only be done in other analyses than nucleotide. Furthermore, the feature distance analysis was temporarily taken out of the targets as mentioned before and, as a result, the pipeline tests showed that it was executed in its entirety a lot faster. Because of this, in the end it was not necessary to filter the gene biotypes used, but the logged file with information was left for potential future use.

7.2.5 First visualization

After everything we have seen, the state of the pipeline is the one seen in Figure 15, where the feature distance analysis has been commented out for the time being. The only thing missing to have a first functional version of the tool is the visualization. It will be programmed in a separate script as well, to improve the readability and modularization of the code, and added under the *script* keyword in the current ”rule all” which is being used to specify targets.

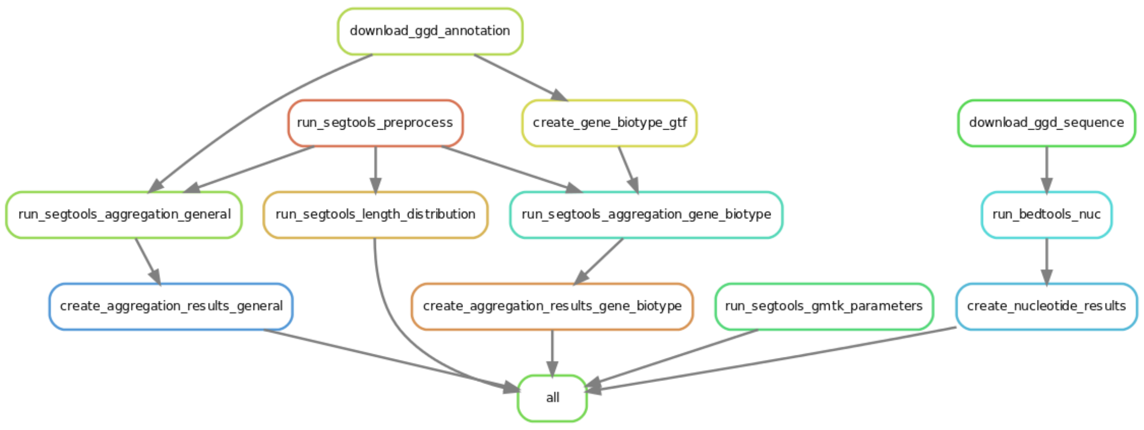

download_ggd_annotation
run_segtools_preprocess
create_gene_biotype_gtf
download_ggd_sequence
run_segtools_aggregation_general
run_segtools_length_distribution
run_segtools_aggregation_gene_biotype
run_bedtools_nuc
create_aggregation_results_general
create_aggregation_results_gene_biotype
run_segtools_gmtk_parameters
create_nucleotide_results
all

Figure 15: *Final rule graph of the pipeline before incorporating the first visualization*

The objective is to present the information in a summarized way, following the idea of Figure 2. A bunch of diagrams could be created, which go in depth for each analysis, but that would not make it any different than the current Segtools html report or other tools. Our main objective is to make the data as compact as possible and easily readable, which is the reason why we are storing the data the way it has been explained in previously, summarizing the original data to single numbers. We want to build a plot which resembles a matrix with all the data stored and that uses colors to make the data more readable and easy to interpret. For this, Seaborn [34] is used, a Python visualization library that provides a high-level interface for drawing attractive statistical graphics. It is based on another Python library called matplotlib [35]. Using both these tools, it is possible to create a type of visualization called heat maps, highly configurable colored matrices that fit our needs.

To develop the script we will be closely working with both visualizations and data tables, which cannot be easily plotted in a normal Python environment. The use once again of the Jupyter notebook will allow constant tests and correction of errors during the development. Specially with the visualization, it is important to continually check what is being obtained with each call to the Seaborn and matplotlib utilities, to keep the debugging simple and quick.

A selection of the data we currently obtain from the workflow needed to be done because, even if not considering the feature distance results, taking into account the 40 different aggregation tables there is a huge amount of data. Because of this, it was decided to divide the visualization in distinguished parts, which will eventually become 3 different heat maps.

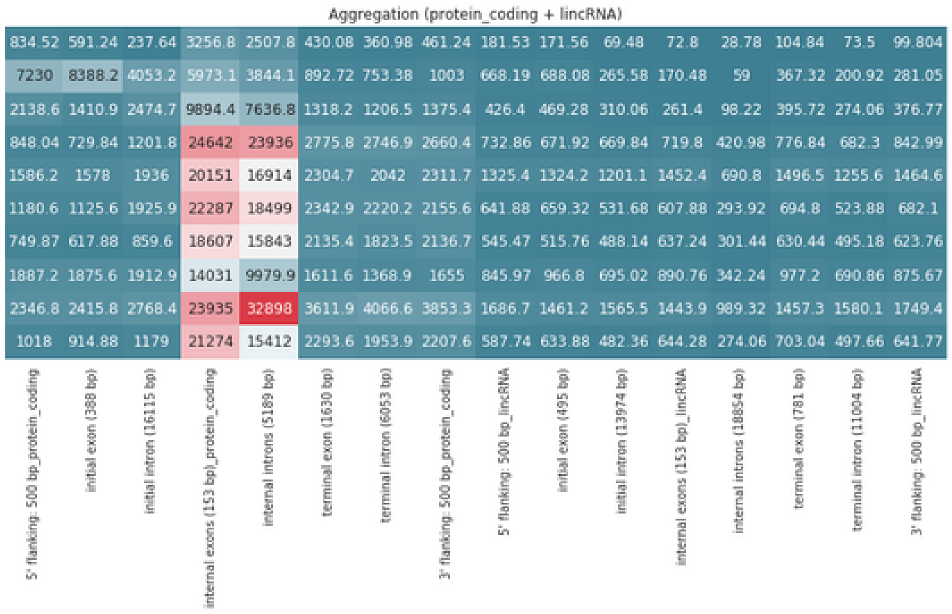
GMTK parameters matrix

Following the convention normally used by the bioinformatics community, the learned parameters should find themselves at the leftmost part of the visualization and differentiated from the rest of it. This distinction is necessary because they represent a different kind of data. While all the rest are results from analyses and information about the segmentation's behavior, the parameters characterize the labels, state their identity. They serve as an introduction to the visualization and must be easily found at all times because they will probably be the most looked-at data. For all these reasons their positioning is on the left, as the visualization will be made in a horizontal manner to increase its readability and take advantage of the format used as convention with the labels being the rows, so that all plots share this axis. Because of its formatting, the use of the data is direct and can be visualized easily with all default parameters from the heatmap function provided by Seaborn.

Aggregation matrix

The aggregation results have the particularity that are very numerous and actually are 3-dimensional if gene biotypes are taken into account. Complex visualizations could be attempted, but this was deemed unnecessary and even counterproductive for this project. The solution suggested is to only visualize both protein coding and lincRNA gene biotype results, as they are the most prominent in the genome and they provide a more accurate and detailed analysis than the general aggregation. As a result, these two data matrix are taken from their respective files and merged together into a single one.

Distinctively from the case with the learned parameters it is now of interest to have in the heat map the actual values of the matrix too. Furthermore, the values are not all between 0 and 1. To take care of this, a copy of the data frame must be created and modified so that the values can be used for the heat map, while the original data frame is used to annotate the heat map cells. If the heat map is created with default options, the results are mostly not readable because two components have very high count values, and so the color map maximum and minimum values are set very apart, making all the smaller values indistinguishable as seen in Figure 16. To solve this problem, the values are re-scaled between 1 and 0, being 1 the higher number in a component and 0 the lower one. This way, the color map limits are set to 1 and 0 and the labels can be compared on each component easily. The original values are still showed in the cells.



5' flanking: 500 bp_protein_coding	initial exon (388 bp)	initial intron (16115 bp)	internal exons (153 bp)_protein_coding	internal introns (5189 bp)	terminal exon (1630 bp)	terminal intron (6053 bp)	3' flanking: 500 bp_protein_coding	5' flanking: 500 bp_lincRNA	initial exon (495 bp)	initial intron (13974 bp)	internal exons (153 bp)_lincRNA	internal introns (18854 bp)	terminal exon (781 bp)	terminal intron (11004 bp)	3' flanking: 500 bp_lincRNA
834.52	591.24	237.64	3256.8	2507.8	430.08	360.98	461.24	181.53	171.56	69.48	72.8	28.78	104.84	73.5	99.804
7230	8388.2	4053.2	5973.1	3844.1	892.72	753.38	1003	668.19	688.08	265.58	170.48	59	367.32	200.92	281.05
2138.6	1410.9	2474.7	9894.4	7636.8	1318.2	1206.5	1375.4	426.4	469.28	310.06	261.4	98.22	395.72	274.06	376.77
848.04	729.84	1201.8	24642	23936	2775.8	2746.9	2660.4	732.86	671.92	669.84	719.8	420.98	776.84	682.3	842.99
1586.2	1578	1936	20151	16914	2304.7	2042	2311.7	1325.4	1324.2	1201.1	1452.4	690.8	1496.5	1255.6	1464.6
1180.6	1125.6	1925.9	22287	18499	2342.9	2220.2	2155.6	641.88	659.32	531.68	607.88	293.92	694.8	523.88	682.1
749.87	617.88	859.6	18607	15843	2135.4	1823.5	2136.7	545.47	515.76	488.14	637.24	301.44	630.44	495.18	623.76
1887.2	1875.6	1912.9	14031	9979.9	1611.6	1368.9	1655	845.97	966.8	695.02	890.76	342.24	977.2	690.86	875.67
2346.8	2415.8	2768.4	23935	32898	3611.9	4066.6	3853.3	1686.7	1461.2	1565.5	1443.9	989.32	1457.3	1580.1	1749.4
1018	914.88	1179	21274	15412	2293.6	1953.9	2207.6	587.74	633.88	482.36	644.28	274.06	703.04	497.66	641.77

Figure 16: *Aggregation heat map obtained without normalization being applied to the data, losing information*

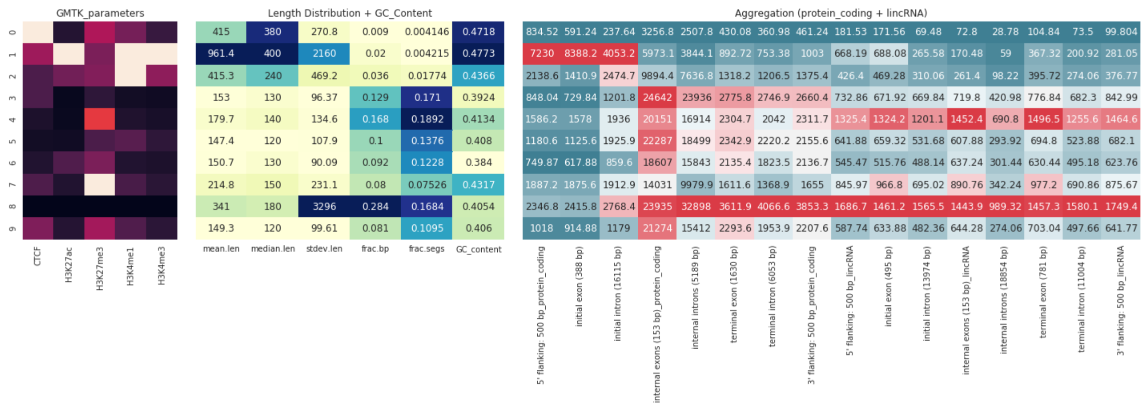
In Figure 5 we saw that the values were normalized to obtain an enrichment. However, the results we obtain are not normalized in any way and only present the counts of aggregations, which lack meaning. An example to show this is that if one label has many more segments than the others, it will naturally have a higher count on the aggregation analysis. Then the heat map created will show it has the highest values, which can be misinterpreted as that label being more in genes than the others, which is not necessarily true. Changes will need to be made in this matrix in particular, but they will be made later on, as it is prioritized to make the tool available beforehand.

Mix matrix

Other than the learned parameters and the aggregation results, all the other data we have obtained with the workflow are basically a set of columns which give independent information on the segmentation. They don't need to be put in any specific way, and can be mixed up. Because of all this, a similar approach to Figure 2 is taken with all these remaining data and it will all be put together in a single heat map, which will be called the "mix matrix" or heat map.

The length distribution results are taken from their respective file and, because they didn't need to be processed before, some small changes are made to them, like adding a new column or taking out the row "all". The nucleotide results are also read, but only the GC content column is actually retrieved. These two data frames are put together in a similar way than the aggregation tables. This resulting data frame is the one that is going to be used as annotation for the heat map.

With putting together a bunch of different data from different sources new problems appear. While the GC content already contains data which ranges from 0 to 1, those from the length distribution analysis are generally not. All possible values must be converted into fractions, which are more readable, and some sort of normalization like the one used in the aggregation results must be made in order to create the heat map. Because all the values are together and will share a color map, the values used to create the heat map cannot be from different scales. Consequentially, all columns are normalized so that the maximum number is 1 and the minimum 0. The GC content could arguably be left without any changes, but for the sake of making the whole matrix homogeneous the normalization is applied too.



Length Distribution + GC_Content

	mean.len	median.len	stdev.len	frac.bp	frac.segs	GC_content
0	415	380	270.8	0.009	0.004146	0.4718
1	961.4	400	2160	0.02	0.004215	0.4773
2	415.3	240	469.2	0.036	0.01774	0.4366
3	153	130	96.37	0.129	0.171	0.3924
4	179.7	140	134.6	0.168	0.1892	0.4134
5	147.4	120	107.9	0.1	0.1376	0.408
6	150.7	130	90.09	0.092	0.1228	0.384
7	214.8	150	231.1	0.08	0.07526	0.4317
8	341	180	3296	0.284	0.1684	0.4054
9	149.3	120	99.61	0.081	0.1095	0.406

Aggregation (protein_coding + lincRNA)

	5' flanking: 500 bp_protein_coding	initial exon (388 bp)	initial intron (16115 bp)	internal exons (153 bp)_protein_coding	internal introns (5189 bp)	terminal exon (1630 bp)	terminal intron (6053 bp)	3' flanking: 500 bp_protein_coding
0	834.52	591.24	237.64	3256.8	2507.8	430.08	360.98	461.24
1	7230	8388.2	4053.2	5973.1	3844.1	892.72	753.38	1003
2	2138.6	1410.9	2474.7	9894.4	7636.8	1318.2	1206.5	1375.4
3	848.04	729.84	1201.8	24642	23936	2775.8	2746.9	2660.4
4	1586.2	1578	1936	20151	16914	2304.7	2042	2311.7
5	1180.6	1125.6	1925.9	22287	18499	2342.9	2220.2	2155.6
6	749.87	617.88	859.6	18607	15843	2135.4	1823.5	2136.7
7	1887.2	1875.6	1912.9	14031	9979.9	1611.6	1368.9	1655
8	2346.8	2415.8	2768.4	23935	32898	3611.9	4066.6	3853.3
9	1018	914.88	1179	21274	15412	2293.6	1953.9	2207.6

	5' flanking: 500 bp_lincRNA	initial exon (495 bp)	initial intron (13974 bp)	internal exons (153 bp)_lincRNA	internal introns (18854 bp)	terminal exon (781 bp)	terminal intron (11004 bp)	3' flanking: 500 bp_lincRNA
0	181.53	171.56	69.48	72.8	28.78	104.84	73.5	99.804
1	668.19	688.08	265.58	170.48	59	367.32	200.92	281.05
2	426.4	469.28	310.06	261.4	98.22	395.72	274.06	376.77
3	732.86	671.92	669.84	719.8	420.98	776.84	682.3	842.99
4	1325.4	1324.2	1201.1	1452.4	690.8	1496.5	1255.6	1464.6
5	641.88	659.32	531.68	607.88	293.92	694.8	523.88	682.1
6	545.47	515.76	488.14	637.24	301.44	630.44	495.18	623.76
7	845.97	966.8	695.02	890.76	342.24	977.2	690.86	875.67
8	1686.7	1461.2	1565.5	1443.9	989.32	1457.3	1580.1	1749.4
9	587.74	633.88	482.36	644.28	274.06	703.04	497.66	641.77

Figure 17: *Final visualization for the first development stage*

The visualization obtained as a result is the one in Figure 17. Note that the color bars showing the color map used for each heat map are not displayed in this initial visualization to make it more compact. The dark blue colored cells in the mix matrix are the highest values for each column and the pale yellow ones the minimum values. The fact that all the heat maps are positioned this way allows for the labels to be omitted in both the mix and the aggregation heat maps, because they share the axis with the GMTK parameters.

7.2.6 File system

During all this section it has been mentioned that files are stored in specific locations. Because of how Snakemake works, it is indispensable to have a designed file system that defines where each file will be downloaded, created or moved. The file system used for this version of the pipeline is showed in Figure 18.

The basic idea is to store all results from running the different software tools into the "data" folder and, after processing them, storing the table obtained in a "results" file inside the "results" folder. As a consequence, these two directories share very similar structures, with the difference of the GTF files and the preprocessed segmentation only being in "data". The final visualization is stored in the "plots" directory, and the "logs" folder is meant to store all sorts of information about the run of the pipeline, in this case the data on the amounts of intervals for each gene biotype.

```
Snakefile
scripts/
        CreateAggregationResults.py
        CreateNucleotideResults.py
        CreateGeneBiotypeGtfs.py
        Visualization.py
results/
        aggregation/
                general/results
                gene_biotype/
                        protein_coding/results
                        lincRNA/results
                        ...
        gmtk_parameters/results
        length_distribution/results
        nucleotide/results
data/
        ...        // The same 4 than in "results/"
        segmentation/segmentation.pkl.gz
        gene_biotype_gtfs/
                protein_coding/hg38.gtf
                lincRNA/hg38.gtf
                ...

plots/plot.png
logs/gene_biotype_results
```

Figure 18: *File system after the first development stage*

This is the structure obtained from scratch when executing the pipeline from within the same folder where the Snakefile is found, but if called from somewhere else, this whole structure except the Snakefile and the *scripts* folder would be created in that place instead. This is because of the use of relative paths throughout the pipeline. Even if the structure is already determined, the names for most of the important folders are stored in variables in the code so that they can be changed easily. Notice that, in case it was of interest, the same name could be specified for the ”data” and ”results” folders, and the result would be that the processed results would find themselves together with the outputs from the tools, because of both directories following the same structure.

7.3 Making the tool available

Once a functional version is developed, the priority is to make it accessible to anyone who may want to use it. This is specially important because of the possibility to obtain feedback from the community. As it has already been mentioned there are a few ways to accomplish this objective. The option selected, involving Python packages, is the most used for Python software overall, as well as being the better option for the users in terms of what they need to do to obtain the software in a functional state. The alternative to this option is the one suggested by Snakemake’s developers [26] to distribute pipelines using their software, which consists on using a GitHub repository to be cloned with instructions for its correct use. However, one of the main objectives of this project is to make the software as easy to use as possible so that it really is a turnkey system. As a result, the alternative is considered as a bad option for this specific project and a Python package needs to be developed, as well as correctly distributed using the tools already mentioned. In the coming sections it will be explained in detail how the development of this stage took place.

7.3.1 Package creation

The first step when creating a package is to come up with a name for it. This may seem something of little importance, but in fact a bad name can make the tool less approachable to the user, as it may not be clear what it refers to. Its originality is important as well in case the tool is finally used by a lot of people. The name chosen in this case is Segzoo, so this is the name to which the tool will be referred from now on. It keeps the ”seg”(from segmentation) half that makes it obvious that it keeps a connection both to Segway and Segtools, while adding the ”zoo” to give the idea that it deals with a lot of data from different sources, combining them all up into one single automated pipeline.

For the tool to be available some changes were needed, both on the structure and the files that are part of the tool, and in the software developed up until now. These changes will be described more precisely next.

Structure and files

To create the package the official guidelines [59] were followed. The library *setuptools* is used, which means having a file called `setup.py` as well as a `setup.cfg` sometimes and keeping a specific structure for the information that needs to be present on these files and more. The whole structure needed for the package is showed in Figure 19.

```
LICENSE.txt
MANIFEST.in
README.md
setup.py
segzoo/
        __init__.py
        main.py
        create_aggregation_results.py
        create_gene_biotype_gtfs.py
        create_nucleotide_results.py
        Snakefile
        version.py
        visualization.py
```

Figure 19: *Segzoo package's file system*

For this project the `setup.cfg` file that normally contains configuration details for the `setup.py` file is not needed, and that's the reason why it is not present in Figure 19. `MANIFEST.in` must contain references to all files that are intended to be included in the package other than those that are already added automatically, like Python scripts (files with the ".py" extension). `LICENSE.txt` contains the license agreement according to the regulation applied, which was explained in section 6. The `README.md` like all README files contains general information about the software it refers to, in this case Segzoo. In this case it uses the markup language Markdown, shown in its extension ".md", which will allow to easily add all sorts of figures to the file. The information stored here will later on be very important, as it will be the first thing the user sees from Segzoo both in the Bitbucket repository where it has been developed and on the PyPi page automatically created for it.

As recommended in the tutorial followed [59], all code is stored inside a folder named like the package itself. The previously used "scripts" folder is eliminated to make all the mutual references between scripts much more intuitive and the files easier to access. Note also that some names were changed to match the Python standards. A new Python file called `version.py` is added containing a single global variable with the version of the package, initially `__version__ = "1.0.0.dev1"`. This variable will be used in multiple scripts from the package, so the objective of creating it in a separate file is to centralize its value for when it needs to be changed. Finally, the `__init__.py` file identifies Segzoo as a Python package, while the use of `main.py` will be mentioned on the following section.

Finally, the base of the package, `setup.py`. In this Python file all sorts of information about the package are provided, from the developers name and email to the description, stable url, dependencies and entry points for the project. The entry points are command line entries with which the software can be invoked. The version must also be specified in this script, as well as a reference to the README and the LICENSE used. All this information is stored in the shape of arguments that are passed onto a single function call with the name ”setup”, from the *setuptools* library.

Command line interface

This tool only need one single entry point, as it doesn't present any independent software like Segtools does. This entry point specified in `setup.py` refers to the `main.py` file where all the options available to the user are stated and the final call to the Snakefile is made. To make the tool as easy to use as possible for the user's perspective it was deemed indispensable that only one argument was mandatory to run Segzoo: the segmentation. The ”params.params” file necessary for the learned parameters was moved to an optional argument as well, which had some implications that will be seen later. The specification of the accepted arguments in the Segzoo call was done using the library *argparse* which automatically creates the argument ”-h” or ”–help” that outputs all the information regarding accepted arguments.

```
 usage: segzoo [-h] [--version] [--parameters PARAM] [-o OUTDIR]
 [--species SPECIES] [--build BUILD][--prefix PREFIX] segmentation

Download necessary files, run workflow and obtain results

positional arguments:
segmentation          .bed.gz file, the segmentation/annotation output
                      from Segway

optional arguments:
-h, --help            show this help message and exit
--version             show program's version number and exit
--parameters PARAM    The params.params file used to obtain the
                      gmtk-parameters
-o OUTDIR, --outdir OUTDIR
                      Output directory to store all the results
--species SPECIES     Species of the genome used for the segmentation
--build BUILD         Build of the genome assembly used for the
                      segmentation
--prefix PREFIX       Prefix where all the GGD data is going to be
                      downloaded
```

Figure 20: *Help displayed when running* `segzoo -h` *on the command line*

As seen in Figure 20, a full guide with descriptions is shown when running the command `segzoo -h`. The optional arguments can be specified in any order, and most of them are followed by a value after their keyword. When added to the Segzoo command line call, their different values are parsed and stored in a dictionary-like object that can be added to the Snakemake call. As a result, all these values are now assigned to the variables that originally had constant values in the pipeline. Because of this, all optional arguments must have default values specified so that these variables always have a valid value.

The output directory's default name is ”outdir”, where the ”data”, ”results” and all the rest of output folders from the file system will be created. For the species and the build, the values used during development are the ones taken as defaults: ”Homo sapiens” and ”hg38”. Furthermore, this is the most used combination of the two as ”hg19” is slowly being used less and less. The prefix, which specifies partially the directory where the GGD downloads will take place, has as a default the path to the current conda environment being used. This is obtained by assuming that Segzoo has been installed either by running conda or using the PyPi package but with a Python installation in anaconda. These are the two most common use cases, so it is a safe assumption to make. Alternatively, the user can specify another location where the path `share/ggd/`*`species/build/recipe/`* will be created for the downloads, as this portion of the path cannot be modified.

Pipeline changes

During this process some changes were needed to be made to the pipeline that had been developed up until this point. As we just saw, a set of optional arguments were created for the user to specify. The ”outdir” argument made it so an extra layer had to be added to all paths specified in the Snakefile, which didn't take into account having to store all outputs in a specific folder. Regarding the learned parameters, its argument doesn't present any default value. This is because it is not possible to predict where the ”params.params” will be, and even whether it exists. For this reason, this argument is different than the rest in that it causes the pipeline to behave differently if it has been specified or not. In this case, if it is not included in the Segzoo call, no GMTK parameters can be shown in the final visualization and, furthermore, all needed analyses regarding them must not be executed. Changes were made to the pipeline so that the learned parameters results were needed conditionally, and the visualization showed an empty space where the parameters should be if no path was specified.

When building the package there was a question that needed to ba answered: how is the user going to install and use this tool?”. The answer given made it so that some changes were needed. For starters, it of interest for the tool to be usable many times with different inputs. If this was done with the current pipeline the GTF file processing, which creates a bunch of new GTF files, would be executed each time a new directory with results is created. As this data is completely independent to the segmentation itself and only depends on some variables like the species or the build used, it was decided to move its creation path to the same one where the GGD data is stored. This way, this data can be shared between all Segzoo runs that share species and build, even if they are done using different segmentations. Moreover, the gene biotype stats that are created by this same rule are also stored in this directory. As a result, future executions of the pipeline will need to execute less rules, which will make them faster, and no duplicate data is created.

Problems encountered

Aside from the small problems that were resolved during the development of this stage, there were important issues that were found when testing the Python package created. As mentioned, in the `setup.py` file all dependencies needed to be specified. One of them is BEDTools, used in the nucleotide analysis of the pipeline. The reason why these dependencies need to be specified is for them to be automatically installed when installing this tool. The issue in this case is that BEDTools is not available on PyPi, and as a result the installation fails when having it as a dependency. Its is available, however, on Bioconda, the Anaconda channel, for which we had plans to create a version of this tool in the future. Once this is done, the dependency can be added, and the only way to manage this situation is by clearly specifying in the README file that BEDTools needs to be previously installed.

In the package it is also necessary to state the Python compatibility of the tool, which basically means to specify a Python version from which the compatibility will be properly tested and guaranteed. In this case, the version for all Python packages used as dependencies was checked, because it is not possible to have a tool compatible with a version if all their dependencies aren't. Snakemake, for example, is only available from Python 3.3 onward, which coincides with the big majority of the tools used. Because of this reason, it was chosen to take the latest Python version, 3.6 as the compatible version, even if it should be able to work on many older versions too.

An exception to all this however is Segtools, which only had compatibility for Python 2.7 at that point. This was already known beforehand but the solution that was first thought of, regarding the use of conda environments to run different versions of Python, was finally considered as not good enough. Doing so is not intuitive and makes the user take a lot of extra steps before being able to use Segzoo, which goes against its objectives. This combined with the fact that Python 2 was going to be dropped from support in less than 2 years (Jan 1st, 2020), made it so another solution was selected to proceed. This solution consists on porting to Python 3 Segtools partially, enough for the project to continue with a full Python 3 dependency and not relying on any Python 2 software.

Results obtained

Although these last issues had some consequences as described in the following section, a version of the package without Segtools or BEDTools as dependencies was already available after this stage was completed, and completely usable if both software were already installed and this tool was installed in a separate Python 3 environment. Thanks to PyPi, to do so is as easy as executing `pip install segzoo` in said environment. Furthermore, a web space is automatically created for the package in which all data specified in its creation is shown and each time a new version is uploaded it gets automatically updated. The url for this page is the following: `https://pypi.org/project/segzoo/` [60].

7.3.2 Schedule changes

Because of the issues previously mentioned this stage caused big changes needed in the planning. This change of plans made it so the initial scheduling had to be remade. Because of the inability to upload the obtained package to Bioconda without the full dependency tree up to date, this new task will have to be done between both main tasks from this stage. The estimated time for the new task, as well as making sure Segtools' new version is available, is of 150 h. As a result, the new schedule is the one in Figure 21. All times for the stages that have actually been finished have been updated to its real time instead of the estimated time from section 4. Note that thanks to the fast end of the first development stage, even with the new task being added, some days are left available in comparison to the initial planning. These days have been given to the second development stage to be able to add more functionality and optimize the pipeline better.

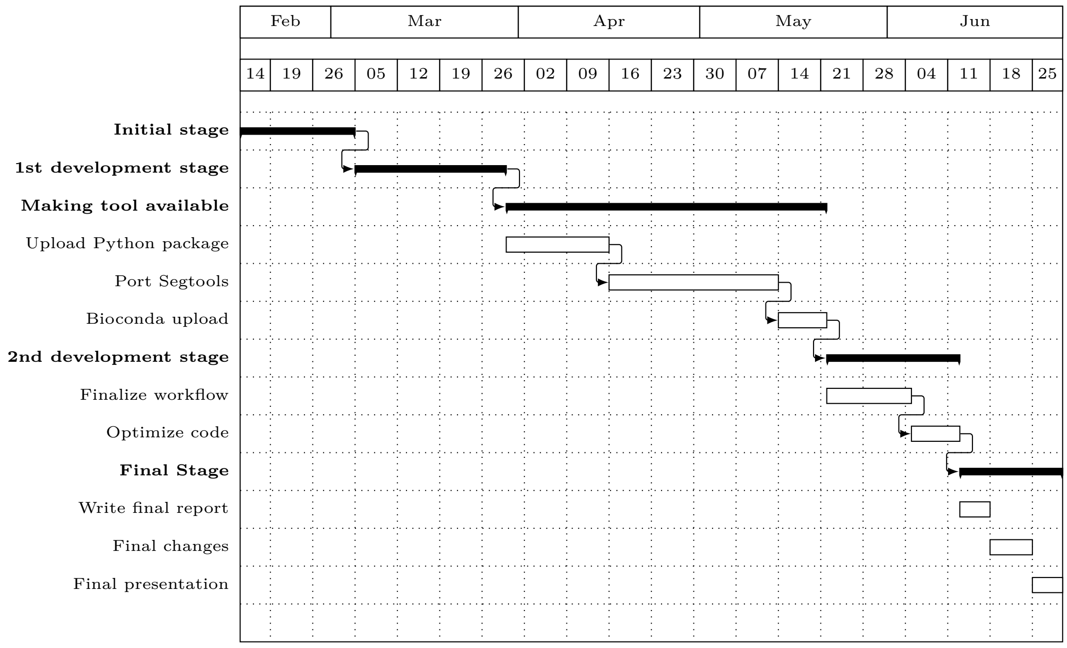


Figure 21: *Update on the Gantt chart of the project*

Although once again this schedule presents the tasks sequentially, it is very probable that this is not developed this way in reality like it has already been done during the first development stage. Specially during the new task's execution, when the new version of Segtools needs to be uploaded there will most likely be "dead" time, during which the development of the tool can continue while waiting. As a result, it can be considered that the second development stage can overlap with this entire stage, including the Bioconda upload for the tool.

This change in the schedule once again does not have any important effect on the final budget for the project.

7.3.3 Porting Segtools to Python 3

Getting into such a big software like Segtools and applying some changes to it, even if only partial, is no easy task, which is why a quite big amount of time was estimated for this process to be finished. The partial port to Python 3 will be done so that all tools from Segtools used by Segzoo are available both in Python 2 and Python 3, while leaving the others on Python 2 only because of time constraints. This stage is not this project's objective in itself, so the least amount of time needs to be invested in it. It will now be described what exactly needed to be done and how the issue was addressed, seeing some of the problems encountered on the way.

Python 2 and 3

As an introduction, the problematic and main differences between Python 2 and Python 3 are going to be explained next. This version update represented a big change in comparison to the smaller version updates like the one from Python 3.3 to 3.4, for example. For years now both Python 2 and 3 have been widely used, as a lot of already developed software is on Python 2 and, because of dependency issues, there are still new tools and packages being developed using this version. However, although support has been maintained for both versions during these years, the official end-of-life for Python 2 has already been announced, being the date the beginning of 2020. Because of this, there is an increasing interest in porting all Python 2 software to Python 3. However, due to the current situation, this is usually done while keeping Python 2 compatibility too, using a set of tools available as well as some official resources that Python developers made available for the community to use.

We will now take a look at the most noticeable changes that Python 3 brought and that may be important for this task:

1. Imports behave differently, the way the division operator works is more consistent on Python 3, and the `print()` function call is made differently. These three changes have been put together because they all are resolved by adding similar lines on top of the code, which import functions from `__future__`, a built-in Python package created to allow compatibility between Python versions. For example, `from __future__ import print_function` allows to make `print()` calls in Python 2 like in Python 3.

2. Among a lot of syntax changes, the syntax regarding both catching and raising exceptions is significantly modified. Luckily, the Python 3 syntax for it is also valid for Python 2.7, which is the latest version of Python 2 and the only one for which the support has been kept.

3. A general change to iterators was made, affecting most notably the functions `map`, `zip`, `range`, `next` and a bunch of dictionary iteration functions. All these functions now return iterators instead of lists, which causes errors when trying to use the result more than once.

4. A list of built-in libraries were changed and some of them even removed. It was made sure however that good alternatives were left for all those libraries still needed.

5. While both *Text* and *Bytes* types of *strings* were treated as one in Python 2, a heavy differentiation is made between them in Python 3. This makes it so most Python 2 software that treats with *strings* without specifying its subtype is incompatible with Python 3.

Methodology

As mentioned previously the objective of this section of the project is to obtain a version of Segtools that gets enough Python 3 functionality for Segzoo to run properly with it as a dependency, while keeping all current functionality with Python 2. To do so the official porting guide [61] was followed while working on a forked version of the original repository for Segtools.

Segtools consists on a lot of different files that take care of the software's functionality partially. An initial study was made to see which files needed to be modified. Those included the main files for each of the tools from Segtools used in Segzoo, those files referenced in them and all the ones responsible for the package creation, which is done the same way as in Segzoo.

For each Python file to modify, a software called *python-modernize* is used. This tool takes care of most of the compatibility issues between Python versions that are easily fixable, like the first 4 mentioned in the last section. The library *six*, which provides functions and classes that behave differently whether if used with Python 2 or 3, is widely used by the tool. For example, the function `six.range()` can be used, which will always return an iterator and so no different results are obtained depending on the version used. The software however is not perfect, so a close look needs to be kept on the changes made by it for each of the files modified. Even after all this when trying to run the code a lot of bugs will most likely still be found. This is because the change number 5, regarding *strings*, cannot be automatically dealt with.

Debugging these issues will require to look at variables and results obtained mid-execution to check for their types and values. The use of the called breakpoints allows to do exactly that. This is done by using debugging tools like the ones offered by PyCharm [62], which are the ones used during the course of this project.

Finally, when no errors are found some proper testing needs to be done on the software. To make sure changes keep all specifications, the tests will consist on comparing the outputs obtained for the same inputs by the version of Segtools being modified both uing Python 2.7 and Python 3.6, as well as by PyPi's version of Segtools. This way, we can make sure that the results are correct using both Python versions. For this purpose a series of scripts were created so that the tests could be done automatically.

Development

During the development of this stage following the methodology mentioned some unexpected issues were found and mostly dealt with. Small packages used by Segtools were found that also needed to be updated to Python 3, as well as being added in the dependencies list. Luckily, all this software, including Segtools, is maintained by the *Hoffman Lab* so it was easily accessible and changes were made quickly.

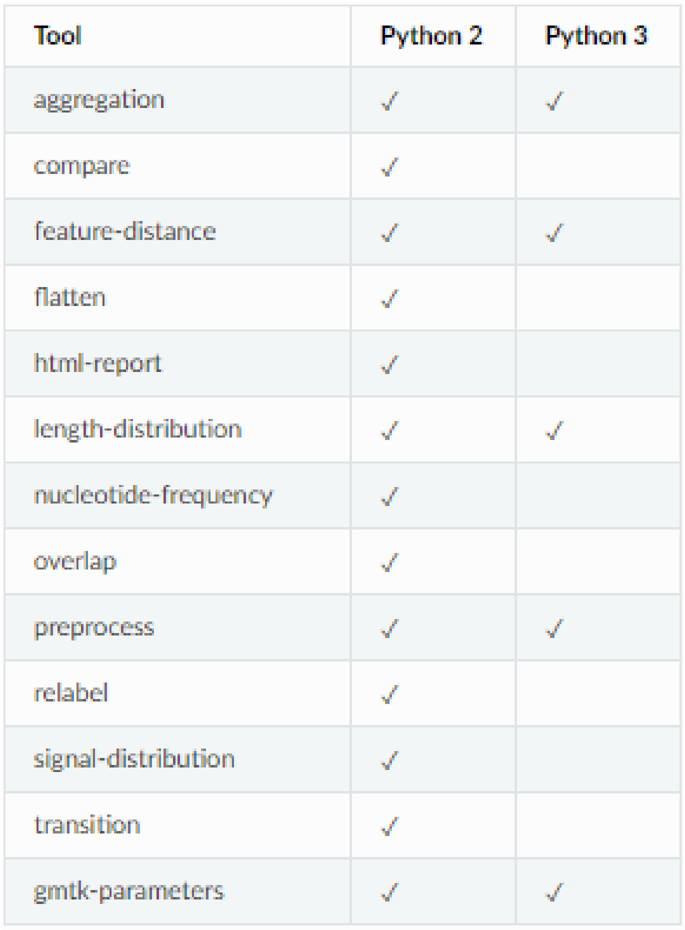
As mentioned previously, a selection of the tools to be updated had to be made. These were *length-distribution*, *aggregation*, *feature-distance*, *preprocess* and *gmtk-parameters*. However, all these tools have code that refers to the creation of an html-like file used for the execution of the *html-report* tool that was mentioned in subsection 2.2. This specific code particularly gave a lot of trouble when being executed with Python 3. As this tool is not wanted for Segzoo, the code was modified so that a message is prompted whenever this file creation is attempted with Python 3, warning the user of its incompatibility.

The `setup.py` file also needed to be modified, and way more than the other ones. This is because the entry points definition is being made there. Segtools has an entry point for each tool, so it was decided to remove all entry points referring to unavailable tools when using Python 3. This way it is much clearer to the user what can be done and what can not with each version, instead of relying on error messages being shown. Two variables were created in the code which contained the list of entry points for Python 2 and Python 3, and the adequate one is used when installing the package with one of the versions.

Decisions like this last one were made so that it will become easy in the future to add tools that are only available for Python 3. It is important to keep in mind that this version of Segtools is a partial one that will need to be expanded in the future, so everything possible was made to make the future transition to Python 3 smoother.

While up to this point the stage was kept within schedule, the duration of it was extended in the end due to various reasons. For some of the tasks performed from now on the time factor is not entirely dependent on the developer but on other stakeholders. As an example, all changes are requested to be added to the code, but a responsible needs to make sure the changes are correct and then accept them. Furthermore, it was considered necessary to modify Segtools' documentation [21] to reflect the changes that have been made. For instance, the table shown in Table 10 was added in a new subsection called *Python 2/3 compatibility status* and the dependencies section was updated.

When the final version of Segtools was finished and merged into the main repository with its documentation updated too, the only thing missing for it to be added as a dependency for Segzoo is to upload the package. As it was uploaded on PyPi, Segtools can already be in the dependencies list for Segzoo's package on PyPi. However, its upload to Bioconda encountered some issues, which affects the ability to do the same with Segzoo.



Tool	Python 2	Python 3
aggregation	✓	✓
compare	✓	
feature-distance	✓	✓
flatten	✓	
html-report	✓	
length-distribution	✓	✓
nucleotide-frequency	✓	
overlap	✓	
preprocess	✓	✓
relabel	✓	
signal-distribution	✓	
transition	✓	
gmtk-parameters	✓	✓

Table 10: *Compatibility table added in Segtools' documentation to keep track of the state of the Python 3 port, created using rst format in Sphinx*

7.3.4 Bioconda upload

Bioconda is the channel from anaconda where bioinformatics software is stored. For a new recipe to be added, which basically means adding a new Python package or tool, a similar procedure than the one followed to modify Segtools. All the contained recipes are hosted by a GitHub repository [50] that must be forked before adding the new recipe and then asking for a merge. The big difference with Segtools is that the procedure of having the changes accepted can take a very long time, specially since some very rigorous automated tests don't allow you to ask for the merge if any type of error is found.

To build a recipe the official tutorials can be followed [63]. A recipe consists on a file named *meta.yaml* which contains all information about the package and a script called *build.sh* which specifies the commands to build the package. A similar extra script is needed if wanting to run it on Windows too, but this is not the case for Segzoo. These two files combined form the recipe, which is used to create the package on a computer for it to be installed afterwards.

As it has been mentioned before, the conventions followed in the making of these two necessary files are very strict and sometimes messy. Even if only the name and version of the package are the obligatory information to add in the configuration file, for the build to actually work there are a series of sections in which more information must be specified: source, build, requirements, test, outputs, about and more. For each of these sections a series of variables can be assigned. The whole set of them with their descriptions and expected values are explained in detail on the conda documentation [64].

Luckily, if looking at the tutorials available [63] one may notice that there is an option called `conda-skeleton`, which allows to create conda recipes almost automatically from PyPi packages. Because of this being the case for Segzoo, this option is chosen as it makes the whole process a lot easier. By using the command `conda-skeleton` and specifying the PyPi package, a *meta.yaml* file is automatically generated with all the available data in the package already stored correctly. This goes from the dependencies to the package description, completing the file in its entirety. The reason why this is considered ”almost” automatic is because the results must be checked for errors and some changes may be wanted. In this case, BEDTools can finally be added as a dependency too, in addition to the rest of dependencies already present on the PyPi package.

Once the recipe is created, it must be tested before attempting to merge it. As explained in the last section, Segtools was still pending for its Python 3 version addition in Bioconda when the final date of the project arrived, so the tests with it specified as a dependency fail. However, all tests done removing this dependency turn out fine, which should mean that, once Segtools is finally available, Segzoo should also be ready to go.

While this stage finally took a lot longer than expected mainly because of the waiting times that were mentioned as possible issues, this doesn't have any direct effect on the rest of the project, either in terms of the schedule or budget. As mentioned previously the second development stage, described in the next section, was started earlier than scheduled to make up for the time this stage was taking to complete. In the end both making the tool available and developing the tool were done in parallel, updating any changes made to the package regularly. Furthermore, with the PyPi version working already the tool could be already used by the community and some feedback could be collected as expected.

7.4 Second development stage

The first development stage ended up when work on the software's availability was decided to begin. However, the state of Segzoo in which it was left then was not the one intended for the end of the project. As seen previously, the schedule planned for the remaining of the project's development after the changes needed after the problems encountered in the last stage could not be followed. Instead of keeping a sequential approach to it, it was decided to begin the second development stage whenever it was possible and the other stages could not see any progress. Starting the beginning of May, the development on the tool, further from the package creation, was retaken. The main decisions and changes applied are going to be described in the following subsections.

7.4.1 Visualization

The most important changes needed in the tool from its state after the first development stage are probably those related to the visualization. The way it was left made it really hard to interpret and even some of the information was not shown in a meaningful way whatsoever. Although this is considered one single stage, the changes that this visualization went through can be separated into three, with a resulting visualization coming out of each of these. We will now see which was the focus for each of the three iterations of changes and an explanation for the decisions taken in each of them.

First iteration

Although the main visualization code went through a lot of changes, a very important one was done in the beginning of this stage. This change referred to the structure of the code, separating into functions the segments of it that prepared each of the results, and adding more global variables on top of the file to easily control some aspects of the visualization if needed. These standards will be kept during the rest of the development to make sure the code is easily readable and modifiable.

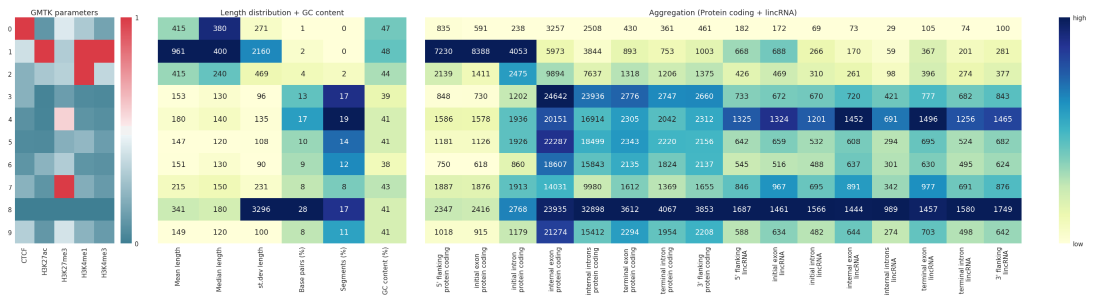


Figure 22: *Example of a visualization obtained after a first round of improvements*

During this first iteration of changes the main focus was to make the figure easier to look at. To this end, the labels for the columns of the heat maps were changed, from the ones used in the data frames in Pandas to more intuitive names without points or dashes between words. Some of the values shown in the heat maps were also altered, changing the fractions to percentages to get rid of the "0." on front (and indicating this in the column name) and getting rid of all unnecessary decimals. This set of changes makes the heat maps look a lot cleaner while holding the same amount of relevant information.

An important isolated change applied during this phase was the normalization in a different manner of the GC content. The previously applied normalization, which consisted on scaling the highest value to 1 and the lowest one to 0, made it so the

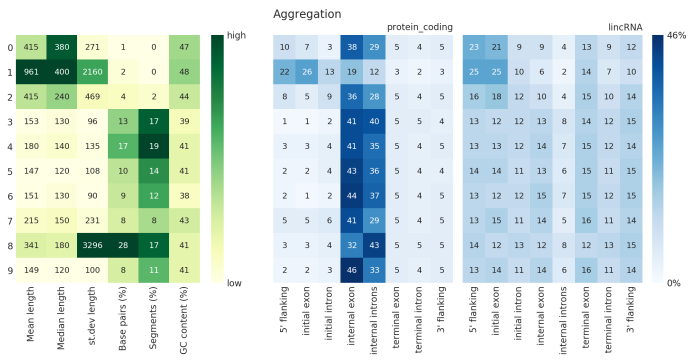
GC content looked very different for all the labels because of their colors when the numbers shown contradicted this statement. While other columns can present very different values and cannot be normalized in any other way safely, the GC content is a percentage that usually presents similar values, so a better normalization can be applied. Another option would be to assign the 0 to 0% and the 1 to 100% of the GC content. However, this would cause the opposite effect from what has just been described. As seen in Figure 22 the values for the GC content are all very similar, so this normalization would make all of them look like the same, assigning a very similar color to all of them. The middle ground between these options is to use different boundaries for the normalization, in this case 35% and 65%. These values were chosen because they are rarely ever surpassed and they make the colors of the heat map more meaningful.

Finally, as seen in Figure 22 different color maps were used for the tables this time. The objective was to have good-looking color maps that share the characteristic of more intense colors being the higher values, while more pale colors means lower values. Furthermore, the heat map used for the learned parameters is the one used in Segtools, so that interpreting them is more straight-forward and intuitive for the researcher. Other than that, color bars were added to the right of the heat maps with different color maps so that it is easily known what the colors mean for each of them. The boundaries for both the *mix matrix* and the aggregation heat map are set to "high" and "low" as no specific value can be assigned to them.

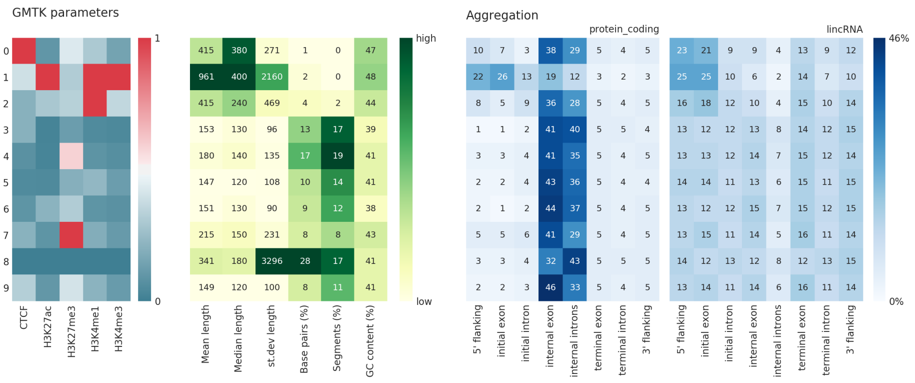
Second iteration

For the second iteration of changes, the focus was to make the aggregation results meaningful. Because of this the *mix* and aggregation heat maps were differentiated and different color maps were assigned to them again. As a result, another color bar is needed to specify the boundaries for the color map added.

As seen in Figure 23 the results for the different biotypes for the aggregation are separated into different heatmaps to make them more distinguishable. Although all examples shown always present the biotypes protein coding and lincRNA, if more were to be added extra heat maps would appear progressively in the visualization. It can also be observed that the values presented in the heat maps has completely changed from the last iteration of changes. These results now present percentages, as specified in the color bar on the right. This percentage is the amount of counts found by the aggregation in a specific component (5' flanking, initial exon, initial intron, etc.) compared to the total counts for that label on a gene, so each row adds up to 100%. For example, if looking at label 9, we see that segments presenting this label are a lot more likely to appear in the internal regions of an idealized gene than in all other components. This information proves a lot more useful than just the amount of counts found by the aggregation, which were the results presented before these changes. In terms of the heat map, the boundaries used for the color map are the 0% and the highest value found across all biotypes. This is necessary so that all aggregation heat maps share the same color map and their results can be compared with each other.



(a) Example of plot obtained without learned parameters being specified



(b) Example of plot obtained with learned parameters being specified

Figure 23: *Visualization resulting from a set of improvements, both with and without the parameters*

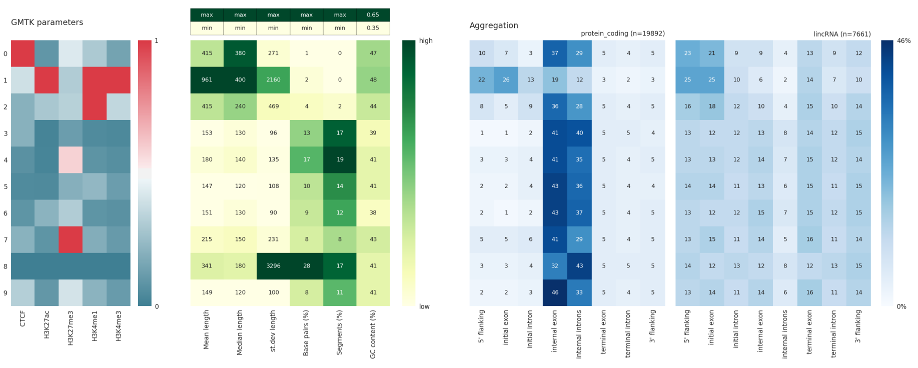
As seen in the results in Figure 23, having different gene biotypes represented in the visualization can prove very useful as they show meaningful differences. Because of this, the code was developed in a way that more gene biotypes can be added easily and the ones present can be modified, as it will be seen in the following section. These changes were added along with other modularization improvements that make the plots' resolution easily modifiable, and that allow a variety of new heat maps to be added automatically in case they are needed. All positioning and scaling is also decided accordingly to the sizes of the matrix, which includes the amount of labels used for the segmentation, which is variable.

Along with other fixes, the possibility of not showing the GMTK parameters was improved, by making sure the label names are shown in the *mix matrix* instead if needed, and by actually removing the space where the parameters would be found instead of leaving a blank space there. The difference between both versions of the visualization can be seen in both plots of Figure 23.

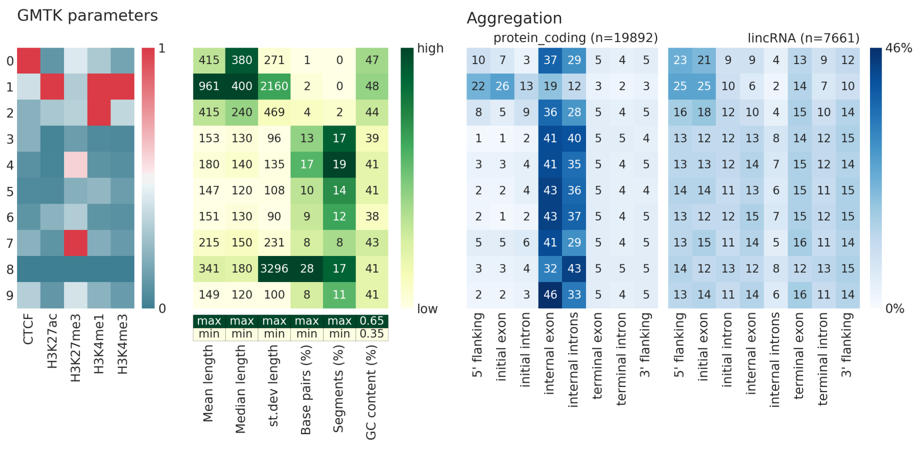
Finally, the heat map titles were improved for readability and sub titles were added to specify the gene biotypes for the aggregation plots. The title for the *mix matrix* was removed so that other columns can be added in the future without requiring further changes. The font sizes for all labels on the plot's axis and the color bar boundaries were made equal to those onside the heat maps, while a bigger font is used in titles. The values for these two font sizes can be easily modified by changing two constants on top of the script.

Third iteration

Up to this point, the most important changes have already been made and the visualization accomplishes all objectives set for it. The plots make the summarized results easy to read, and they are shown in a way that meaningful hypothesis can be concluded from them. All information is compacted so that cross-results analyses can be done. The techniques used to accomplish all these objectives like the use of heat maps with differentiated color maps are intuitive and easy to understand at first sight with the information provided. The focus for this final iteration of changes to the visualization is to polish some aspects of the plot to obtain a final version for the tool obtained in this project.



***(a)** Example of plot with the table on top of the heatmap and small font scale*



***(b)** Example of plot with the table on bottom of the heatmap and big font scale*

***Figure 24:** Comparison of visualizations with different values for the table position and font scale*

Although most of the information is intuitive, the fact that the boundaries for the *mix matrix* are "high" and "low" doesn't give much information. Furthermore, there was no way currently to know how the normalization for the GC content was made by looking at the plot. Something needs to be added so that it is obvious that the normalization for the columns of the heat map is sometimes done differently. The solution that was suggested for this is to add a table that shows what the ”high” and ”low” values mean for each of the columns. This table, which can be seen in Figure 24, must be located near the heat map so that the columns align and uses the ”high” and ”low” values in the color map to indicate the rows of the table referring to each of them.

As observed, two positions are considered for the table. The first one is on top (Figure 24a) which takes advantage of the space left by the absence of title in this heat map. The second option is to put it on the bottom of the heat map (Figure 24b), which means having to move the axis labels to make space for it to fit. The latter option makes the columns more easily identifiable because of the column names being next to the table, and it avoids having to constantly jump from reading the top and the bottom of the heat map. For these reasons, this option is considered the most appropriate for the project and will be set as the default.

Another decision to be made is whether to put the words ”max” and ”min” in the cases of the length distribution results or actually put the value of the maximum and minimum values in the table. Although both options are viable and would make clear which are the scales used for each of the columns, the ”min” and ”max” option is accepted as the default because it causes the scale used for the GC content to stand out even more. It becomes intuitive and easy to see that this scale is special compared to the rest, which wouldn’t be so obvious if all values were numbers.

Along with all the options for the table added, it was considered the usefulness of being able to scale all font sizes easily. This can be useful depending on the use thought for the visualization obtained: a publication, a presentation, research, etc. In some cases like in a presentation it is of interest to make the numbers as big as possible, even if the figure looks a bit weird that way, while this is not necessary for publications like papers. In Figure 24 two different versions of the scale can be compared, being the scale 1 in Figure 24a and 2 in Figure 24b. A middle ground is set as the default, with the scale being 1.5. Notice that the scale also affects titles and the labels in the axis, as well as the color bar boundaries.

All these aspects that can be modified like the table position, the values shown in it and the font scale of the general visualization, as well as others that have not been mentioned like the height of the table (which is set to the same height as a row of the heat maps as default), are stored in constants on top of the file and commented so that they are easily reachable. Although this is done so that they can be modified or turned into configuration parameters of the tool in the future, it is important to keep good defaults for all of them. This way the simplicity of the tool is kept intact, while adding more possibilities for experienced users.

A final little addition is the number of genes found for each gene biotype, which can be found next to the biotype name on top of its aggregation heat map. This information is facilitated so that the researcher has an idea of the validity of the results obtained, as an aggregation with a small amount of genes does not give meaningful results. This information is obtained from the file that was already created to obtain statistics on the gene biotypes back in the first development stage, during the creation of the GTF files used for the aggregation.

7.4.2 Pipeline

Although the visualization is the element that was changed the most during this stage, the entire pipeline saw some modifications that need to be mentioned. Starting by those related to what we just discussed, some improvements were made related to the gene biotypes used to obtain the results for the aggregation. A new file was created in the Python package called `gene_biotypes.py` which, similarly to `version.py`, only contains one variable that in this case consists on a list of the gene biotypes to be displayed in the visualization. This variable is shared to the rest of the package, and so the script that creates the visualization has access to it. The default state of this variable contains both protein coding and lincRNA, followed by all other biotypes accepted commented. This way, if it is of interest to change this default, it would be enough to uncomment or comment biotypes. This scenario, however, is very rare as most of the genes pertain to one of these two biotypes, and the rest are not usually studied. It is important to note that both repetitions and reorganizations in the list will be reflected in the visualization.

Optimizations

As specified in the original schedule the objective of this second development stage wasn't only to add more functionalities or improve the visualization of the tool, but also optimize it. Several code optimizations were constantly applied, but the following two changes are the ones that had the biggest impact in the performance of the tool. The first optimization consists on running the aggregation analysis only on the biotypes that are of interest, which are accessible through the `gene_biotypes.py` file as it has been just mentioned. Taking advantage of the ability to modify this information easily by the user, it is possible to now implement an optimization like this one, which resembles the one originally suggested when in the first development stage of limiting the analysis to the most important biotypes. By doing this, the use of the dynamic files is no longer necessary, as the exact number and names of the files needed is known now.

It was also considered to only create the GTF files for these biotypes, but because of how their creation works almost no execution time is avoided. Creating all the files has the downside of storing a bit more information than necessary, but it prevents future runs from having to run the whole creation script all over again, so it was considered as the optimal solution for this. The aggregation analysis, however, can be treated separately much easier and no problem appears because of all the biotypes not being analyzed always.

Talking about storage optimization, the second big optimization was regarding all the GTF files involved with the pipeline, both the one downloaded using GGD and the ones created by the tool's script. This optimization consists in compressing all these files into the Gzip format. This format allows for big files to be compressed down to a lot smaller sizes, and presents the characteristic of being very easily undone on runtime, so many tools can work directly with files of this format. Both the pybedtools [58] library and *segtools-aggregation* can directly work with them, so not many changes are needed more than converting all the output files.

After these optimizations were implemented some tests were run to test the performance of the tool and provide the users an expected time and space consumed. The size of the directory where all the GTF files were stored was reduced dramatically from around 2 GB to less than 100 MB, with a ratio of around 1/30. Time-wise, both the installation process and running a full Segzoo exxecution (with external data needed) were tested resulting in around 8 min for the installation and around 30 min for the execution. It is important to note that this time will be greatly reduced in future runs because of all external data already being present, down to around 10 min.

Feedback implementation

As mentioned in the stakeholders section (subsection 1.4) it is of high importance for a tool like this one to open up to the users as soon as possible to obtain as much feedback as possible. Because of the changes in the schedule the Bioconda upload was not finished on time to be able to use this method to distribute the tool. A temporary solution was suggested using the PyPi package installation so that the tool could be tested by some users. From these tests the main feedback received was that the tool didn't work when running it on a cluster instead of a local computer. Although this tool has always been thought out to run on a single computer, it is true that this kind of analysis is normally run on a cluster of computers to do so quicker. The issue involved the library used for the implementation of the visualization called matplotlib, and was fixed thanks to this feedback.

A bigger problem still exists, however. The way most clusters work is that they have a single node with connection to the internet, while the ones used for computation-heavy operations which are the big majority do not have this connection. When trying to run Segzoo on a cluster without connection the GGD downloads fail, so it becomes necessary to do it on the shared node, which is not recommended. The best way to deal with this is to execute one run on this node just for the downloads to happen, and then stop it. All future executions of Segzoo can be done on any other node while specifying the correct prefix to the data downloaded previously, because no internet connection is required for the rest of the pipeline. To make this process easier a new argument was added to the tool, `--download-only`, which makes the tool execute only the downloading rules and therefore should be used when doing the first run on the shared node. This is accomplished creating a new rule with all downloads as inputs and targeting that rule instead of the one creating the visualization when the argument is specified.

7.4.3 Documentation

One last thing that was taken care of during this stage was documenting Segzoo. The most basic things like commenting crucial parts of the code extensively and such, although done during the whole development, were checked to be on point in this stage. Moreover, the README was edited to serve as an initial informative point for all Segzoo users. Initially this file, created during the packaging stage, was almost empty holding only a brief description of the project. A lot of new contents were added progressively to it.

After an initial introductory section briefly explaining what Segzoo is, sections were added with instructions on how to install the tool and how to use it. These sections were updated during the development continuously. On the first one, both PyPi and Bioconda options are described. The only difference between them is the fact that PyPi doesn't take care of non-Python dependencies, so they all need to be installed previously. On the second section a list of all arguments is presented with descriptions for each one of them, as well as a description of the way to modify aspects of the visualization by changing some information in the source files. Finally, a sample visualization obtained is shown with all results obtained being described.

This README file already contains all necessary data for a user to be able to use Segzoo without any trouble. The guidelines presented in the paper 'Creating great documentation for bioinformatics software' [25] were considered at this point. The following summary is given as requirements for a good documentation, sorted in order of importance:

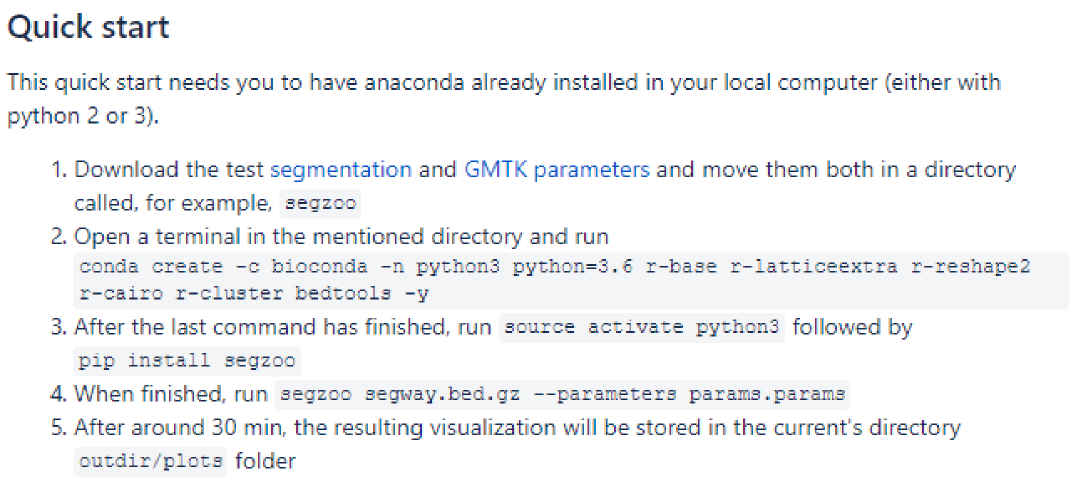
1. GitHub or Bitbucket page with code and issue tracker.
2. Readme that covers installation, quick start, input formats, and output formats.
3. Reference manual with detailed description of every user-configurable parameter.

At this point it was decided to aim for the two first points for Segzoo, staying with a complete README instead of creating a whole reference manual in HTML format as suggested in the paper. For this objective to be achieved the only thing missing is a "quick start" section. A quick start is a short enumeration of instructions to follow to get the tool working and get results from it by using some facilitated input data. In this case, links to the segmentation and "params.params" file used for testing during the entirety of this project were added, as well as the instructions to install and execute Segzoo with all needed dependencies. This quick start can be seen in Figure 25. This README file that consists on all the documentation for the project can both be found in the Bitbucket public repository found in `https://bitbucket.org/hoffmanlab/segzoo` [54] and also in the PyPI page found in `https://pypi.org/project/segzoo/` [60], which consist on the two main access points to Segzoo at the moment.

7.5 Final state

In this section an overview will be taken at the state of the tool as of the end of the project. Most of these things have already been mentioned in previous sections but the finality of repeating them in this section is, by adding them all up together, to generate an idea of what the tool does and how some of the objectives set from the beginning have been accomplished.

Segzoo is currently only available through the command `pip`. Because of the work done on Segtools, the version 1.2 which adds Python 3 compatibility to some of the tools is also on PyPI and will be installed automatically as a dependency. The only thing preventing the installation from being as easy as running one command are the non-Python dependencies of which Bioconda would take care: in this case, BEDTools and some R packages that Segtools needs. This is the reason why the current preferred way to install Segzoo is the one shown in the quick start section of the documentation, shown in Figure 25. However, when Segzoo is available for Bioconda in the future, it will be enough to run `conda install -c bioconda segzoo` on a Python 3 environment for the installation to be complete.



Quick start

This quick start needs you to have anaconda already installed in your local computer (either with python 2 or 3).

1. Download the test segmentation and GMTK parameters and move them both in a directory called, for example, `segzoo`
2. Open a terminal in the mentioned directory and run `conda create -c bioconda -n python3 python=3.6 r-base r-latticeextra r-reshape2 r-cairo r-cluster bedtools -y`
3. After the last command has finished, run `source activate python3` followed by `pip install segzoo`
4. When finished, run `segzoo segway.bed.gz --parameters params.params`
5. After around 30 min, the resulting visualization will be stored in the current's directory `outdir/plots` folder

Figure 25: *Snippet from the documentation written for Segzoo showing the quick start section*

If working directly in a Python 3 environment, there is no need to create a new one, but it is useful anyways to isolate the installation of all dependencies on a separate environment. After installing it, the command `segzoo segway.bed.gz --parameters params.params` can directly be run, specifying the segmentation and the parameters file previously obtained with Segway. From executing this, a figure very similar to the one in Figure 26 is obtained. In said figure all the defaults mentioned previously can be seen, as it is the figure obtained by default using the input files provided by the quick start section of the documentation.

Although all defaults are usually good enough, a set of optional arguments is available. The current ”help” shown for Segzoo after executing `segzoo -h` on the command line is the one presented in Figure 27.

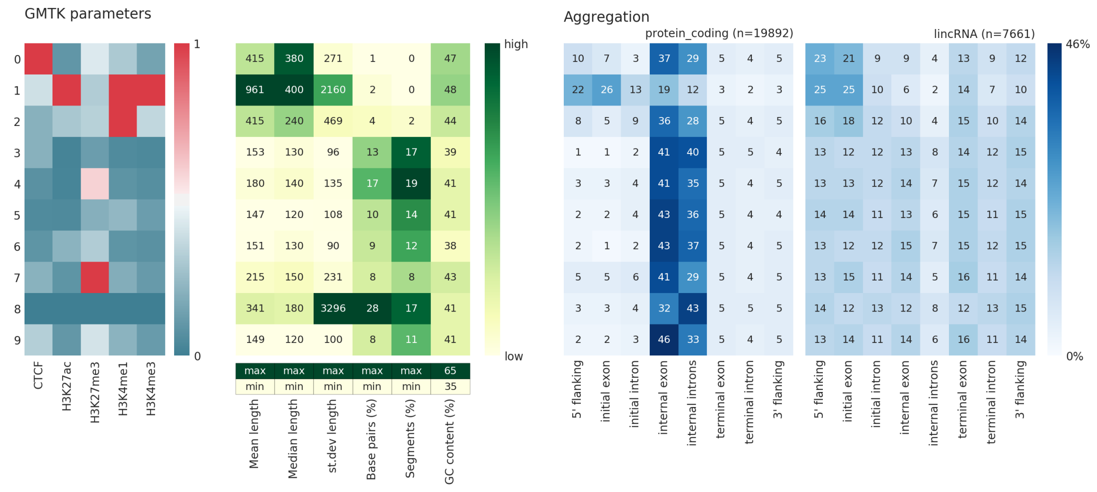


Figure 26: *Example of a visualization created by the final tool developed in the project*

```
usage: segzoo [-h] [--version] [--parameters PARAMETERS] [-o OUTDIR]
[-j CORES] [--species SPECIES] [--build BUILD] [--prefix PREFIX]
[--download-only]
segmentation

Segzoo is a tool that allows to run various genomic analysis on a segmentation
obtained by segway. The results of each analysis are made available as well as
a summarizing visualization of the results. The tool will download all
necessary data into a common directory and run all the analysis, storing the
results in an output directory. All this information is then transformed into
a set of tables that can be found in this same directory under the "data"
folder, that are used to generate a final visualization.

positional arguments:
segmentation          .bed.gz file, the segmentation/annotation output from
Segway

optional arguments:
-h, --help            show this help message and exit
--version             show program's version number and exit
--parameters PARAMETERS
    The params.params file used to obtain the gmtk-
          parameters (default: False)
-o OUTDIR, --outdir OUTDIR
          Output directory to store all the results (default:
          outdir)
-j CORES              Number of cores to use (default: 1)
--species SPECIES     Species of the genome used for the segmentation
          (default: Homo_sapiens)
--build BUILD         Build of the genome assembly used for the segmentation
          (default: hg38)
--prefix PREFIX       Prefix where all the external data is going to be
          downloaded, followed by /share/ggd/SPECIES/BUILD
          (default: ~/anaconda3/)
--download-only       Execute only the rules that need internet connection,
          which store data in a shared directory (default: False)
```

Figure 27: *Help displayed when running* **segzoo -h** *on the command line on its final version*

8 Future directions

Although the project has come to an end, Segzoo is open source software that will hopefully continue to grow and improve. In this section some of the ideas for near future directions for the project will be presented. The most obvious one is the upload of Segzoo on Bioconda. Although the recipe is already ready to be added to the GitHub repository [50], the issues with Segtools' new version not being yet in Bioconda makes this task impossible at the moment, as all dependencies must be already in an anaconda channel before. However, once this is resolved, it should be a priority to work on the addition of Segzoo so that the installation of the tool turns to be a lot simpler than the current alternative, as seen throughout this thesis.

Another obvious thing to consider is adding more information to the pipeline and the visualization, which should prove easy due to the existence of the *mix matrix*. Two first options are suggested for this, the first one consisting on the addition of results coming from the feature distance analysis that is already found, although not used, in the pipeline's code. The second option is one that has been studied briefly already, consisting in the addition of the FANTOM5 enhancers[65] overlap. FANTOM5 is a project that provides biological data of use for the bioinformatics field. The enhancers are interesting annotations to compare a segmentation to, like the gene annotation used in the current visualization. The original annotation, which is downloaded as a BED file, can be used in an aggregation just like with genes. The results present three components in this case only, two flanking regions and a single enhancer region.

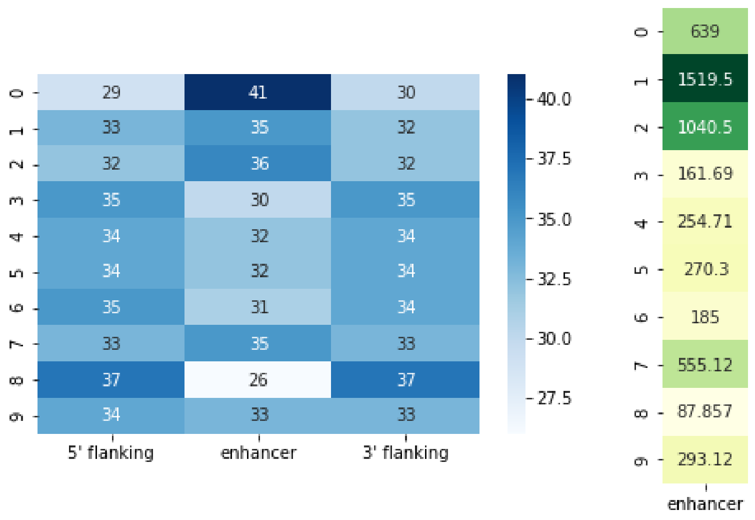


Figure 28: *Plots of the enhancer results treated like the gene aggregation and ready for addition into the mix matrix*

As seen in Figure 28 some options have been already considered for the results representation. In the left, the results are treated just like the gene aggregation ones and in the right only the enhancer overlaps are taken and the total number of base pairs covered by each label are taken into account. The second option gives a result almost ready to be added into the mix matrix, with some proper and better normalization being applied. On top of this, the analysis should be added to the pipeline as well as the annotation file used being added in GGD.

Although, as it has been mentioned, several strategies have been used to make sure it is easy for the user to modify the visualization's default parameters if needed, Segzoo's ability to be configured is something that could be widely improved. While the main point of the project was to make the tool as simple and easy to use as possible, it would be interesting to explore the possibility to add ways for the user to configure the visualization to his likings for example to make it ready for a publication. This could be done by adding the option to include a configuration file with a format like YAML or JSON which specifies information like those mentioned during this thesis, for example the font scale and the table position, and many more.

Finally, as described in the last section, Segzoo is not thought out to be used on a cluster. While some adjustments have been made after the feedback received to make it usable, the tool's usability on a cluster could be widely improved. This could be done by taking advantage of the cluster option that Snakemake provides, for example, which allows to specify detailed information for each rule on how it needs to be executed.

9 Conclusions

In spite of the fact that not everything went according to plan during the development of this project, the final result is a software that accomplishes all initial objectives. The tool developed is able to do automatically much of the work a researcher normally must do by himself manually, like downloading data or running analysis. The final visualization of the results has also been improved considerably throughout the project until a point where it already is very useful for any researcher trying to obtain a quick analysis on a segmentation.

Although the initial goals have been achieved, some of the ones that have been defined during the development of the project, due to the appliance of agile methodologies, could not be finished in time. This project was dependent on very rigorous time constraints which, combined with the appearance of unforeseen issues and third party time issues, made the schedule impossible to fully complete. However, these aspects that in the end could not be taken care of were at most secondary to the real purposes of this project. For this reason, the results can be considered all the same as very satisfactory. What has been accomplished here is the creation of a new software tool with a lot of potential to grow and have these little issues fixed eventually, even if this is done outside the scope of this project.

With every little issue found during its development, this project has made me learn a variety of skills both in the computer science and bioinformatics fields. Taking for example the Python 3 porting stage, which was not originally planned, but nevertheless was one that taught me a lot about a different scope of computer science from those originally planned for the project. This combined with the nature of the project itself, being based on the full development of a software tool, has enriched me with a lot of different skills and experience.

Personally, the development of this whole project has given me invaluable knowledge and experiences that will serve me good for the rest of my life. Even if there were difficult moments and a bunch of new things to learn all the way, because of the support I received and the good planning and methodology followed I was always able to move forward. I specially appreciate the fact that I could apply so many different skills I have learned during my years of study in the development of Segzoo, from optimization of software to project management. To conclude, the results obtained and the fact that they are already useful to its users show that the whole development and the whole project was successful in achieving its goals.

Glossary

annotation A division of a genome (or part of a genome) into segments assigned to a fixed set of labels. Unlike with a segmentation, the segments can be overlapping. 14, 18–21, 27, 43–45, 47, 50, 52

ChIP-seq Stands for chromatin immunoprecipitation sequencing, it is a powerful method for identifying genome-wide DNA binding sites for transcription factors and other proteins. [66]. 11, 20

chromatin A complex of macromolecules found in cells, consisting of DNA, protein, and RNA with the primary functions of compacting, reinforcing, preventing damage and controling replication for DNA. 6, 15

chromosome A structure of nucleic acids and protein found in the nucleus of most living cells, carrying genetic information in the form of genes. Human cells contain 23 pairs of chromosomes which is the total of the genetic information in the cell.. 45

enhancer Genomic region that influences transcription of a gene distant along the one-dimensional chromosome.. 80

exon Any part of a gene that will encode a part of the final mature RNA produced by that gene after introns have been removed by RNA splicing. In other words, is a coding part of a gene. 19, 20, 43, 47, 52

histone Class of protein that acts to package DNA into nucleosomes.. 83

histone modification A covalent post-translational modification (PTM) to histone proteins, which can alter gene expression [67]. 11, 20

intellectual property Inventions (whether or not patentable), technology, technical information, confidential information, concepts, ideas, etc. Intellectual property includes research works, institutional works and traditional academic works.. 36

intron The opposite of exons. Introns are the sections of a gene that don't contain coding information for RNA and are removed by RNA splicing. 19, 52

label Identifier for a pattern or cluster describing multiple regions of the genome. 11, 16, 18–20, 28, 54

lincRNA Short for Long intergenic noncoding RNA, a transcript longer than 200 nucleotides that is not translated into protein. 52, 54, 70, 74

nucleotide Organic molecules that form nucleic acids like DNA or RNA. There are 4 primary different nucleotides: adenine (A), cytosine (C), guanine (G), thymine(T). 10, 21, 43, 45–47, 50

protein coding DNA sequences that are transcribed into mRNA and in which the corresponding mRNA molecules are translated into a polypeptide chain (protein). 52, 54, 70, 74

recipe A set of documents including configuration files and scripts that suffice to specify a set of actions to execute and data to download automatically. 24, 29, 40, 41

segmentation A division of a genome (or part of a genome) into non-overlapping segments, each of which is assigned one of a fixed set of labels. Ideally, segments that share a common label are somehow similar to one another, and vice versa [21]. 6, 9–11, 13, 14, 16, 18–22, 35, 43, 44, 46–48, 51–54, 56, 58, 60, 61, 83

turnkey system A system that only requires one action by a user, like turning a key or pressing a button, to function correctly. 9, 58

Acronyms

API Application Programming Interface. 16, 24

BED Browser Extensible Data. 44, 46, 53, 80

CSV Comma Separated Values. 48

DBN dynamic Bayesian network. 10

GGD Go Get Data. 24, 27, 28, 40–44, 46, 61, 74, 75, 81

GMTK The Graphical Models Toolkit. 10, 11, 16, 22, 48, 51, 54, 56, 61, 71

GNU GNU's Not Unix!. 36

GPLv2 General Public License version 2. 36, 37

GTF General Transfer Format. 44, 45, 47, 48, 52, 53, 57, 61, 74, 75

IP Intellectual Property. 36

PMCRT The Princess Margaret Cancer Research Tower. 9, 27

PyPi Python Package Index. 29

rst reStructured Text. 8, 67

UCSC University of California, Santa Cruz. 14, 20

UHN University Health Network. 36, 38

UPC Universitat Politècnica de Catalunya. 36

References

[1] *Hoffman Lab.* URL: https://hoffmanlab.org/ (visited on 14/02/2018).

[2] Michael M. Hoffman et al. 'Unsupervised pattern discovery in human chromatin structure through genomic segmentation'. In: *Nature Methods* (2012). ISSN: 15487091. DOI: 10.1038/nmeth.1937.

[3] *Segway.* URL: https://segway.hoffmanlab.org/ (visited on 14/02/2018).

[4] The Encode and Project Consortium. 'A user's guide to the encyclopedia of DNA elements (ENCODE).' In: *PLoS biology* 9.4 (2011), e1001046. ISSN: 1545-7885. DOI: 10.1371/journal.pbio.1001046. URL: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3079585%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract.

[5] Jeff A. Bilmes and Chris Bartels. 'Graphical model architectures for speech recognition'. In: *IEEE Signal Processing Magazine* 22.5 (2005), pp. 89–100. ISSN: 10535888. DOI: 10.1109/MSP.2005.1511827.

[6] Nathan Day et al. 'Unsupervised segmentation of continuous genomic data'. In: *Bioinformatics* 23.11 (2007), pp. 1424–1426. ISSN: 13674803. DOI: 10.1093/bioinformatics/btm096.

[7] Michael M. Hoffman, Orion J. Buske and William Stafford Noble. 'The genomedata format for storing large-scale functional genomics data'. In: *Bioinformatics* (2010). ISSN: 13674803. DOI: 10.1093/bioinformatics/btq164.

[8] Jeff Bilmes and Geoffrey Zweig. 'The graphical models toolkit: An open source software system for speech and time-series processing'. In: *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings* 4 (2002), pp. 3916–3919. ISSN: 15206149. DOI: 10.1109/ICASSP.2002.5745513.

[9] A. P. Dempster, N. M. Laird and D. B. Rubin. 'Maximum Likelihood from Incomplete Data via the EM Algorithm'. In: *Journal of the Royal Statistical Society. Series B* 39.1 (1977), pp. 1–38. ISSN: 00359246. DOI: 10.2307/2984875. arXiv: 0710.5696v2. URL: http://www.jstor.org/stable/2984875.

[10] Andrew J. Viterbi. 'Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm'. In: *IEEE Transactions on Information Theory* 13.2 (1967), pp. 260–269. ISSN: 15579654. DOI: 10.1109/TIT.1967.1054010.

[11] Tony Kouzarides. *Chromatin Modifications and Their Function.* 2007. DOI: 10.1016/j.cell.2007.02.005. arXiv: NIHMS150003.

[12] Peter J. Park. *ChIP-seq: Advantages and challenges of a maturing technology.* 2009. DOI: 10.1038/nrg2641. arXiv: NIHMS150003.

[13] Jason Ernst and Manolis Kellis. 'Discovery and Characterization of Chromatin States for Systematic Annotation of the Human Genome'. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).* 2011. ISBN: 9783642200359. DOI: 10.1007/978-3-642-20036-6_6.

[14] Belinda Giardine et al. 'Galaxy: A platform for interactive large-scale genome analysis'. In: *Genome Research* 15.10 (2005), pp. 1451–1455. ISSN: 10889051. DOI: `10.1101/gr.4086505`.

[15] Aaron R. Quinlan and Ira M. Hall. 'BEDTools: A flexible suite of utilities for comparing genomic features'. In: *Bioinformatics* 26.6 (2010), pp. 841–842. ISSN: 13674803. DOI: `10.1093/bioinformatics/btq033`.

[16] Heng Li et al. 'The Sequence Alignment/Map format and SAMtools'. In: *Bioinformatics* 25.16 (2009), pp. 2078–2079. ISSN: 13674803. DOI: `10.1093/bioinformatics/btp352`. arXiv: `1006.1266v2`.

[17] Simon G Coetzee et al. 'rapid and reproducible chromatin state StateHub-StatePaintR: evaluation for custom genome annotation [version 1; referees: 1 approved with reservations]'. In: 11 (2018), pp. 214214–7. DOI: `10.12688/f1000research.13535.1`.

[18] Ricardo Wurmus et al. 'Reproducible genomics analysis pipelines with GNU Guix'. In: *bioRxiv* (2018), p. 298653. DOI: `10.1101/298653`.

[19] *PiGx*. URL: `http://bioinformatics.mdc-berlin.de/pigx/` (visited on 15/05/2018).

[20] Orion J. Buske et al. 'Exploratory analysis of genomic segmentations with Segtools'. In: *BMC Bioinformatics* (2011). ISSN: 14712105. DOI: `10.1186/1471-2105-12-415`.

[21] *Segtools*. URL: `https://hoffmanlab.org/proj/segtools/` (visited on 14/02/2018).

[22] W. J. Kent et al. 'The Human Genome Browser at UCSC'. In: *Genome Research* 12.6 (2002), pp. 996–1006. ISSN: 1088-9051. DOI: `10.1101/gr.229102`. URL: `http://www.genome.org/cgi/doi/10.1101/gr.229102`.

[23] Kate R. Rosenbloom et al. 'The UCSC Genome Browser database: 2015 update'. In: *Nucleic Acids Research* 43.D1 (2015), pp. D670–D681. ISSN: 13624962. DOI: `10.1093/nar/gku1177`.

[24] D. Karolchik. 'The UCSC Table Browser data retrieval tool'. In: *Nucleic Acids Research* 32.90001 (2004), pp. 493D–496. ISSN: 1362-4962. DOI: `10.1093/nar/gkh103`. URL: `https://academic.oup.com/nar/article-lookup/doi/10.1093/nar/gkh103`.

[25] Mehran Karimzadeh and Michael M Hoffman. 'Creating great documentation for bioinformatics software'. In: (2016).

[26] *Snakemake*. URL: `https://snakemake.readthedocs.io/` (visited on 01/03/2018).

[27] Johannes Köster and Sven Rahmann. 'Snakemake-a scalable bioinformatics workflow engine'. In: *Bioinformatics* 28.19 (2012), pp. 2520–2522. ISSN: 13674803. DOI: `10.1093/bioinformatics/bts480`.

[28] *Bpipe*. URL: `https://github.com/ssadedin/bpipe` (visited on 01/03/2018).

[29] *Go Get Data recipes*. URL: `https://github.com/gogetdata/ggd-recipes` (visited on 01/03/2018).

[30] *Anaconda*. URL: `https://anaconda.org/` (visited on 01/03/2018).

[31] *Conda.* URL: https://conda.io/ (visited on 10/04/2018).

[32] *GenomePy.* URL: https://github.com/simonvh/genomepy (visited on 01/03/2018).

[33] *Pandas.* URL: https://pandas.pydata.org/ (visited on 15/03/2018).

[34] *Seaborn.* URL: https://seaborn.pydata.org/ (visited on 25/03/2018).

[35] *Matplotlib.* URL: https://matplotlib.org/ (visited on 25/03/2018).

[36] *Bitbucket.* URL: https://bitbucket.org/ (visited on 20/02/2018).

[37] *Mercurial.* URL: https://www.mercurial-scm.org/ (visited on 20/02/2018).

[38] *GitHub,* URL: https://github.com/ (visited on 20/02/2018).

[39] *Git.* URL: https://git-scm.com/ (visited on 20/02/2018).

[40] *PyPI.* URL: https://pypi.org/ (visited on 10/04/2018).

[41] *Bioconda.* URL: https://bioconda.github.io/ (visited on 10/04/2018).

[42] *cnet.* URL: https://www.cnet.com (visited on 25/04/2018).

[43] *payscale.* URL: https://www.payscale.com (visited on 25/04/2018).

[44] *UHN Policies.* URL: http://www.uhn.ca/corporate/AboutUHN/Governance_Leadership/Policies (visited on 05/06/2018).

[45] *UHN Policy and Procedure Manual.* URL: http://www.uhn.ca/Corporate/For_Staff/New_Employees/Documents/Policies.pdf (visited on 05/06/2018).

[46] *GPLv2.* URL: https://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html (visited on 03/04/2018).

[47] William Stafford Noble. *A quick guide to organizing computational biology projects.* 2009. DOI: 10.1371/journal.pcbi.1000424.

[48] *Python.* URL: https://www.python.org/ (visited on 14/02/2018).

[49] *Go Get Data Anaconda.* URL: https://anaconda.org/ggd-alpha/ (visited on 01/03/2018).

[50] *Bioconda recipes.* URL: https://github.com/bioconda/bioconda-recipes (visited on 10/04/2018).

[51] *BED File Format.* URL: https://genome.ucsc.edu/FAQ/FAQformat.html (visited on 10/03/2018).

[52] *GTF File Format.* URL: https://useast.ensembl.org/info/website/upload/gff.html (visited on 10/03/2018).

[53] *FASTA Format.* URL: http://www.bioinformatics.nl/tools/crab_fasta.html (visited on 10/03/2018).

[54] *Segzoo Bitbucket.* URL: https://bitbucket.org/hoffmanlab/segzoo (visited on 18/06/2018).

[55] *Gene Biotypes.* URL: https://www.gencodegenes.org/gencode_biotypes.html (visited on 10/03/2018).

[56] *Jupyter.* URL: https://jupyter.org/ (visited on 10/03/2018).

[57] *IPython*. URL: `https://ipython.org/` (visited on 10/03/2018).

[58] *pybedtools*. URL: `https://daler.github.io/pybedtools/` (visited on 20/03/2018).

[59] *Python Packaging*. URL: `https://packaging.python.org/` (visited on 05/04/2018).

[60] *Segzoo PyPi*. URL: `https://pypi.org/project/segzoo/` (visited on 18/06/2018).

[61] *Python 2 to 3*. URL: `https://portingguide.readthedocs.io/` (visited on 15/04/2018).

[62] *PyCharm*. URL: `https://www.jetbrains.com/pycharm/` (visited on 20/02/2018).

[63] *Conda-build*. URL: `https://conda.io/docs/user-guide/tutorials/` (visited on 10/04/2018).

[64] *Conda Meta.yaml*. URL: `https://conda.io/docs/user-guide/tasks/build-packages/define-metadata.html` (visited on 15/05/2018).

[65] Robin Andersson et al. 'An atlas of active enhancers across human cell types and tissues'. In: *Nature* 507.7493 (2014), pp. 455–461. ISSN: 14764687. DOI: `10.1038/nature12787`. arXiv: `NIHMS150003`.

[66] *ChIP-seq*. URL: `https://www.illumina.com/techniques/sequencing/dna-sequencing/chip-seq.html` (visited on 20/02/2018).

[67] *Histone modifications*. URL: `https://www.whatisepigenetics.com/histone-modifications/` (visited on 20/02/2018).