



Quality comparison of published genomes: Definition of methods and techniques.

Abel Jiménez Mollà

Master's degree in Bioinformatics and Biostatistics
Bioinformatics genome quality

Guillem Ylla

David Merino Arranz

June 6, 2019

This work is licensed under a Creative Commons “Attribution-NonCommercial-ShareAlike 3.0 Unported” license.



Table 1: Thesis sheet

Thesis title:	Quality comparison of published genomes: Definition of methods and techniques.
Author:	Abel Jiménez Mollà
Consultant's name:	Guillem Ylla
Teacher responsible:	David Merino Arranz
Deadline (mm/aaaa):	06/2019
University degree:	Master's degree in Bioinformatics and Biostatistics
Thesis area:	Bioinformatics
Language:	English
Keywords	Genomics, Genomes quality, Methods, Techniques, Analysis, Genome assembly, Genome annotation

Abstract

Based on recent versions of the genomes that are publicly available within the scientific community, we have decided to investigate the quality of such existing genomes. The aim of this analysis is to evaluate current methods and techniques to measure and compare the quality of assemblies and annotations, considering genomes from different origins and evolutionary lineages (both from model and non-model organisms).

The aim of this study is to further evaluate the consequences of the use of a poor-quality genome and avoid inaccuracies which are intrinsic in order to minimize the range of error in research. Moreover, we want to assess if by using genomes with different quality standards in research, we can optimize results. The above-mentioned questions justify the development for new methods and techniques which will help us control the quality of genomes and improve investigation in the future. The main objective of this project is the identification of varied techniques using a range of existing software technologies to automate and make the quality control of genomes more accessible for researchers, as well as the possibility to compare genome quality as a result.

Keywords— Genomics, Genomes quality, Methods, Techniques, Analysis, Genome assembly, Genome annotation

Contents

1	Introduction	6
1.1	Project Context and justification	6
1.1.1	General description	6
1.1.2	Justification of the Master's thesis	6
1.2	Project Objectives	6
1.2.1	General objectives	6
1.2.2	Specific objectives	6
1.3	Approach and Methods	7
1.3.1	Software	8
1.3.2	Hardware	8
1.4	Project Planning	8
1.4.1	Tasks	8
1.4.2	Calendar	8
1.4.3	Milestones	9
1.4.4	Risk analysis	11
1.4.5	Contingency plan	11
1.5	Summary of Products Obtained	12
1.6	Description of the Report Chapters	12
2	Genome quality tools	12
2.1	Genome quality characteristics	12
2.2	Existing tools	13
2.3	Understanding the software	14
2.3.1	Installation	14
2.3.1.1	Assembly-stats	14
2.3.1.2	Dogma	14
2.3.1.3	Busco	15
2.3.2	First run	15
2.3.2.1	Assembly-stats	15
2.3.2.2	Dogma	16
2.3.2.3	Busco	18
2.4	Development of alternatives	20
3	Results	21
3.1	Menu	21
3.2	Assembly-stats	22
3.3	Dogma	23
3.4	Busco	24
4	Conclusions	25
5	List of terms	26
	Acronyms	26
	Glossary	26
6	Bibliography	27

7 Annexes	28
7.1 Load requirements	28
7.2 Create Shiny app	28
7.2.1 UI	28
7.2.2 Server	30
7.3 Important functions	30
7.3.1 StartMainProcess	30
7.3.2 ReadFile	33
7.3.3 Init	34
7.3.4 StartProcess	36

List of Figures

1	Original Planning Gantt chart	9
2	Effective Planning Gantt chart	9
3	Assembly-stats command output example	16
4	pfam`scan.pl command output example	18
5	Dogma command output example	18
6	Busco command output example	20
7	Genome quality tool overview	22
8	Genome quality tool Assembly-stats overview	23
9	Genome quality tool Dogma overview	23
10	Genome quality tool Busco overview	25

List of Tables

1	Thesis sheet	2
2	Proposed Milestones Deadlines	10
3	Effective Milestones Deadlines	10
4	Software technical comparison	13
5	Software comparison from a biological point of view	14
6	Comparison of the main programming languages	21

List of Listings

1	Assembly-stats install steps	14
2	Assembly-stats command help	15
3	Assembly-stats command example	15
4	Dogma command help	16
5	pfam`scan.pl command help	17
6	pfam`scan.pl command	18
7	Dogma command	18
8	Busco command help Mandatory	19
9	Busco command help Optional	19
10	Busco command Example	20
11	Source Code - Load requirements	28
12	Source Code - Create Shiny app	28
13	Source Code - UI	28
14	Source Code - Server	30
15	Source Code - StartMainProcess	31
16	Source Code - ReadFile`assembly	33
17	Source Code - ReadFile`busco	33
18	Source Code - ReadFile`dogma	33
19	Source Code - init`assembly	34
20	Source Code - init`busco	34
21	Source Code - init`dogma	35
22	Source Code - StartProcess`assembly	36
23	Source Code - StartProcess`busco	37
24	Source Code - StartProcess`dogma	37

1 Introduction

1.1 Project Context and justification

1.1.1 General description

A reference genome also known as a reference assembly is a comprehensive, integrated, non-redundant, well-annotated set of reference sequences including genomic, transcript, and protein [9]. It is a database, assembled by scientists to obtain a representative example of a species set of genes. Once we know what the reference genome is, we can introduce the term DNA annotation. It is the process of identifying the locations of the genes and all the coding regions in a genome and determining what function those genes have [1].

By definition the quality of the reference genomes and also the quality of its annotation might vary greatly, due to technical limitation. Current trends emphasize the need for improving the quality of those techniques which determine the quality of such genomes. Our intention is to create a tool that helps us to evaluate the quality of different genome versions. We intend to combine existing tools, the methods, and techniques to measure the quality of assemblies and their annotations. After having carried out a thorough look at existing genome quality tools, we have considered an in-depth analysis of three specific tools: Busco, assembly-stats and Dogma, considering these are the most up-to-date tools regarding software maintenance. In order to test that the tool we propose works correctly, we will use data from genomes of different origins and evolutionary lineages (both from model and non-model organisms), specifically eukaryotes.

1.1.2 Justification of the Master's thesis

Our aim is to study the quality of different genomes sequences and also the quality of its annotations in an attempt to rise a warning to researchers to make them aware of the mistakes that and incorrect genome might induce in their research. These mistakes are not necessarily inherent to the research processes but rather related to the poor quality of genomes itself used as the starting point for such research. This is the reason why we want to evaluate if through the use of the tool we have developed, we can obtain higher quality results that can be used during the investigation, obtaining much more precise and relevant results. This clearly justifies the need for development of new methods and techniques which will certainly help the scientific community control the quality of genomes and improve research in the future. The genome quality tool developed will determine the quality of the genome sequence we opt to use and its annotations. Therefore, the researcher will know in advance which problems, errors or bias might face in his analysis mainly due to the genome quality.

1.2 Project Objectives

The objectives that are to be achieved with the Master's Final Project are the ones which follow. We believe it deemed necessary to split them into general and specific objectives.

1.2.1 General objectives

The general objectives of this thesis are:

1. Analyze methods and techniques to measure and compare the quality of assemblies and annotations of genomes.
2. Develop a tool which researchers can use to verify the quality of the genome sequence they are working with.

1.2.2 Specific objectives

The specific objectives have been organized into two blocks. The first block is an analysis phase which will serve as the basis of the design and implementation of the tool we suggest. The second stage is that of

the actual development of the tool itself. Obviously, both phases intertwine, and they may well happen simultaneously as, certainly, improvements will have to be made on the development of the tool itself based on the findings of the analysis. The rationale behind this being simply a logical way of grouping such objectives.

1. Analysis phase
 - (a) Perform a market survey to analyze software programs currently on the market.
 - (b) Study the characteristics of the software programs obtained in the survey.
 - (c) Identify the characteristics of the genomes to be studied, the qualitative variables as the quantitative variables.
 - (d) Decision making, is it convenient to implement a new method or technique? Or, on the contrary, does the existing software cover all the needs?
2. Development phase
 - (a) Analysis of the most used technologies and with greater distribution among the community.
 - (b) Decision making, making a completely new software or using existing software and combining it to generate a more robust solution.
 - (c) Implement the software, in which the different analyzes performed previously are applied.
 - (d) Delivery the software to the community.

1.3 Approach and Methods

Several strategies can be adopted to carry out this project. One can focus the development by using a much more traditional methodology with the following phases: strategic analysis, specification of requirements, design and implementation. The traditional methodology is characterized by a strong analytical phase in which most of the time is spent on the actual analysis of what needs to be developed [11]. This is not a very flexible kind of methodology which does not serve the purpose of this project.

On the other hand, the possible changes in the requirements of the design of the program during its development or any special requests from the community make us opt for agile methodology techniques. This is certainly more flexible and adaptable to changes [11]. A functional prototype needs to be created every two weeks in order to increase the interaction between the end user and the developer, thus improving the quality of the software solution provided. The end user is presented with each of the prototypes fortnightly and encouraged to test it and suggest improvements, detect misfunctions, propose new functionalities of the tool which are deemed necessary from this end user perspective. The developer will adapt those changes in order to present the new prototype which includes those updated functionalities along with any other improvements.

During the analysis phase that will not last longer than two weeks, the utmost need is to perform a market survey to analyze the software programs that are currently on the market. The main interest in this first step is to determine each of the characteristics of the genome in which each of this software programs focuses on. This will facilitate identify both the qualitative variables and the quantitative variables and provide a preview on the development of the software program to be developed.

During the development phase, as it has already been anticipated, the delivery of a functional prototype of the application every two weeks is key, increasing the functionalities in each of the prototypes and applying the corrections and changes proposed by the end user.

The software and hardware required and used to perform the correct development stages will be detailed in the next sections.

1.3.1 Software

In this section the software characteristics are detailed to have an overview of all the software components.

- The final application is developed in Shiny.[10]
- R is the main software used to manipulate the data.
- Third-party applications were developed using different software such as Python, Perl or C++.
- Linux terminal is the connection point between the R Shiny app and the Third-party applications.
- Virtualbox is used to group and run all the software that is needed.

1.3.2 Hardware

The application is running in a Virtualbox machine with the following characteristics:

- **Processor:** 2 virtual CPU's.
- **Memory:** 4 GB of RAM.
- **Storage:** Dynamic storage.
- **Operating System:** Ubuntu 16.04 LTS (Xenial Xerus) 32 bits.

1.4 Project Planning

In this section, tasks and its deadlines variations, Gantt charts and project milestones will be exposed. The task section lists the various tasks in which the objectives have been divided. The Gantt charts is used to specify a date for each of the tasks and project milestones. It also includes an analysis of the risks that could occur initially and those that have finally happened.

1.4.1 Tasks

Among the identified tasks we have also included those required by university criteria. Although not directly related to the project itself we consider them indispensable for the correct resolution of the project. Therefore tasks have been identified and divided into the following ones:

- Initial analysis (1a, 1b, 1c , 1d)
- Technological analysis (2a , 2b)
- Prototype design (2c)
- Master's thesis redaction conclusion. (2d)
- Presentation development.
- Public defense.

1.4.2 Calendar

The thesis was planned using a temporary structure based on days per month. It has taken into account the work as a freelancer of the author. The calendar indicates the days on which to work on each task, without entering into detail on the hours as this does fully depend on the author. To this purpose we used Teamgantt[©][13] implementing it by adding the milestones to the chart. This will be explained in more detail in the next section.

As we can see in the figure 1 and figure 2, changes have been made to the timeline. The delivery of the second prototype has been delayed of two weeks due to problems with the hosting server. This happened because of the occurrence of 4 and 5 that were identified in the initial phase. For this reason the development of the second prototype overlaps with the delivery of the third prototype. At this point we have decided to reduce the number of prototypes to be delivered and continue with the already established plan.

Figure 1 and figure 2 show the differences between the original planning and the effective one:

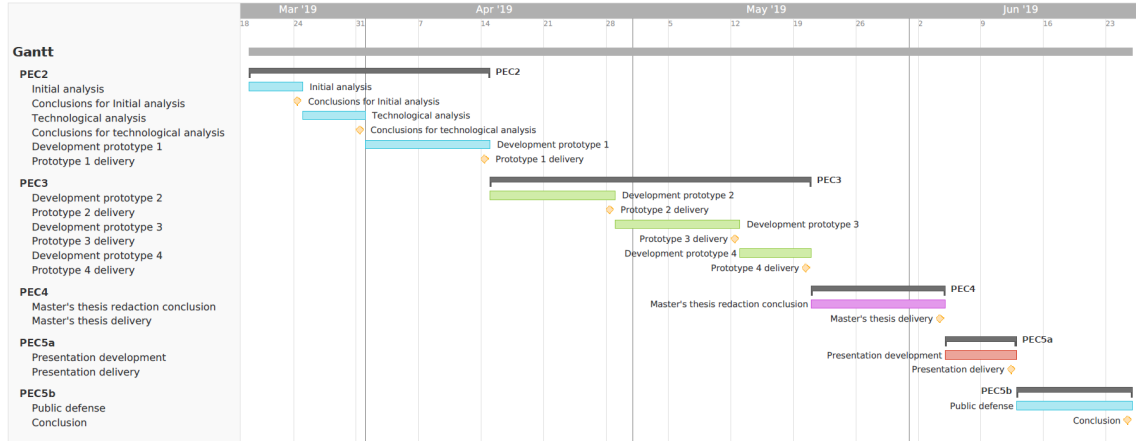


Figure 1: Original Planning Gantt chart

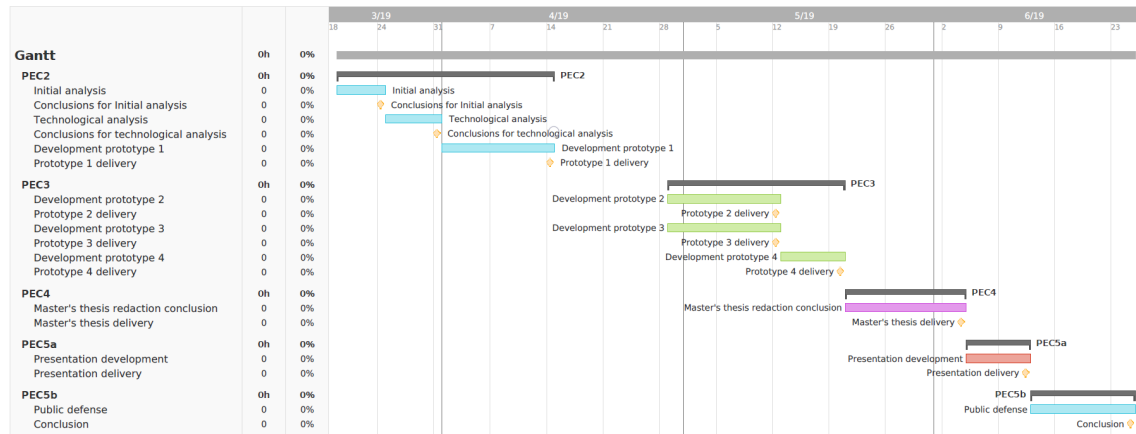


Figure 2: Effective Planning Gantt chart

1.4.3 Milestones

As already mentioned in the calendar, the planning has changed and this has also affected the milestones. In this section, we will compare the milestones initially proposed with those that have finally been carried out.

As we can see in table 2 within milestones that were initially proposed we had two initial analysis that guided the development of each of the 4 prototypes and three milestones that were related with mandatory steps that have to be performed for the successful outcome of this thesis.

The difference with the proposed milestones and the effective milestones is mainly due to the elimination of the delivery of one of the prototypes more specifically the prototype number two because it overlaps with

the delivery of the third prototype. We can observe the changes in detail in the following tables: table 2 and table 3 .

Table 2: Proposed Milestones Deadlines

Milestone	PEC number	Deadlines
Conclusions for Initial analysis	PEC2	24/03/2019
Conclusions for Technological analysis	PEC2	31/03/2019
Proposed Prototype 1 delivery	PEC2	14/04/2019
Proposed Prototype 2 delivery	PEC3	28/04/2019
Proposed Prototype 3 delivery	PEC3	12/05/2019
Proposed Prototype 4 delivery	PEC3	20/05/2019
Master's thesis delivery	PEC4	04/06/2019
Presentation delivery	PEC5a	12/06/2019
Public defense	PEC5b	25/06/2019

- Proposed Prototype 1 delivery includes:
 - Implementation of the software architecture;
 - Creation of a draft project.
- Proposed Prototype 2 delivery includes:
 - Implementation of the modifications reported by the user in the previous step;
 - Creation of the first user interface.
- Proposed Prototype 3 delivery includes:
 - Implementation of the modifications reported by the user in the previous step;
 - Interpretation of the data from files and databases;
 - Presentation of the first results and calculations based on the input genome.
- Proposed Prototype 4 delivery includes:
 - Implementation of the modifications reported by the user in the previous step;
 - Release of the candidate version.

Table 3: Effective Milestones Deadlines

Milestone	PEC number	Deadlines
Conclusions for Initial analysis	PEC2	24/03/2019
Conclusions for Technological analysis	PEC2	31/03/2019
Effective Prototype 1 delivery	PEC2	14/04/2019
Effective Prototype 3 delivery	PEC3	12/05/2019
Effective Prototype 4 delivery	PEC3	20/05/2019
Master's thesis delivery	PEC4	04/06/2019
Presentation delivery	PEC5a	12/06/2019
Public defense	PEC5b	25/06/2019

- Effective Prototype 1 delivery includes:

- Implementation software architecture;
- Creation of a draft project.
- Effective Prototype 3 delivery includes:
 - Implementation of the modifications reported by the user in the previous step;
 - Creation of the first user interface;
 - Interpretation of data from files and databases.
- Effective Prototype 4 delivery includes:
 - Implementation of the modifications reported by the user in the previous step;
 - Presentation of the first results and calculations based on the input genome;
 - Release of the candidate version.

1.4.4 Risk analysis

As already mentioned, during the carrying out of the project there are certain factors that could occur and negatively affect the planning and execution of the project. These may lead the master's thesis not to reach a good outcome. The factors that were identified are the following ones:

1. Time for developing is limited and the project can not be done on time;
2. Software licenses to be used are not public;
3. Not being able to identify qualitative and quantitative variables;
4. Hardware limitation on the tools that we are using to develop the software;
5. Economic difficulties: software, hardware, hosting server;
6. Size of the genomes and their availability;
7. Time for testing the prototypes can be short due to the duration of the cycles.

Some of the risks anticipated in the initial analysis eventually occurred. We have adapted the delivery of the prototypes because we have found problems due to the size of genomes(6) and the databases used by third-party applications. Moreover, the free/student versions of the cloud appeared to be not enough to run the project thus resulting in high costs for running the application(5).

1.4.5 Contingency plan

In this section we will detail how we can act in each of the risk cases detected in the risk analysis section. Obtaining an alternative in the case of having problems during the development.

1. Few things can be done if time is not enough, if this happens I would try again next year. (Risk.1)
2. Luckily there are many free software alternatives, but if you still do not satisfy our needs we can always write to the owner of the software and explain that its use will be for academic purposes. (Risk.2)
3. We are lucky not to be alone in this adventure, so if we are not able to identify the qualitative and quantitative variables we can always ask for help from teachers and colleagues. (Risk.3)
4. If we are faced with any hardware limitation, we can always access a university cluster or use one of the free versions of super computers offered by large companies. (Risk.4)
5. If we can not pay the cost of development, we can apply for grants or scholarships. (Risk.5)
6. We can always access a supercomputer from the university or use one of the free versions of super computers offered by large companies. (Risk.6)
7. We can reduce the number of prototypes and increase their duration to extend the test time. (Risk.7)

1.5 Summary of Products Obtained

This section shows a list of tangible items of what is expected obtain at the end of its development.

- List with the main characteristics of the genome that we intend to study.
- Market survey to analyze genome quality analysis software that currently exists in the market.
- Table with the comparison of the main characteristics of the software detected in the market analysis.
- Table comparing the existing alternatives to develop our application.
- Fully functional application for the analysis of genome quality through files in FASTA format.
- The document of the master thesis itself.
- A Power point presentation summarizing the most important features of the application and a video with its the operation.

1.6 Description of the Report Chapters

Below are described the theoretical and practical chapters related with this project, each of them will be detailed in the following sections:

- **Genome quality characteristics:** the aim of the chapter contextualizes the different characteristics genomes that can be used to measure their quality;
- **Existing tools:** the purpose is to make a market analysis of the software of the currently existing software, to further make a comparison of the qualities in order to be able to select the best software options to control the quality of the genome;
- **Understanding the software:** once the market analysis is done and the best software is selected, it is indispensable to familiarize with it. This chapter explains the peculiarities of each of the chosen software and shows how to use it;
- **Development alternatives:** an important step in the development of our application to measure the quality of the software is to show the results obtained in a clean and clear way. To do so, this chapter analyzes the available options in the market, to select the best technology in order to carry out our application.

2 Genome quality tools

2.1 Genome quality characteristics

These are some of the characteristics of the genome that we can consider to evaluate the quality of the genome.

- **Genome size:** this is an important value to bring to the sequencing facility, as the genome size will greatly influence the amount of data that needs to be ordered. To assemble a genome, a certain amount of sequences (also called reads) is needed. A number of >60x sequence depth is often mentioned[2]. This means that the number of total nucleotides in the sequencing reads used for the genome assembly need to be at least 60 times the number of nucleotides in the genome.[3] for this reason the genome size is a characteristic to consider during the genome quality control.
- **N50 statistics:** it is a measure to describe the quality of assembled genomes that are fragmented in contigs or scaffolds of different lengths. To get the N50 contig length, all contigs of a genome must be sorted by their length, then we must look for the base that is in the center at 50% of the total genome length, at this point we can get the contig size to which this base belongs and we will have the N50 contig length. There are also similar statistics like N70 or N90 that also used during the genome quality control. They are the same as N50 but instead of 50%, N70 corresponds to the 70% and N90 corresponds to the 90% [7].

- **Benchmarking Universal Single-Copy Orthologs (OrthoDB)** : the term ortholog refers to any of two or more homologous gene sequences found in different species related by linear descent, [5] namely, genes present in different species coming from the same ancestral gene. We can use this characteristic to order and quantify the genome assemblies by using a database called OrthoDB [14]. From its conception, OrthoDB promoted delineation of orthologs at varying resolution by explicitly referring to the hierarchy of species radiations. The current release provides comprehensive coverage of animals, fungi and bacteria. This provides us with a broad database where we can consult and assess the similarities of our genome assemblies.
- **Conserved protein domains:** another way to control the quality of our genome assemblies is by consulting the proteins conserved during the evolution of the species also known as Conserved Domain Arrangements (CDAs) [6].

2.2 Existing tools

At this point we will analyze the different software alternatives that are currently available for analyzing the qualities of genomes and genomes annotations. In particular, this is important in order to see if they are good enough to measure the quality of the software or on the contrary to test whether they do not meet the minimum requirements to measure the quality of the genome. We will analyze in detail the characteristics of these applications in order to know them and have an objective idea of which are the best options.

From a more technical point of view we could study many different features but in this case we have decided to focus on the following ones and proceed to explain each of them with more detail and they are shown in table 4.

- **Maintenance:** the questions will be: does the software have support from its creators? Is there an open and participatory community that brings updates and improvements to the software? To answer them we will use two possible values: supported or obsolete.
- **Platform:** the aim would be to understand whether the platforms support the software, if it is a cross-platform and if there is a version for each of the different platforms
- **Language:** in terms of efficiency, some programming languages are better than others,[4] but this needs a separate study to see which one is better. At an informative level we show the programming languages in which each software is written.
- **Version:** this column refers to the type of CPU in which the software can be executed. 32 or 64 bits are the most extended types of CPU. In most cases with a version 64 we can obtain a higher performance.[8]

Table 4: Software technical comparison

Software	maintenance	platform	language	version
BBMap	Supported	Unix	Python	32-bit/64-bit
Dogma	Supported	cross-platform	Perl	32-bit/64-bit
Assembly-stats	Supported	cross-platform	C++	32-bit/64-bit
Busco	Supported	Unix	Python	32-bit/64-bit
Quast	Supported	Unix	Python	32-bit/64-bit
Cegma	Obsolete	Unix	Perl	32-bit/64-bit

Focusing on the biological point of view we will discuss some more features, which are displayed in table 5. These can help us to choose the most interesting software for the development of the final solution.

- **Genome:** when deciding which software to use it has to be considered whether the software supports a Genome assembly file.

- **Transcriptome:** support for a transcript set (DNA nucleotide sequences).
- **Proteome:** it has to be considered whether the analyzed software supports a gene set (protein amino acid sequences).
- **Input:** the last column displays the type of input file that is supported by each program, in this case all of them support FASTA files.

Table 5: Software comparison from a biological point of view

Software	genome	transcriptome	proteome	input
BMap	TRUE	TRUE	FALSE	FASTA
Dogma	FALSE	TRUE	TRUE	FASTA
Assembly-stats	TRUE	FALSE	FALSE	FASTA
Busco	TRUE	TRUE	TRUE	FASTA
Quast	TRUE	TRUE	TRUE	FASTA
Cegma	TRUE	FALSE	TRUE	FASTA

These tables will help us to make a decision about which software to use and what type of software will be most useful when measuring the quality of the genome.

2.3 Understanding the software

After having obtained an overview of the software, we can decide which software is more convenient for the development of the application. At this point it is better to carry out tests with the software and learn how they work, how they are installed and know their advantages and disadvantages.

2.3.1 Installation

In this section the installation process for each of the programs is detailed.

2.3.1.1 Assembly-stats

Assembly-stats is a program written in c++ and does not have a pre-compiled version of the software, so we must compile it ourselves. To install the software we have to download the source code of the official Github repository and then execute the commands as shown in code 1.

Code 1: Assembly-stats install steps

```

1  mkdir build
2  cd build
3  cmake ..
4  make
5  make test
6  make install

```

2.3.1.2 Dogma

Dogma is written in Python language and therefore it is not necessary to install it, since Python is an interpreted programming language. Its characteristic is that it has dependencies: this means that Dogma needs other programs to work correctly, such as RADIANT or pfam'scan.pl. Considering that RADIANT

is in beta phase, its use is not recommended. On the other hand pfam'scan.pl is still supported by its developers.

For this reason we have chosen pfam'scan.pl to be able to execute Dogma normally. pfam'scan.pl is written in Perl as Python is a language interpreted for that reason we do not need to install anything. As long as we have Python and Perl installed on our computer they will be enough to run the two programs.

2.3.1.3 Busco

Similarly to Dogma, Busco is a program written in Python and therefore does not need an installation. It has more dependencies than Dogma because it needs three more programs to function correctly. These programs are: NCBI BLAST +, HMMER and Augustus. It is a bit more complicated to configure and install due to its dependencies, that is why Busco itself offers us a pre-installed virtual machine with everything necessary to start it up.

We have chosen this option and we have used it as a base to install the other programs, thus obtaining a closed system with everything necessary to operate. This gives us the option of transporting the entire system to a server if it is necessary to deploy the application to make it accessible to the whole world.

2.3.2 First run

Once the software is installed we will proceed to test it, in this section we will explain the necessary commands to operate each of the programs and we will cover in detail its most important parameters.

2.3.2.1 Assembly-stats

Assembly-stats has only 5 parameters, the 4 that we describe in Code 2 and the fifth argument for the sequence in .FASTA format.

Code 2: Assembly-stats command help

```
1 assembly-stats
2 usage: stats [options] <list of fasta/q files>
3
4 Reports sequence length statistics from fasta and/or fastq files
5
6 options:
7 -l <int>
8     Minimum length cutoff for each sequence.
9     Sequences shorter than the cutoff will be ignored [1]
10 -s
11     Print 'grep friendly' output
12 -t
13     Print tab-delimited output
14 -u
15     Print tab-delimited output with no header line
```

The code section 3 and the figure 3 show an example of using Assembly-stats and its respective console output. These will help us understand how it works and provide us with the first visible results.

Code 3: Assembly-stats command example

```
1 assembly-stats -t test_files/fasta_unittest.fasta
```

```
osboxes@osboxes: ~/ASSEMBLY-STATS
File Edit View Search Terminal Help
osboxes@osboxes:~/ASSEMBLY-STATS$ assembly-stats -t test_files/fasta_unittest.fasta
filename      total_length  number  mean_length  longest  shortest  N_count  Gaps  N50  N50n
  N70      N70n      N90      N90n
test_files/fasta_unittest.fasta 1050    2      525.00  650    400    0      0    650  1    400
osboxes@osboxes:~/ASSEMBLY-STATS$
```

Figure 3: Assembly-stats command output example

2.3.2.2 Dogma

Dogma is the only one of the 3 chosen programs that needs a previous command to function. So in this section we will detail the parameters of Dogma and the parameters of pfam`scan.pl since these are necessary for the correct functioning of Dogma.

Code 4: Dogma command help

```
1 -h, --help          show this help message and exit
2 -a ANNOTATION_FILE, --annotation_file ANNOTATION_FILE
3                   Annotation file of the transcriptome to be quality
4                   checked asRADIANT or PfamScan output.
5 -i SEQUENCE_FILE, --initial_radiant_run SEQUENCE_FILE
6                   The transcriptome file (in fasta format) that should
7                   be used for an initial run of RADIANT (domain
8                   annotation) and subsequently analyzed with DOGMA.
9 -r REFERENCE_TRANSCRIPTOMES, --reference_transcriptomes REFERENCE_TRANSCRIPTOMES
10                  A directory that contains annotation files of selected
11                  core species (*.rad for RADIANT annotated files and
12                  *.pfsc for pfam scan annotated files). Used to
13                  construct the core set with conserved domain
14                  arrangements. If omitted, the script looks for default
15                  values stored in the "pfamXX/reference-
```

```

16         sets/eukaryotes" directory. Valid values for analysis
17         with the default core sets are "eukaryotes",
18         "mammals", "insects", "bacteria" and "archaea"
19         (without quotes)
20 -o OUTFILE, --outfile OUTFILE
21         Summary will be saved in a file with the given name
22         (and path), instead of printed in the console.
23 -s CDA_SIZE, --cda_size CDA_SIZE
24         Specifies up to which size subsets of CDAs should be
25         considered (default=3; A-B-C-D --> A-B-C, A-B-D, B-C-D
26         etc.).
27 -m PFAM, --pfam PFAM The version number of the pfam database that should be
28         used (Default is 32).
29 -d DATABASE, --database DATABASE
30         If the RADIANT database is not located in the RADIANT
31         directory, please specify path and name of the
32         database. (Just necessary for -i option)
33 -cov COVERAGE, --coverage COVERAGE
34         Specifies how much of a domain has to be annotated to
35         count as a partial domain. Default=0.5 This would mean
36         if less than 50% of the domain is annotated it is
37         considered a partial domain. The partial domain
38         analysis is just available with PfamScan annotations.

```

Code 5: pfam'scan.pl command help

```

1  -h                : show this help
2  -outfile <file>  : output file, otherwise send to STDOUT
3  -clan_overlap    : show overlapping hits within clan member families (applies to Pfam-A families only)
4  -align          : show the HMM-sequence alignment for each match
5  -e_seq <n>      : specify hmmscan evalue sequence cutoff for Pfam-A searches (default Pfam defined)
6  -e_dom <n>      : specify hmmscan evalue domain cutoff for Pfam-A searches (default Pfam defined)
7  -b_seq <n>      : specify hmmscan bit score sequence cutoff for Pfam-A searches (default Pfam defined)
8  -b_dom <n>      : specify hmmscan bit score domain cutoff for Pfam-A searches (default Pfam defined)
9  -as             : predict active site residues for Pfam-A matches
10 -json [pretty]  : write results in JSON format. If the optional value "pretty" is given,
11                  the JSON output will be formatted using the "pretty" option in the JSON
12                  module
13 -cpu <n>         : number of parallel CPU workers to use for multithreads (default all)
14 -translate [mode] : treat sequence as DNA and perform six-frame translation before searching. If the
15                  optional value "mode" is given it must be either "all", to translate everything
16                  and produce no individual ORFs, or "orf", to report only ORFs with length greater
17                  than 20. If "-translate" is used without a "mode" value, the default is to
18                  report ORFs (default no translation)

```

Below are some examples of how to run each of the two softwares together with their respective outputs.

Code 6: pfam`scan.pl command

```
1 perl pfam_scan.pl -fasta test_data/target.fa -dir Bio/Pfam/ -outfile TFM_outputs/pfam_out_TFM
```

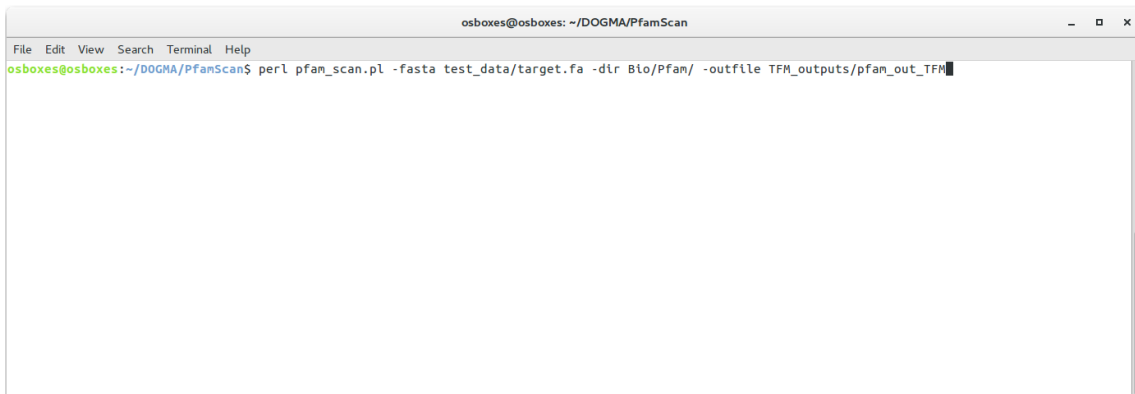


Figure 4: pfam`scan.pl command output example

Code 7: Dogma command

```
1 python dogma.py transcriptome -a PfamScan/TFM_outputs/pfam_out_TFM -o TFM_outputs/Dogma_out_TFM
```



Figure 5: Dogma command output example

2.3.2.3 Busco

Busco is the program with the most parameters of all those that we have analyzed. To work, you only need 4 parameters that are mandatory but it also includes many optional parameters that help us improve our results. In Code 8 and 9 we detail all BUSCO parameters, both optional and mandatory.

Code 8: Busco command help Mandatory

```

1  -i FASTA FILE, --in FASTA FILE
2          Input sequence file in FASTA format. Can be an assembled genome or transcriptome (DNA),
3          or protein sequences from an annotated gene set.
4  -o OUTPUT, --out OUTPUT
5          Give your analysis run a recognisable short name. Output folders and files will be
6          labelled with this name. WARNING: do not provide a path
7  -l LINEAGE, --lineage_path LINEAGE
8          Specify location of the BUSCO lineage data to be used.
9          Visit http://busco.ezlab.org for available lineages.
10 -m MODE, --mode MODE Specify which BUSCO analysis mode to run.
11          There are three valid modes:
12          - geno or genome, for genome assemblies (DNA)
13          - tran or transcriptome, for transcriptome assemblies (DNA)
14          - prot or proteins, for annotated gene sets (protein)

```

Code 9: Busco command help Optional

```

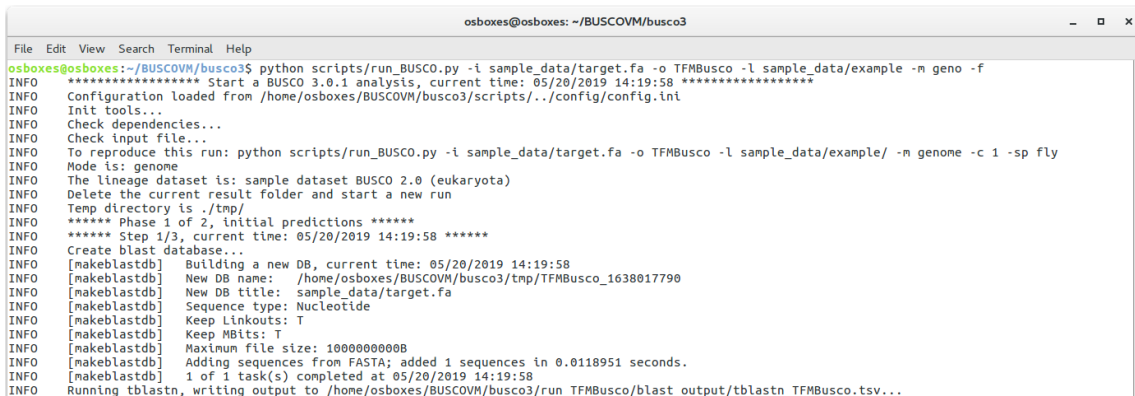
1  -c N, --cpu N          Specify the number (N=integer) of threads/cores to use.
2  -e N, --evaluate N    E-value cutoff for BLAST searches. Allowed formats, 0.001 or 1e-03 (Default: 1e-03)
3  -f, --force           Force rewriting of existing files. Must be used when output files with the provided
4  name already exist.
5  -r, --restart         Restart an uncompleted run. Not available for the protein mode
6  -sp SPECIES, --species SPECIES
7          Name of existing Augustus species gene finding parameters. See Augustus
8          documentation for available options.
9  --augustus_parameters AUGUSTUS_PARAMETERS
10         Additional parameters for the fine-tuning of Augustus run. For the species, do not
11         use this option.
12         Use single quotes as follow: '--param1=1 --param2=2', see Augustus documentation
13         for available options.
14 -t PATH, --tmp_path PATH
15         Where to store temporary files (Default: ./tmp/)
16 --limit REGION_LIMIT  How many candidate regions (contig or transcript) to consider per BUSCO (default: 3)
17 --long               Optimization mode Augustus self-training (Default: Off) adds considerably to the run time,
18 but can improve results for some non-model organisms
19 -q, --quiet          Disable the info logs, displays only errors
20 -z, --tarzip         Tarzip the output folders likely to contain thousands of files
21 --blast_single_core Force tblastn to run on a single core and ignore the --cpu argument for this step only.
22 Useful if inconsistencies when using multiple threads are noticed
23 -v, --version        Show this version and exit
24 -h, --help          Show this help message and exit

```

A simple example to understand the code, it would be 10 while the output it produces is shown in 6.

Code 10: Busco command Example

```
1 python scripts/run_BUSCO.py -i sample_data/target.fa -o TFMBusco -l sample_data/example -m geno -f
```



```
osboxes@osboxes: ~/BUSCOVM/busco3
File Edit View Search Terminal Help
osboxes@osboxes:~/BUSCOVM/busco3$ python scripts/run_BUSCO.py -i sample_data/target.fa -o TFMBusco -l sample_data/example -m geno -f
INFO ***** Start a BUSCO 3.0.1 analysis, current time: 05/20/2019 14:19:58 *****
INFO Configuration loaded from /home/osboxes/BUSCOVM/busco3/scripts/./config/config.ini
INFO Init tools...
INFO Check dependencies...
INFO Check input file...
INFO To reproduce this run: python scripts/run_BUSCO.py -i sample_data/target.fa -o TFMBusco -l sample_data/example/ -m genome -c 1 -sp fly
INFO Mode is: genome
INFO The lineage dataset is: sample dataset BUSCO 2.0 (eukaryota)
INFO Delete the current result folder and start a new run
INFO Temp directory is ./tmp/
INFO ***** Phase 1 of 2, initial predictions *****
INFO ***** Step 1/3, current time: 05/20/2019 14:19:58 *****
INFO Create blast database...
INFO [makeblastdb] Building a new DB, current time: 05/20/2019 14:19:58
INFO [makeblastdb] New DB name: /home/osboxes/BUSCOVM/busco3/tmp/TFMBusco_1638017790
INFO [makeblastdb] New DB title: sample_data/target.fa
INFO [makeblastdb] Sequence type: Nucleotide
INFO [makeblastdb] Keep Linkouts: T
INFO [makeblastdb] Keep MBits: T
INFO [makeblastdb] Maximum file size: 1000000000B
INFO [makeblastdb] Adding sequences from FASTA; added 1 sequences in 0.0118951 seconds.
INFO [makeblastdb] 1 of 1 task(s) completed at 05/20/2019 14:19:58
INFO Running tblastn, writing output to /home/osboxes/BUSCOVM/busco3/run_TFMBusco/blast_output/tblastn_TFMBusco.tsv...
```

Figure 6: Busco command output example

2.4 Development of alternatives

In this section we will analyze the main programming languages for the development of our application. We will solve mainly technical issues about the different programming languages available. The analysis of the advantages and disadvantages is important to obtain a general vision of which of them suits us more.

- **Biological packages:** the programming language should support specific packages focused on the biological world;
- **Platform:** it has to be considered whether the platforms support the software, if it is a cross-platform and if there is a version for each of the different platforms;
- **Suitability for statistical calculations:** it has to be considered whether the software is specifically designed or not for statistical operations;
- **Embed frontend:** in this case the question would be if the programming language contains a simple way to implement the frontend, or if it is necessary to implement two different applications one for frontend and one for backend.

Table 6: Comparison of the main programming languages

Software	biological packages	platform	statistical calculations	easy frontend
R	Supported	cross-platform	suitable	TRUE
PYTHON	Supported	cross-platform	suitable	TRUE
JAVA	Supported	cross-platform	less suitable	FALSE
C++	Supported	cross-platform	less suitable	FALSE
JAVASCRIPT	Supported	cross-platform	less suitable	TRUE
PHP	Supported	cross-platform	less suitable	TRUE

3 Results

The application developed Genome quality tool also known as GQT, which is the combination of 3 different softwares and facilitates its use and helps us to obtain a clear output in a simple way. GQT uses R to compute and manipulate the data, and with the help of the Shiny package it presents an application with a graphical interface that is very intuitive and easy to use. Everything is executed in an Ubuntu virtual machine, one of the most known distributions of Linux, offering the chance, if necessary, to implement the application in a server in a simple and fast way. It has to be kept in mind that the implementation in a server occur outside the project scope due to technical and economic limitations.

The application is divided into two parts, the left part 7 contains the application menu, where we can change the input parameters and load our files in .FASTA format while the center/right 10 part contains three tabs with the outputs of each of the programs containing each of them a table with the data obtained and a graph that shows us in a visual way the data. Below we will detail each of the parts of the application in detail.

3.1 Menu

As shown in figure 7, the menu is vertically divided into three sections. The first, allows us to upload our archives in .FASTA format. The second one gives us crucial information to understand the operation of the application and to finish the buttons where we can change the modes.

Genome quality tool

Choose FASTA File

No file selected

Right now you are in test mode. You can load a FASTA file to get your own results but the processing time will depend on the size of the file and the characteristics of your machine.

Selected species:

Eukaryota

Mode:

Transcriptome

Proteome

Figure 7: Genome quality tool overview

3.2 Assembly-stats

In the Assembly-stats tab, we obtain information about the sequence we have entered. It shows the total size of the sequence, the average, the shortest, the longest among others. The most important information that shows Assembly-stats shown in the bar chart are the N statistics these are a measure to describe the quality of assembled genomes that are fragmented in contigs of different length.

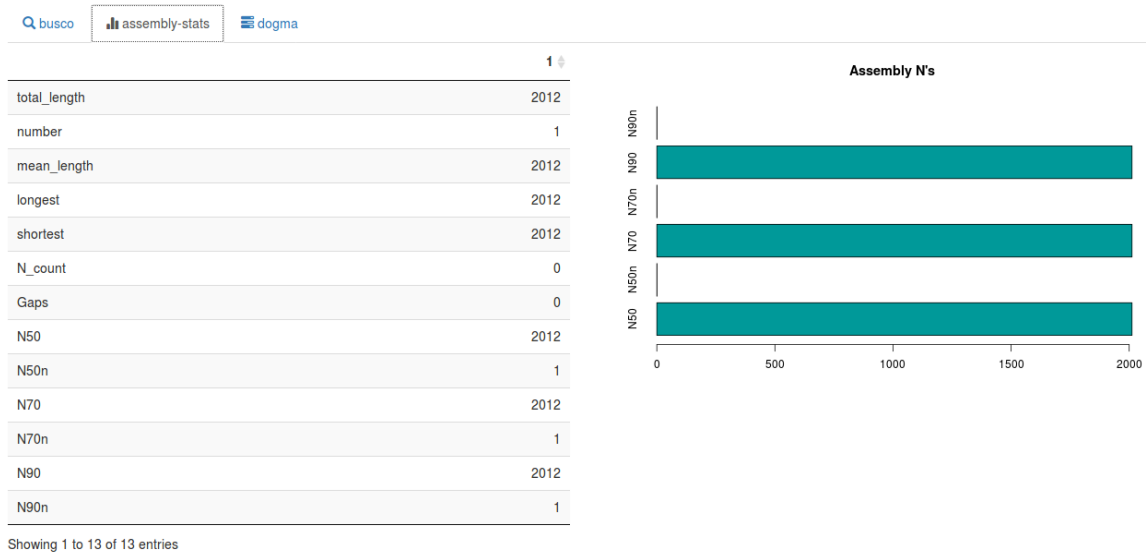


Figure 8: Genome quality tool Assembly-stats overview

3.3 Dogma

DOGMA is a program that assesses the quality of transcriptome and proteome data based on conserved protein domains. A core set of Conserved Domain Arrangements or CDAs is used in DOGMA to be compared against the transcriptome or proteome provided by the user. As it can be seen from figure 9, in DOGMA we will be able to see two different sections one with a table and another with a bar plot. In the table we should focus on the columns found and expct. Found refers to the number of CDA's found while expct refers to the number that the program expects to find. The percentage of how much completed each one of the CDA's sizes is can be displayed on the bar plot in visual way.

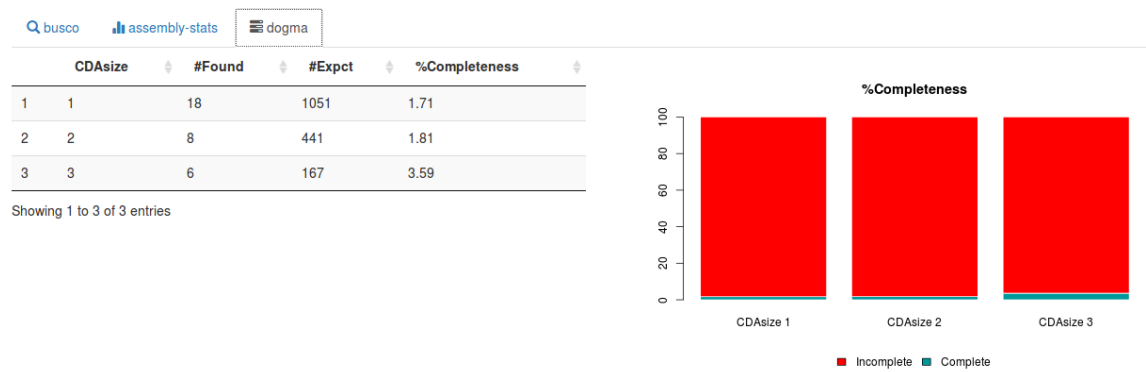


Figure 9: Genome quality tool Dogma overview

3.4 Busco

BUSCO is a program that provides quantitative measures for the assessment of genome assembly, gene set, and transcriptome completeness, based on evolutionarily-informed expectations of gene content from near-universal single-copy orthologs selected from OrthoDB. As usual the tab is divided in two vertical sections one displays a table and the other displays a graph. In this case the graph shows in a dynamic way the percentages of: Complete and single-copy BUSCOs, Complete and duplicated BUSCOs, Fragmented BUSCOs or Missing BUSCOs. We will detail each one of these values to understand it better.

- **Complete:** if found to be complete, either single-copy or duplicated, the BUSCO matches are scored within the expected range of scores and within the expected range of length alignments to the BUSCO profile. If in fact an orthologue is not present in the input dataset, or the orthologue is only partially present (highly fragmented), and a high-identity full-length homologue is present, it is possible that this homologue could be mistakenly identified as the complete BUSCO. The score thresholds are optimised to minimise this possibility, but it can still occur; [12]
- **Fragmented:** If found to be fragmented, the BUSCO matches have scored within the range of scores but not within the range of length alignments to the BUSCO profile. For transcriptomes or annotated gene sets this indicates incomplete transcripts or gene models. For genome assemblies this could indicate either that the gene is only partially present or that the sequence search and gene prediction steps failed to produce a full-length gene model even though the full gene could indeed be present in the assembly. Matches that produce such fragmented results are given a ‘second chance’ with a second round of sequence searches and gene predictions with parameters trained on those BUSCOs that were found to be complete, but this can still fail to recover the whole gene. Some fragmented BUSCOs from genome assembly assessments could therefore be complete but are just too divergent or have very complex gene structures, making them very hard to locate and predict in full. [12]
- **Missing:** If found to be missing, there were either no significant matches at all, or the BUSCO matches scored below the range of scores for the BUSCO profile. For transcriptomes or annotated gene sets this indicates that these orthologues are indeed missing or the transcripts or gene models are so incomplete/fragmented that they could not even meet the criteria to be considered as fragmented. For genome assemblies this could indicate either that these orthologues are indeed missing, or that the sequence search step failed to identify any significant matches, or that the gene prediction step failed to produce even a partial gene model that might have been recognised as a fragmented BUSCO match. Like for fragments, BUSCOs missing after the first round are given a ‘second chance’ with a second round of sequence searches and gene predictions with parameters trained on those BUSCOs that are complete, but this can still fail to recover the gene. Some missing BUSCOs from genome assembly assessments could therefore be partially present, and even possibly (but unlikely) complete, but they are just too divergent or have very complex gene structures, making them very hard to locate and predict correctly or even partially. [12]

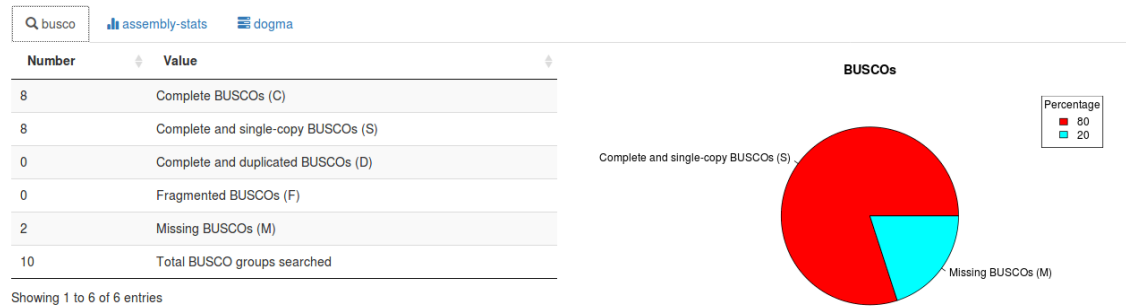


Figure 10: Genome quality tool Busco overview

4 Conclusions

- During the analysis phase we have learned the main qualitative and quantitative characteristics of a genome, we have also learned the methodology used to obtain the genome. Thus, we could be able to understand why quality is so important in these processes.
- After probing the different applications available, we have opted for Busco, Assembly-stats and Dogma because they meet all the necessary requirements to measure the quality of a genome in different ways. We have opted for them because they still receive updates from their developers.
- In the development phase we have established our knowledge of R with the development of a fully functional application starting from scratch. We have also learned to generate dynamic interfaces with Shiny. These two tools are often used together in bioinformatics. The development an application with these technologies has helped to develop confidence with them and be able to use them without problem.

5 List of terms

Acronyms

CDAs Conserved Domain Arrangements. 13

CPU Central Processing Unit. 8

DNA Deoxyribonucleic Acid. 14

GB Gigabyte. 8

GQT genome quality tool. 21

LTS Long Term Support. 8

PEC comes from Spanish "pruebas de evaluación continua" which means continuous assessment tests. 10

RAM Random-Access Memory. 8

UI User Interface. 28

Glossary

contigs Contigs are contiguous fragments of DNA sequence from an incomplete draft genome. 22

eukaryotes any organism having as its fundamental structural unit a cell type that contains specialized organelles in the cytoplasm, a membrane-bound nucleus enclosing genetic material organized into chromosomes, and an elaborate system of division by mitosis or meiosis, characteristic of all life forms except bacteria, blue-green algae, and other primitive microorganisms. 6

Gantt charts a diagram of the stages of a piece of work, showing stages that can be done at the same time, and stages that must be completed before others can start. 8

genomes Genetic material of an organism. 2

hardware the physical and electronic parts of a computer. 7

Shiny Shiny is an R package that helps to build interactive web apps. 8

software the instructions that control what a computer can do. 7

6 Bibliography

References

- [1] J. F. Abril and S. Castellano. Genome annotation. In S. Ranganathan, M. Gribskov, K. Nakai, and C. Schönbach, editors, *Encyclopedia of Bioinformatics and Computational Biology*, pages 195 – 209. Academic Press, Oxford, 2019.
- [2] A. Desai, V. S. Marwah, A. Yadav, V. Jha, K. Dhaygude, U. Bangar, V. Kulkarni, and A. Jere. Identification of optimum sequencing depth especially for de novo genome assembly of small genomes using next generation sequencing data. *PloS one*, 8(4):e60204, 2013.
- [3] V. Dominguez Del Angel, E. Hjerde, L. Sterck, S. Capella-Gutierrez, C. Notredame, O. Vinnere Petersson, J. Amselem, L. Bouri, S. Bocs, C. Klopp, J.-F. Gibrat, A. Vlasova, B. L. Leskosek, L. Soler, M. Binzer-Panchal, and H. Lantz. Ten steps to get started in Genome Assembly and Annotation. *F1000Research*, 7:148, feb 2018.
- [4] N. Fatima and S. Arabia. Performance Comparison of Most Common High Level Programming Languages. *International Journal of Computing Academic Research (IJCAR)*, 5(5):246–258, 2016.
- [5] R. A. Jensen. Orthologs and paralogs - we need to get it right. *Genome biology*, 2(8):INTERACTIONS1002, 2001.
- [6] A. Marchler-Bauer, M. K. Derbyshire, N. R. Gonzales, S. Lu, F. Chitsaz, L. Y. Geer, R. C. Geer, J. He, M. Gwadz, D. I. Hurwitz, C. J. Lanczycki, F. Lu, G. H. Marchler, J. S. Song, N. Thanki, Z. Wang, R. A. Yamashita, D. Zhang, C. Zheng, and S. H. Bryant. CDD: NCBI’s conserved domain database. *Nucleic acids research*, 43(Database issue):D222–6, jan 2015.
- [7] Metagenomics.wiki. N50 statistics - Metagenomics, 2019. <http://www.metagenomics.wiki/pdf/definition/assembly/n50>.
- [8] J. C. Mogul, J. F. Bartlett, R. N. Mayo, and A. Srivastava. Performance Implications of Multiple Pointer Sizes. In *Proceedings of USENIX Winter*, pages 187–200, 1995.
- [9] National Center for Biotechnology Information. RefSeq: NCBI Reference Sequence Database, 2018.
- [10] I. RStudio. Shiny, 2019. <https://shiny.rstudio.com/>.
- [11] H. Salameh. What, When, Why, and How? A Comparison between Agile Project Management and Traditional Project Management Methods. *International Journal of Business and Management Review*, 2(52):52–74, 2014.
- [12] F. A. Simão, R. M. Waterhouse, P. Ioannidis, E. V. Kriventseva, and E. M. Zdobnov. BUSCO: Assessing genome assembly and annotation completeness with single-copy orthologs. *Bioinformatics*, 31(19):3210–3212, 2015.
- [13] TeamGantt. Online Gantt Chart Software — TeamGantt. <https://www.teamgantt.com/>.
- [14] R. M. Waterhouse, F. Tegenfeldt, J. Li, E. M. Zdobnov, and E. V. Kriventseva. OrthoDB: a hierarchical catalog of animal, fungal and bacterial orthologs. *Nucleic Acids Research*, 41(Database issue):D358, jan 2013.

7 Annexes

Here we will explain in detail the most important parts of the source code from the application. The code of this application is written in R, thus we decided to write the application in the mode one single app, that means that the application is written in a single R file.

7.1 Load requirements

Most of R applications starts loading the external packages that will be used subsequently by the application. This app imports different packages to help us to manipulate the data once loaded and imports the Shiny package, to be able to display the graphical part of the app.

Code 11: Source Code - Load requirements

```
1 library(shiny)
2 library(DT)
3 library(data.table)
4 library(RColorBrewer)
```

7.2 Create Shiny app

Every Shiny app has two parts: UI and Server. The UI defines the components that will be displayed on the browser and the server part contains the functions with the logic of the app.

Code 12: Source Code - Create Shiny app

```
1 shinyApp(ui, server)
```

7.2.1 UI

The Ui is almost self-explanatory, we define the layouts and create the html components like radiobuttons, inputs and panels.

Code 13: Source Code - UI

```
1 ui <- fluidPage(# App title ----
2   titlePanel("Genome quality tool"),
3
4   # Sidebar layout with input and output definitions ----
5   sidebarLayout(
6     # Sidebar panel for inputs ----
7     sidebarPanel(
8       # Input: Select a file ----
9       fileInput(
```

```

10         "file1",
11         "Choose FASTA File",
12         multiple = FALSE,
13         accept = c("text/csv",
14                   "text/comma-separated-values,text/plain",
15                   ".csv")
16     ),
17
18     # Horizontal line ----
19     tags$hr(),
20
21     #Help text
22     helpText(
23         "Right now you are in test mode. You can load a FASTA file to get your own results but
24         the processing time will depend on the size of the file and the characteristics of your
25         machine."
26     ),
27
28     tags$hr(),
29
30     radioButtons(
31         "radio",
32         label = h4("Selected species:"),
33         choices = list(
34             "Eukaryota" = 1
35         ),
36         selected = 1
37     ),
38
39     radioButtons(
40         "radio2",
41         label = h4("Mode:"),
42         choices = list("Transcriptome" = 1,
43                       "Proteome" = 2),
44         selected = 1
45     )
46 ),
47
48
49     # Main panel for displaying outputs ----
50     mainPanel(# Output: Tabset w/ plot, summary, and table ----
51               tabsetPanel(
52                 type = "tabs",
53                 tabPanel(
54                     "busco",
55                     fluidRow(column(6, DT::dataTableOutput("busco")),
56                             column(6, plotOutput('plot1'))),
57                     icon = icon("glyphicon glyphicon-search", lib = "glyphicon")
58                 ),
59                 tabPanel(

```

```

60         "assembly-stats",
61         fluidRow(column(6, DT::dataTableOutput("assembly")),
62                 column(6, plotOutput('plot_assembly'))),
63         icon = icon("glyphicon glyphicon-stats", lib = "glyphicon")
64     ),
65     tabPanel(
66         "dogma",
67         fluidRow(column(6, DT::dataTableOutput("dogma")),
68                 column(6, plotOutput('plot_dogma'))),
69         icon = icon("glyphicon glyphicon-tasks", lib = "glyphicon")
70     )
71 ))
72 ))

```

7.2.2 Server

The server part contains the logic of the application. Here we see the `init` functions for the first time but we have to highlight two things: the function `observe` and the function `StartMainProcess`. The `observe` function executes the `StartMainProcess` function every time it detects a change in the input. The `StartMainProcess` function initializes the general process that is responsible for processing the data.

Code 14: Source Code - Server

```

1 server <- function(input, output) {
2   init_assembly(output)
3   init_busco(output)
4   init_dogma(output)
5
6
7   observe({
8     inFile <- input$file1
9     if (is.null(inFile))
10      return(NULL)
11
12     StartMainProcess(inFile$datapath, output, input)
13
14   })
15
16
17 }

```

7.3 Important functions

7.3.1 StartMainProcess

This function covers the general process, so every time a new file is loaded, all the code necessary is executed. It is also responsible to prepare the system commands that will be executed later.

Code 15: Source Code - StartMainProcess

```
1 StartMainProcess <- function(seq_filepath, output,input) {
2   Sys.getenv("$PATH")
3
4   from_assembly <- "/home/osboxes/ASSEMBLY-STATS/"
5   from_busco <- "/home/osboxes/BUSCOVM/busco3/"
6   from_pfmscan <- "/home/osboxes/DOGMA/PfamScan/"
7   from_dogma <- "/home/osboxes/DOGMA/"
8
9
10  if (input$radio2 == "1"){
11    busco_mode <- "geno" # other option tran
12    dogma_mode <- "transcriptome"
13  }
14  if (input$radio2 == "2"){
15    busco_mode <- "prot"
16    dogma_mode <- "proteome"
17  }
18
19
20  # perl pfam_scan.pl -fasta test_data/target.fa -dir Bio/Pfam/ -outfile TFM_outputs/pfam_out_TFM
21  command_pfmscan <-
22    paste(
23      "perl pfam_scan.pl -fasta",
24      seq_filepath ,
25      "-dir Bio/Pfam/",
26      "-outfile TFM_outputs/pfam_out_TFM" ,
27      sep = " "
28    )
29
30
31  # python dogma.py transcriptome -a PfamScan/TFM_outputs/pfam_out_TFM -o TFM_outputs/Dogma_out_TFM
32  command_dogma <-
33    paste(
34      "python dogma.py",
35      dogma_mode,
36      "-a PfamScan/TFM_outputs/pfam_out_TFM",
37      "-o TFM_outputs/Dogma_out_TFM" ,
38      sep = " "
39    )
40
41  # assembly-stats -t test_files/fastq_unittest.fastq > TFM_outputs/assembly
42  command_assembly <-
43    paste(
44      "assembly-stats -t",
45      seq_filepath ,
46      ">",
```



```

47     paste(from_assembly, "TFM_outputs/assembly", sep = ""),
48     sep = " "
49 )
50
51 # python scripts/run_BUSCO.py -i sample_data/target.fa -o TFMBusco -l sample_data/example -m geno -f
52 command_busco <-
53     paste(
54         "python3",
55         paste(from_busco, "scripts/run_BUSCO.py -i", sep = " ") ,
56         seq_filepath ,
57         "-o TFMBusco",
58         paste("-l ", from_busco, "sample_data/example" , sep = " "),
59         "-m",
60         busco_mode,
61         "-f" ,
62         sep = " "
63     )
64
65
66 # Create a Progress object
67 progress <- shiny::Progress$new()
68 # Make sure it closes when we exit this reactive, even if there's an error
69 on.exit(progress$close())
70 progress$set(message = "General status:", value = 0)
71 # Number of steps
72 n <- 3
73
74
75
76
77 # assembly
78 progress$inc(1 / n, detail = "assembly is running")
79 StartProcess_assembly(command_assembly)
80 init_assembly(output)
81
82
83 #busco
84 progress$inc(1 / n, detail = "busco is running")
85 StartProcess_busco(command_busco)
86 init_busco(output)
87
88
89 #dogma
90 progress$inc(1 / n, detail = "dogma is running")
91 StartProcess_dogma(command_pfmscan,command_dogma)
92 init_dogma(output)
93
94
95
96 }

```

7.3.2 ReadFile

The ReadFile' functions are responsible for reading the file generated by each of the programs, making a first manipulation of the files to make them more user-friendly.

Code 16: Source Code - ReadFile'assembly

```
1 ReadFile_assembly <- function(workingDir) {
2   file <-
3     read.table(file.path(workingDir, "assembly"),
4               sep = "",
5               header = TRUE)
6   return(file[,-1])
7 }
```

Code 17: Source Code - ReadFile'busco

```
1 ReadFile_busco <- function(workingDir) {
2   fileBUSCO <-
3     read.table(
4       file.path(workingDir, "run_TFMBusco/short_summary_TFMBusco.txt"),
5       sep = '\t',
6       comment.char = "#",
7       header = FALSE,
8       fill = TRUE
9     )
10  #levels(fileBUSCO$V3)[levels(fileBUSCO$V3)==''] <- NA
11
12  fileBUSCO <- fileBUSCO[-1,-1]
13  rownames(fileBUSCO) <- NULL
14
15  colnames(fileBUSCO) <- c("Number", "Value")
16  return(fileBUSCO)
17 }
```

Code 18: Source Code - ReadFile'dogma

```
1 ReadFile_dogma <- function(workingDir) {
2   res <- readLines(file.path(workingDir, "Dogma_out_TFM_clean"))
3   return(res)
4 }
```

7.3.3 Init

Init functions are those in charge of finalizing the manipulation of the data and sending them to the objects that are in charge of showing the results in a visual way.

Code 19: Source Code - init`assembly

```
1 init_assembly <- function(output) {
2   workingDir_assembly <- "/home/osboxes/ASSEMBLY-STATS/TFM_outputs/"
3   file_assembly <- ReadFile_assembly(workingDir_assembly)
4
5   test <- transpose(file_assembly)
6   colnames(test) <- rownames(file_assembly)
7   rownames(test) <- colnames(file_assembly)
8
9   #output assembly ----
10  output$assembly <-
11    DT::renderDataTable(DT::datatable(test, options = list(
12      paging = FALSE, searching = FALSE
13    )))
14
15
16  output$plot_assembly <- renderPlot({
17    local_data <- data.matrix(file_assembly[, 8:13])
18    barplot(
19      local_data,
20      col = c("#009999"),
21      main = "Assembly N's",
22      horiz = TRUE
23    )
24
25  })
26
27
28
29 }
```

Code 20: Source Code - init`busco

```
1 init_busco <- function(output) {
2   #Working Directories
3   workingDir_busco <- "/home/osboxes/BUSCOVM/busco3/"
4   file_busco <- ReadFile_busco(workingDir_busco)
5
6   #output busco ----
7   output$busco <-
8     DT::renderDataTable(DT::datatable(
```

```

9     file_busco ,
10    rownames = FALSE,
11    options = list(paging = FALSE, searching = FALSE)
12  ))
13
14  output$plot1 <- renderPlot({
15    # Create data for the graph.
16    local_data <- file_busco[-1,]
17    local_data <- local_data[-5,]
18    #local_data<- local_data[-4,]
19
20    local_data[local_data == 0] <- NA
21    local_data <- local_data[complete.cases(local_data),]
22
23    x <- as.numeric(as.character(local_data$Number))
24    #labels <- row.names(local_data$Value)
25    leg <- local_data$Value
26    #print(as.numeric(as.character(local_data$Number)))
27    piepercent <- round(100 * x / sum(x), 1)
28
29    # Plot the chart.
30    pie(x,
31        labels = leg,
32        main = "BUSCOs",
33        col = rainbow(length(x)))
34    legend(
35        "topright",
36        title = "Percentage",
37        as.character(piepercent),
38        fill = rainbow(length(x))
39    )
40  })
41
42 }

```

Code 21: Source Code - init'dogma

```

1  init_dogma <- function(output) {
2    workingDir <- "/home/osboxes/DOGMA/TFM_outputs/"
3    file_dogma <- ReadFile_dogma(workingDir)
4
5    MYDF <-
6      rbind.data.frame(unlist(strsplit(file_dogma[11], "\t", fixed = TRUE)), unlist(strsplit(file_dogma[12], "\t", fixed =
7      colnames(MYDF) <-
8        unlist(strsplit(file_dogma[9], "\t", fixed = TRUE))
9
10   output$dogma <-

```

```

11     DT::renderDataTable(DT::datatable(MYDF, options = list(
12         paging = FALSE, searching = FALSE
13     )))
14
15
16
17     output$plot_dogma <- renderPlot({
18         par(oma = c(4, 1, 1, 1))
19
20         locald <- t(data.matrix(MYDF[, 4]))
21
22         locald <-
23             rbind(locald, c(
24                 100 - as.double(locald[1, 1]),
25                 100 - as.double(locald[1, 2]),
26                 100 - as.double(locald[1, 3])
27             ))
28
29         barplot(
30             locald,
31             col = c("#009999", "red") ,
32             border = "white",
33             main = "%Completeness" ,
34             names.arg = c("CDAsize 1", "CDAsize 2", "CDAsize 3"),
35             legend = c("Complete", "Incomplete"),
36             args.legend = list(
37                 x = "bottom",
38                 bty = "n",
39                 inset = c(0, -0.4),
40                 horiz = TRUE,
41                 xpd = TRUE
42             )
43         )
44     })
45
46
47 }

```

7.3.4 StartProcess

These functions are responsible for executing third-party programs in which the environment is prepared and the system commands are executed.

Code 22: Source Code - StartProcess'assembly

```

1 StartProcess_assembly <- function(command_assembly) {
2     system(command_assembly)
3 }

```

Code 23: Source Code - StartProcess`busco

```
1 StartProcess_busco <- function(command_busco) {  
2   setwd("/home/osboxes/BUSCOVM/busco3")  
3   system(command_busco)  
4   setwd("../TFM/app/")  
5 }
```

Code 24: Source Code - StartProcess`dogma

```
1 StartProcess_dogma <- function(command_pfmscan,command_dogma) {  
2   setwd("/home/osboxes/DOGMA/PfamScan")  
3   system("rm -f TFM_outputs/pfam_out_TFM")  
4   system(command_pfmscan)  
5   setwd("../")  
6   system("rm -f TFM_outputs/Dogma_out_TFM")  
7   system(command_dogma)  
8   system("head -n 18 TFM_outputs/Dogma_out_TFM > TFM_outputs/Dogma_out_TFM_clean")  
9 }
```
