Universidade do Minho
Escola de Engenharia

Gil da Lomba Afonso

Dotlet Reborn: diagonal dot plots in a web browser

February 2021

Dotlet Reborn: diagonal dot plots in a web browser
Gil da Lomba Afonso
Uminho | 2021

Universidade do Minho
Escola de Engenharia

Gil da Lomba Afonso

Dotlet Reborn: diagonal dot plots in a web browser

Master dissertation in Bioinformatics

Dissertation supervised by
Björn Johansson

February 2021

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Acknowledgements

For the development of this dissertation the support and collaboration from multiple people was important. To them I shall give a special acknowledgment. First I want to thank my dissertation supervisor, Professor Björn Johansson, for giving me the opportunity to improve this application, leading to either my academic and personal growth. I also want to thank for the trust and respect shown for my work. I also want to thank Sebastien Moretti from the Swiss Institute of Bioinformatics (SIB) for the availability to discuss and help with any issue I encountered during the development of this dissertation.

Now I want to thank my other half, Beatriz Ribeiro, for always being there through my ups and downs, always willing to listen and support me. Also, for the affection shown especially when I was feeling mentally tired, for believing in my skills, and for trying to guide me in the proper direction. I also want to thank Pedro Quintas for the interest shown and for the willingness to listen and help in any issue I encountered. I want to thank my family for the support shown in many levels, especially to my grandparents and bigger sister, for the courage transmitted and the affection, and availability shown throughout this dissertation.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration. I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Resumo

O método de gráfico de pontos diagonais é usado para comparar sequências de aminoácidos em diferentes proteínas, de modo a relacionar as similaridades entre ambas as sequências, e também para apresentar uma classificação das proteínas baseada nessas similaridades. É também usada para detetar repetições nas sequências de aminoácidos de múltiplas proteínas, e para detetar regiões de ligações existentes de bases nas sequências de nucleótidos de um ácido nucleico.

Existe um vasto número de tipos de matrizes de *score* usadas para comparar pares de sequências. Na comparação entre as sequências, a matriz de *score* apresenta um valor para as igualdades, para as diferenças, e para as substituições, influenciando assim o alinhamento da sequência de DNA ou proteína. O *score* e a análise dos resultados dependem do tipo de matriz de *score* aplicada. O tipo de matrizes que usamos na plataforma (dotlet) são a PAM, BLOSUM e Gonnet, em que a última apenas é utilizada na primeira versão da aplicação dotlet.

A aplicação dotlet foi inicialmente desenvolvida em 1998 e em Java, tendo como programadores Marco Pagni e Thomas Junier, ambos do Instituto Suíço de Bioinformática. A aplicação foi desenvolvida devido à falta de aplicações que usem o método das matrizes de *score*, sejam possíveis de utilizar em qualquer sistema operativo, e também devido à necessidade da existência de uma aplicação com estas ferramentas para as aulas práticas de bioinformática no Instituto de Bioquímica. Esta aula prática iria ser baseada na *World Wide Web* (www), pelo que a existência de um programa que funcionasse num *browser* era crucial. Devido a estes requisitos e também de modo a criar uma versão melhorada de outras aplicações já existentes que aplicassem o método dos gráficos de pontos diagonais, o dotlet foi desenvolvido. A distribuição da primeira versão do dotlet pode ser encontrada no sítio https://github.com/sib-swiss/dotlet.

Apesar de ter estado disponível por uma larga extensão de tempo, os *browsers* acabaram por deixar de suportar Java, o que levou a que a versão original do dotlet deixasse de ser possível de ser utilizada na *web*. Desta forma, uma nova versão do dotlet foi desenvolvida por Julien Delafontaine com tecnologias atualizadas. Esta versão contém algumas das ferramentas que a versão original oferecia ao utilizador, sendo ao mesmo tempo uma versão aprimorada em termos de usabilidade por parte do utilizador, levando a um melhor desempenho comparando à versão anterior. Este projeto pode ser encontrado no sítio https://github.com/sib-swiss/react-dotlet.

Apesar da nova versão do dotlet ser uma melhoria da versão anterior, existem algumas ferramentas que ainda não estão implementadas. Estas ferramentas são a possibilidade de o utilizador guardar sequências identificadas por um nome escolhido pelo mesmo utilizador, e posteriormente a possibilidade

de selecionar uma das sequências guardadas para o alinhamento de sequências. Outra funcionalidade imperativa é a existência de uma aplicação de *desktop*, para assim abranger mais elementos da comunidade científica e também para ser possível de ser utilizada quando o acesso à internet não é possível. Outra funcionalidade desenvolvida foi a capacidade de, ao ser feito o *copy-paste* de um sítio da base de dados da UniProt (por exemplo, https://www.uniprot.org/uniprot/E5G0U9.fasta), que contenha um ficheiro *fasta*, a sequência é retirada deste ficheiro e aplicada no alinhamento de sequências. O código desta dissertação está disponível em https://github.com/gilafonso/react-dotlet.

Keywords: gráfico de pontos diagonais, dotlet, matriz de *scores*

Abstract

The diagonal plot method is used to compare sequences of amino acids in different proteins, to estimate the similarities between both sequences, and to present a classification of the proteins based on those similarities. It is also used to detect repetitions in the compared amino acid sequences of multiple proteins, and to detect regions of existing base-pairing in the nucleotide sequence of a nucleic acid.

There are multiple types of scoring matrices used to compare a pair of sequences. The scoring matrix provides a score for matches, mismatches, and substitutions, influencing the DNA or protein sequence alignment. The score and analysis outcome differs from each type of scoring matrix. The type of scoring matrices we use in the dotlet platform are the PAM, BLOSUM and Gonnet, the latter being only used in the first dotlet version.

Dotlet, first developed in Java, in 1998, was written by Marco Pagni and Thomas Junier from the Swiss Institute of Bioinformatics (SIB). It was developed since there was a lack of cross-platform applications that applied the diagonal plot tool, and also due to a need of having such a platform for practical lessons in bioinformatics at the Institute of Biochemistry. The practical lesson was going to be based on the World Wide Web (www), so there was a need to have a program that would run in a web browser. Due to this need and also to create an improved application of the previous existing applications that applied the diagonal plot method, dotlet was developed. The distribution of the first dotlet version can be found in https://github.com/sib-swiss/dotlet.

Even though it was available for many years, browsers eventually stopped supporting Java, leading the original dotlet version to become obsolete. Therefore, a new version of dotlet, developed by Julien Delafontaine was created with updated technology. This version contains multiple features that the original dotlet version offered to the user, while having an improved user interface leading to an overall platform enhancement compared to the previous one. This project can be found in https://github.com/sib-swiss/react-dotlet.

Despite the newer dotlet version being an improvement of its previous version, there were still some features missing. These features were the capability of the user to save sequences which can later be selected for the sequence alignment and identified with a name the user creates for it. Another feature was the offline version of the application, a crucial feature that embraces more users and can be used whenever one does not have internet access. A feature that was also implemented in the application was the capability of the user to paste an URL from the UniProt database, containing the sequence within a *fasta* file (e.g., https://www.uniprot.org/uniprot/E5G0U9.fasta), which then is accessed and the

sequence is gathered and presented in the alignment. The codebase of this dissertation is available in https://github.com/gilafonso/react-dotlet.

Keywords: diagonal plot, dotlet, scoring matrices

INDEX

FIGURES LIST

1 Introduction

1.1 Context and Motivation

Dot plot is a conceptually simple, yet very powerful method for the pairwise alignment of biological sequences. Alignments are important in the inference of phylogenetic relationships and in comparative genomics. The dotlet software was developed due to a lack of cross-platform programs that applied the diagonal plot method [1, 2]. Programs that applied this method already exist (e.g. the GCG package DotPlot [3]) but although users could produce results with these programs, they were outdated and required the common user to be familiar with a command-line interface. Also, the implementation did not allow the change of signal-to-noise ratio without recomputing the whole comparison plot, leading to several runs performed until the user could get a satisfactory result. Also, the similarity between the sequences was also represented as either a black or a white pixel filtering out information regarding degrees of homology.

With versions implementing a graphical user interface (GUI), these problems started to change. Dotter, a software developed in 1995, is a graphical diagonal plot program for Windows which can compare DNA versus DNA, protein versus protein, or DNA versus protein sequences [4]. The main upgrade to the previous versions was that only one calculation needed to be done for the comparisons to be made. Since it is GUI driven, users change parameters in real time, achieving a maximal signal-to-noise ratio without having to recompute the whole plot multiple times. This real time manipulation of the output result is possible due to the dynamical change of the greyscale render of the dots, achievable because a “Greyramp tool” was developed. This tool allows the user to interactively change the score limits for the greyscale rendering, altering the signal and noise, achieving a result that best fits the objectives. Additional features present in this software that are useful for the user are the diagonal plot compression, the ability to control the zoom with one’s mouse, the display of the sequence alignment and the ability to save or load diagonal plots.

There are several methods to compare sequences of amino acids from different proteins, to estimate the similarity of the sequences and to use these estimates to classify the proteins. One of these methods is the transition matrix method to compare sequences [5]. It is used in describing and classifying proteins by their amino acid sequences. By comparing the frequency with which different amino acid pairs occurred, two hundred and sixteen proteins were classified. There is also a method that detects gaps in amino acid sequences [6]. In order to optimize the homology between two proteins, there are three steps: a demonstration that proves the homology of the sequences, the location where the homologous pairing occurs, and the location of gaps between

these regions in order to minimize the number of mutations necessary for the differences between the two sequences to be accounted for. These methods are similar to the ones used to measure edit distance between text strings such as the widely used Levenshtein distance measure. The diagonal plot method, developed in 1970, is an alternative to the methods explained above, while also being a similar principle to the latter method. This method compares sequences in pairs and presents the results in a diagram. It is thoroughly explained in the next chapter.

Dotlet applies this method within its program, while enhancing the interaction between itself and the user. Dotlet is meant to run in a web browser and is therefore platform independent and does not require software installation. For these reasons, the broad community was able to take advantage of this software, as both an investigative and educational tool.

However, as technology is updated, browsers stopped supporting Java, which led to this primary version to become obsolete [7]. A new version of dotlet, as a web application, was then developed. This version, compared to the previous one, is modernized with up-to-date technologies that enhance the user experience by highlighting key aspects of the sequence comparison and by performance improvements of the whole web application.

Even though the previous web application was a great tool, it lacked a version of it as a desktop application. This desktop application can be run offline, ideal for situations when there is no connection to the internet. This can be important for the educational community, since it can be used in exams and other situations where the internet access can be restricted. Due to these reasons, we developed a desktop application that replicates the interface of the web application, allowing for its smoother comprehension and usability.

1.2 Goals

Taking into consideration the scientific community needs and the existing gap of the platform independent version of dotlet, we decided to improve the application. For this to be achieved, we delineated some key points for the project that will improve the user experience. These are:

- Enhance the current web application design;
- Load sequences through an external URL;
- Save the list of loaded sequences;
- Create a stand-alone version that may run offline for educational purposes.

With the improvement of the user experience and to reach a broader community as the main objective, we initially upgraded the current web application design by retorting to CSS and Javascript [8, 9]. This upgrade was a continuous work throughout the completion of the other objectives.

In order to load sequences from a URL address, a set of functions were written to fetch the sequences from a URL that contains a *fasta* file. We will use the UniProt database as a reference, due to its popularity, easefulness at searching sequences, and because it stores its sequences in a *fasta* file with the common format.

For the development of the offline version of dotlet we considered resorting to Docker, by creating a container in a linux OS (operating system) and saving the dotlet version into the container [10, 11]. Upon a further research we decided to use *electronjs*, a known framework that is used to build desktop applications and can replicate the dotlet web application [12].

1.3 Structure

This dissertation is divided into six chapters. In the first chapter we have the introduction, where we present the context of the subject in study along with the motivation that led to the development of this project. Also, we mention what goals we defined for this project and how we intended to achieve them.

In the second chapter we have the state of art, where recurring to the bibliographic search, many studies involving the development of our software are thoroughly explained.

In the third chapter we present the technologies that were used to develop dotlet. We thoroughly explain how and why we utilized certain frameworks, software's, and programming languages.

In the fourth chapter we go through the functions that we wrote, in order to improve dotlet. We divide the chapter into sub-chapters, where each of those are relative to a goal present in chapter 1.2.

The fifth chapter is a walk-through of every step one could take to make use of the application. We thoroughly explain the workflow of the application and also how one should use the desktop application.

Lastly we have the sixth chapter, where we conclude the work done in this dissertation. We also present possible future improvements for the application.

2 State of the Art

2.1 Diagonal Plot Method

The Diagonal Plot Method is simple, giving anyone the opportunity to do sequence comparison without resorting to any computer software [5]. It provides similarity options for classifying 25 cytochromes (proteins) by their amino acid sequences; it is used to detect repetitions in the amino acid sequences of various proteins; and is used to detect regions of existing base-pairing in the nucleotide sequence of a nucleic acid.

The sequence comparison of amino acids from different proteins are made in pairs. For every single comparison, both sequences are displayed in a diagram, on adjacent sides. Within this diagram the matched amino acids from the same row and column position are displayed with a dot. The similarities of the two sequences are represented by the diagonal line the dots form. In order to test if there are more matches than expected, two methods were used:

1. A direct calculation is made between the frequencies of lengths of all unbroken diagonal matches and the comparison of these with the frequencies of runs expected.
2. The matches in the diagonals of the diagram are compared with the numbers expected in each diagonal, if the sequences show no similarity.

The following figure 1 shows three diagrams, comparing human cytochrome c with the cytochrome c of rhesus monkey, tuna fish, and the bacterium *Rhodospirillum rubrum*. It illustrates the different types of diagrams obtained with sequences of different similarities. The monkey and fish cytochromes almost entirely match with the human cytochrome. In contrast, *Rhodospirillum* cytochrome, compared with the human cytochrome, have its line of matches in different parts of the sequences, in different diagonals, proving that parts of one sequence do not match the other.

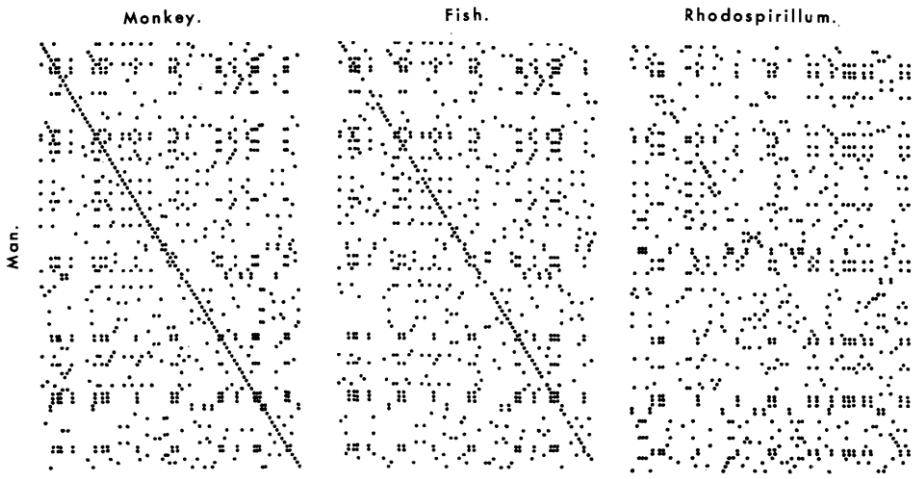

Figure 1 - Diagrams of the comparisons between human cytochrome c (left margin) and the cytochromes c of monkey, fish, and *Rhodospirillum rubrum* (upper margin).

2.2 Scoring Matrices

There are different types of scoring matrices with the purpose of comparing a pair of protein sequences. This method comprises a score for matches, mismatches, and substitutions which can influence the DNA and protein sequences alignment. For each type of matrix there are different scores and analysis outcomes. The types of matrices we utilize in dotlet are PAM, BLOSUM, and Gonnet [13, 14].

The PAM matrices are based on the Markov model of protein evolution and is a probabilistic model for amino acid replacement. It is focused on the change of one amino acid to another in closely related sequences, and the origin of its evolutionary path. The amino acid replacements conserve their size, charge, hydrophobicity, among other characteristics. If these characteristic replacements present a higher-than-expected prevalence within the two aligned sequences, then these sequences are related.

The PAM matrices have a computing method in which the base model is called PAM 1 matrix. This results in the computed probability of one substitution per 100 amino acids. For higher PAM matrices, one can simply compute it by multiplying the base model against itself, the number of times defined. For example, the PAM 50 matrix means that the base model was multiplied by 50 times, meaning in 100 amino acids 50 substitutions were made. Alongside this example, the PAM 250 matrix means that in 100 amino acids, 250 substitutions were made, meaning there are 2.5 amino acid replacements per site.

A protein family and superfamily are defined by the percentage of identical amino acids within the sequences. A protein family comprises 85%, or greater, similarity between sequences, and a

protein superfamily comprises 30%, or greater, resemblance to each other. The latter may contain many protein families.

For the whole range of possible PAM matrices, some stand out because of the different tasks they compute. The PAM 250 matrix is the most used to locate all the potential similarities for a pair of sequences, providing the best evolutionary view of the protein sequence comparison. It is commonly used when one does not know the origin of the protein or if it is part of a larger protein, and to build a phylogenetic tree of the protein. It is capable of accurately detecting similarities in the 30% range. The PAM 160 matrix is used to determine if a certain protein sequence is part of a particular protein family, reducing possible irrelevant matches. It detects the similarity between the protein sequences in the 50% to 60% range. The PAM 40 matrix aims to reduce the unwanted matches even greatly, finding protein sequences in the 70% to 90% matching range.

Although the PAM matrix is a successful method to score protein sequences, any errors in the PAM 1 matrix are magnified in the PAM 250 matrix due to the matrix multiplications. This led to an improvement of this method, entitled the BLOSUM matrices.

BLOSUM matrices rely on data from the *blocks* database. These *blocks*, with varying similarity, are partial ungapped protein sequence alignments. In comparison to the PAM matrices, frequently represented sequences were not excluded from the equation. To avoid biased matrices during the model construction, highly related and frequent sequences were removed. By the time the model calculation was concluded, the matrices presented a high number of hydrophobic and non-globular proteins.

Many sequences in the block are close to be identical or even identical, leading to a matrix with a bias for evolutionarily close sequences. To resolve this issue, similar sequences in a certain block, above the specified similarity threshold percentage, are clustered. This effectively reduces the influence of closely related sequences, improving the influence of distantly related sequences. There are many types of BLOSUM matrices, and the different types differ according to the clustering threshold. BLOSUM 80 means the clustering threshold is equal to 80%. For the most commonly used BLOSUM matrix, BLOSUM 62, the clustering is equal to 62%, reducing the number of blocks by 25%. Although this is a significant reduction in data redundancy, there are still 1.25×10^6 amino acid pairs taken into consideration for the calculation of the scoring matrix. If the clustering threshold decreases, the distantly related sequences have an increased contribution to the final scoring matrix. The BLOSUM matrix number is inversely proportional to the ability to detect distantly related sequences.

The BLOSUM 62 matrix, in direct comparison to the PAM 250 matrix, is superior in detecting distant related sequences, even if current data sets are applied in both methods. Also, it provides the best evolutionary view in a protein sequence comparison with the widest range of proteins similar to the protein of interest. When the protein is unknown or is possible to be a fragment of a larger one, then this type of matrix is best to use. It is also the best matrix to use in the case of building a phylogenetic tree of our protein of interest and examining the protein resemblance to other proteins. This matrix detects similarities of around 30%, detecting superfamilies.

The BLOSUM 80 matrix detects matches with similarities of around 50%, reducing possible irrelevant matches the BLOSUM 62 matrix could return. This type of matrix serves the purpose to determine if a certain protein sequence is part of a particular protein family.

The BLOSUM 90 matrix determines which proteins are the most similar to the given protein sequence.

The PAM matrices derive from protein data available from the 1960s to 1970s. Most of the protein sequences known at that time were small, globular, and hydrophilic due to the possibility to isolate those types of proteins. For hydrophobic protein sequences, using BLOSUM matrices is more useful to the user. Mathematical analysis compared these two methods, showing the equivalence between PAM 250 and BLOSUM 45, PAM 160 and BLOSUM 62, and PAM 120 with BLOSUM 80 matrices.

PAM 160 is more tolerant to hydrophilic amino acid substitutions compared to BLOSUM 62, while being less tolerant to hydrophobic amino acid substitutions compared to the same BLOSUM matrix. Even though PAM 250 and BLOSUM 62 matrices detect similarities in the 30% range, BLOSUM uses more recent data, which translates into a higher number of proteins. Therefore, PAM 250 is equivalent to BLOSUM 45 while considering all types of proteins.

The matrices explained above are intended to compare protein sequences against each other. However, if one wishes to detect specific features in a protein family, one has to take advantage of other methods. For this end, the Gonnet matrix was created. It is based on the alignment of the SwissProt database, dated 1991, with a total of 1.7×10^6 matches used from sequences differing by the interval of 6.4 to 100 PAM units. This method has a vast yet selective coverage of protein sequences, since the SwissProt database contains only selected families. This matrix is convenient to use due to the high quality annotation of proteins included in SwissProt.

2.3 Dotlet

Dotlet, a software developed in 1998, compares sequences using the diagonal plot method. Although versions of the diagonal plot tools developed previously to dotlet were usable, they were

strict to some operating systems, creating a barrier to potential users. This led to the development of dotlet so it could run on multiple platforms. This software, developed in Java, was designed to be platform independent, by being able to be run in a web browser, and/or to be run as a desktop application [7].

To interact with the platform, initially the user imports sequences using the copy-paste method and then from the menu selects which two sequences to compare. Within the menu one also selects the matrix score, zoom feature and comparison window size. There are numerous scoring matrices available, including PAM, Gonnet, and BLOSSUM.

Within the menu one can either select protein versus protein, DNA versus DNA, or mixed comparisons. The mixed comparison works with the DNA being firstly translated in the three forward reading frames, and then a protein versus protein comparison is performed.

After the comparison is complete, the diagonal plot is shown in the main panel (fig. 2A), along with a histogram of score frequencies (Fig. 2B) and the sequence alignment panel (fig. 2C). The histogram allows the user to select between the region of similarities by setting the grey scale in order to highlight the significant regions. From the main panel the user can select any pixel which makes the sequence alignment panel show the exact sequence comparison, in that precise pixel position. In the alignment panel the positively scored pairs are colored in blue. For the mixed comparison (DNA versus protein), this panel will show the three reading frames of the DNA sequence versus the protein sequence. This allows them to easily identify introns and frame shifts. For DNA versus DNA comparisons, the panel presents a second alignment containing the horizontal sequence against the reversed, complemented with the vertical sequence.

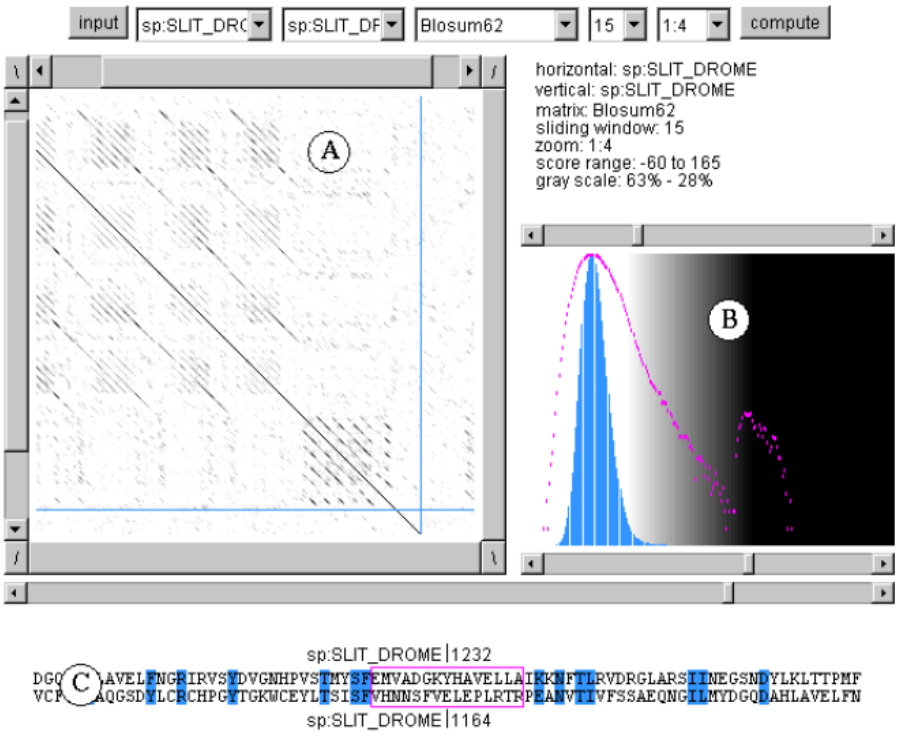

Figure 2 - Dotlet: comprises a main panel (A), a histogram of score sequences (B) and the alignment panel (C).

2.4 Updated Dotlet

With the evolution of time, browsers stopped supporting Java, making the original version of dotlet unusable. This created the need to update the technologies the diagonal plot method is implemented in. The new version of dotlet was developed resorting to React, a Javascript framework that allows the development of a solid web application [15].

This version can be accessed through a web browser and is an improved version of the original dotlet in terms of user-friendliness, while also presenting many of the features the previous version offered (Fig. 3). It is composed with a menu, where one can import the protein or DNA sequences to compare by using the same copy-paste method. Also, one is able to select the scoring matrices and select the window size which will alter the view of the main panel. This main panel, similar to the original version, shows the diagonal plot to the user, where one can see the matches between the compared sequences. In the histogram window we can see the representation of the frequency of each score, over all pixels, on linear (blue) and logarithmic (red) scales. As said above the gray-scale is customizable, allowing the user to make the background noise (low scores) disappear and the similar regions stand out. Under the main panel and histogram, the alignment window displays the sequences present around the cursor position in the main panel. Positively scoring pairs are highlighted with a blue background and a yellow colored letter. The comparison window is encapsulated by red square brackets.

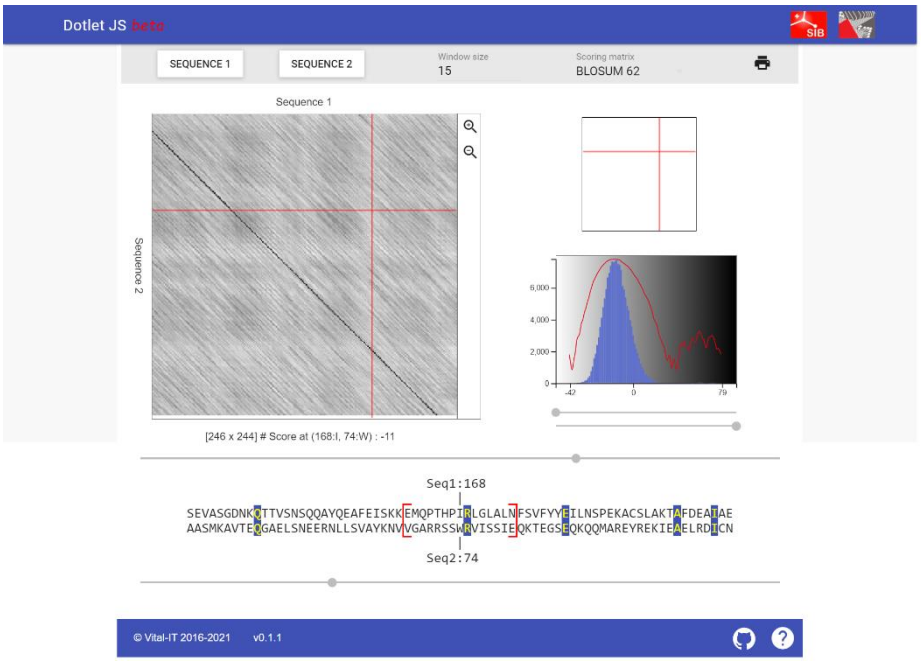

Figure 3 - Updated dotlet: client-side application that comprises all the previous version features with performance improvements and a modern design.

3 Software/Technologies used to develop dotlet

As we headed to the development of the new dotlet platform, we had to decide which technologies we considered would be the best to achieve our goals. This includes which code editor to use, which programming languages were the most suitable, and also which frameworks and packages to use. In this chapter we are going to explain our decision making along with the usage of certain technologies in the previous versions of dotlet.

3.1 Visual Studio Code

Firstly, we chose the code editor named Visual Studio Code due to its support of multiple programming languages and frameworks, the built-in features such as a terminal and git commands, and also because of our previous experience with this editor [16].

Visual Studio Code was developed by Microsoft, presented in 2015 and released to the community in 2016. It has a large and increasing number of extensions, where one can find multiple programming languages, frameworks, formatters, themes, and others. It also provides developers with features such as customizable key bindings, syntax highlighting, automatic indentation, among others. It is also cross-platform, meaning that you can write code in the OS (operating system) of your choice and then execute it in any other OS. This can prove to be useful when trying to check for bugs in your targeted environment.

For this project, we also made use of the built-in git commands in order to share our application updates to the community, resorting to github [17].

3.2 NodeJS and npm

To develop dotlet, there are two main technologies used in order to have a practical environment and all the necessary technologies at hand. These technologies are NodeJS and npm [18, 19].

NodeJS is a javascript runtime environment that allows the developer to write server-side code, being designed to build scalable network applications.

Npm, short for node package manager, is the world's largest software library. It allows developers to share their developed packages under the npm registry, or to borrow other packages. A developer uses the npm website to find packages and then runs commands under the terminal to install them. The most common command is "*npm install*" which installs dependencies present in "*./package.json*" file and saves them in the root folder, under the file "*./node_modules*". If one desires do install a new package for their project, the command to achieve this is "*npm install name_of_package*". Resorting to packages can prove to be an improvement of any developer's application since it could reach the same objective while simplifying one's work.

3.3 HTML, CSS and JavaScript

To develop any web application there are technologies that are crucial to the proper development of the web application.

HTML, short for HyperText Markup Language, is the main language to build web applications, giving them a structure [20]. It contains multiple tags that can be used for specific tasks such as making a form, having a button, among others. It was updated over time, with the latest being named "HTML 5". This language is in almost every case accompanied with CSS, short for Cascading Style Sheets, a stylesheet language used to change the aspect of the web application [21]. It contains properties that aid the developer to style one's website. In addition, JavaScript is a programming language used to make the web applications interactive, converting a static page into an interactive one [22]. Also, JavaScript is used either for front-end or back-end, when accompanied, for example, with NodeJS.

JavaScript contains multiple built-in objects to aid in the development of any web application. In our specific case, we use the async/await keywords in certain functions to enable an asynchronous, promise-based behavior in order to retrieve the intended information. A promise associates handlers with an asynchronous action's possible success or failure. It allows asynchronous methods

to return values similarly to a synchronous method, by returning a promise to later supply the user with the value. A promise is separated into three main states: a pending state, where we wait for the promise to run (initial state); a successful state, where we get the expected value; a rejected state, where we get an error message of the failed operation. Upon receiving either a successful or failed state, the promise's associated methods *then()*, *catch()* or *finally()* are called. The *then()* method is used for either the successful cases or rejected cases. If the value is rejected, the *catch()* method is run. The *finally()* method is used to provide additional information after the promise is complete and also for the code to be run, if the promise is either successful or rejected.

Another JavaScript built-in method we use is the *fetch()* method (GET request), from the fetch API. An API, short for application programming interface, is used for an application to communicate with another without the need for the developer to know how they were implemented [23]. The *fetch()* method is used to fetch information from another link, resorting to the promise behavior.

3.4 React

For the development and improvement of the dotlet platform we resorted to React, a Javascript library used to build single or multiple page web applications. This framework resorts to components which comprise different parts of the web application, as we see in fig 4.

Within the components we first define the class and use a constructor, where we can pass the props (short for properties) and the state. These props and state are both JavaScript objects but are distinct from one another. Props are passed to the component and are similar to the function parameters, whereas state is managed within the component and is similar to the variables declared within a function.

In React we call for props within a function with "*this.props*" and for state with "*this.state*", as both represent the rendered values. One can update the state value by resorting to "*setState*" and, to afterwards display it, one has to use an updater function due to the asynchronous behaviour of "*setState*" calls.

Synchronous tasks are performed one at a time meaning it can result in delayed operations, whereas asynchronous tasks wait for all the components to call "*setState*" in their event handlers before starting to re-render. The asynchronous tasks boost the platform performance since it avoids unnecessary re-renders.

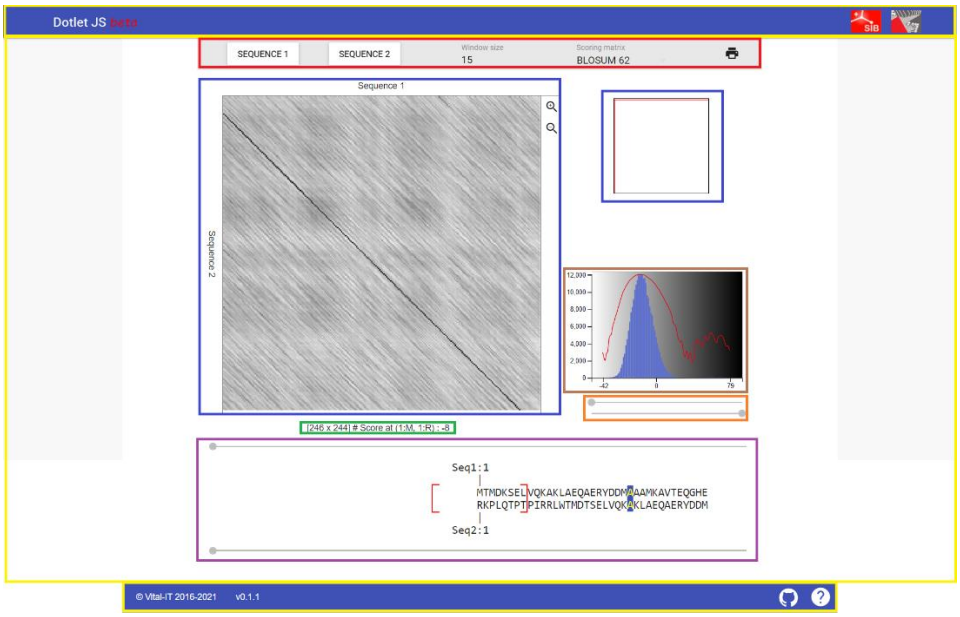

Figure 4 – Dotlet is comprised of multiple components: in yellow we have the layout component; in red we have the input panel component; in blue we have the dotter panel component; in green we have the info panel component; in brown we have the density panel component; in orange we the have greyscale component; in purple we have the two sequences component.

3.4.1 Layout component

The layout component comprises three different zones of the web page, the header, the content of the page, and the footer. The header is where one finds the title of the project along with two links to the Swiss Institute of Bioinformatics (SIB) and Vital-IT, respectively. The content of the page is present in the layout component simply to have a parent component. Lastly, in the footer we find links to the source code, due to this being an open project where everyone can contribute, and also to the documentation.

3.4.2 Input panel component

Shown in figure 4 surrounded by a red rectangle, the input panel component is relative to the area where one can input sequences, select the window size, the scoring matrix, and even print the result of the sequences alignment.

3.4.3 Dotter panel component

This component is divided into different files. In figure 4, the left blue square is in the dotter panel file, and the inner part of the dotter panel, where we see the sequence alignment, is presented in the position lines layer file. The latter one computes the pixel coordinates selected by the user and are presented as a red horizontal and vertical line, as we see in figure 5.

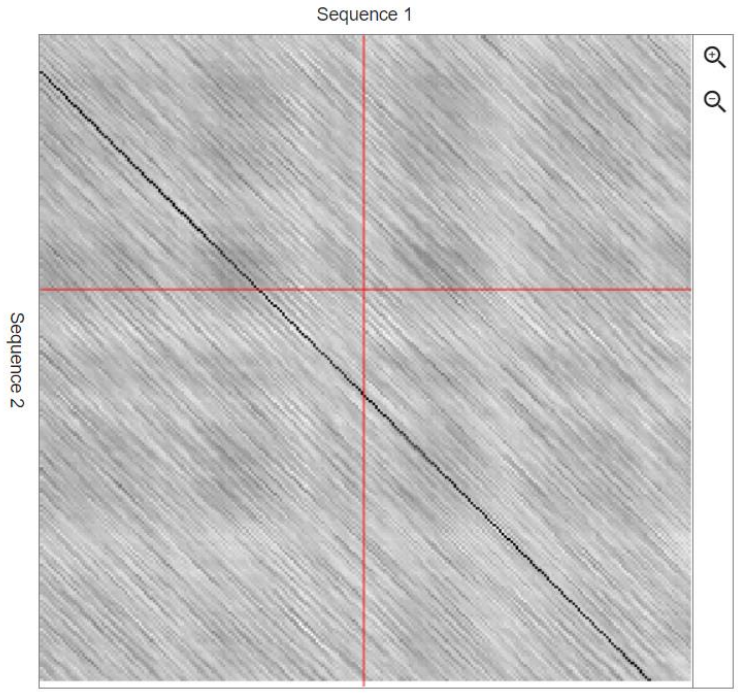

Figure 5 – Dotlet's dot plot method displaying the computed pixel coordinates.

The right blue square comprises the minimap which replicates the pixel coordinates presented in fig.5, but in a dot plot free environment, as we see in figure 6.

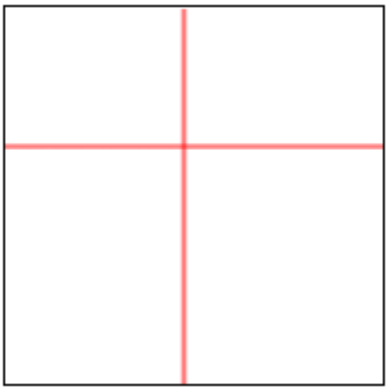

Figure 6 – Dotlet's minimap displaying the computed pixel coordinates.

3.4.4 Info panel component

The info panel component represents the alignment score display of the selected pixel coordinates presented in figure 5 and 6. In figure 7 we can see this panel divised with multiple information. First, we have the **[246 x 244]** mentioning how many letters each sequence has (sequence 1 x sequence 2). Secondly we have **(166:S, 91:Q)** where, for sequence 1, 116 is the current position counting from the original position from the X axis, and S being the letter of the sequence in this position. The same happens for sequence 2, however, the axis we take into consideration is the Y axis. Lastly, we have the alignment score (in this case being **-6**).

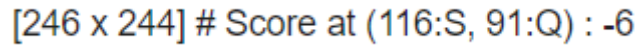

Figure 7 – Dotlet's info panel displaying the alignment score regarding the selected pixel coordinates.

3.4.5 Density panel component

Displayed in figure 4 surrounded in a brown square, we have the density panel component. The objective of the density panel is to display a bar chart where one views the scores density.

3.4.6 Greyscale component

The greyscale component is directly connected to the density panel, being used in that component in order to scale the grey shades and display the intended results.

3.4.7 Two sequences component

This component presents the currently inspected alignment. One can drag the circle present in the grey top and bottom line to change the position of the pixel coordinates. The top grey line represents sequence 1, and the bottom grey line the sequence 2.

3.4.8 Constants

Within the React components we also have the constants, where we store the scoring matrices (BLOSUM, PAM, Identity), taken from Blast 2.2.26. Also, we store other constants to ease the development flow and maintain consistent values.

3.5 Redux store

Redux is a standalone JavaScript library that can be used with multiple frameworks, including React. In our case, we use the React Redux store. This store contains the *state*, *actions* and *reducers*. It holds the whole state tree of the application [24].

The state is an object that contains information about the application. In figure 8 we present an example of default state values present in the dotlet application.

```
const smallExample = {
    s1: "ATGC",
    s2: "ATTAGGCGAGG",
    s1Type: DNA,
    s2Type: DNA,
    scoringMatrix: SCORING_MATRIX_NAMES.BLOSUM80,
    windowSize: 3,
};
```

Figure 8 – Default state example.

The *actions* are objects that have properties that will be used to describe certain actions. In figure 9 we have a function that represents the *actions* mechanism from the existing dotlet application. It takes three values: sequence number (1 or 2), the sequence string, and the sequence type. These will substitute the sequence that is aligned under the specified sequence number.

```
function changeSequence(seqn, sequence, seqtype) {
    return {
        type: CHANGE_SEQUENCE,
        seqn: seqn,
        sequence: sequence,
        seqtype: seqtype,
    };
}
```

Figure 9 – Actions function from dotlet.

There are also *action types*, which are variables set by the developer to be used instead of copying strings, to ensure there is no typo. Since dotlet makes use of Redux, an *action types* file was created, with the variables present in figure 10.

```
const CHANGE_SEQUENCE = "CHANGE_SEQUENCE";
const CHANGE_WINDOW_SIZE = "CHANGE_WINDOW_SIZE";
const CHANGE_SCORING_MATRIX = "CHANGE_SCORING_MATRIX";
const INSPECT_COORDINATE = "INSPECT_COORDINATE";
const CHANGE_GREY_SCALE = "CHANGE_GREY_SCALE";
const RESIZE_CANVAS = "RESIZE_CANVAS";
const OPEN_TOAST = "OPEN_TOAST";
const ZOOM = "ZOOM";
const DRAG_MINIMAP = "DRAG_MINIMAP";
const CHANGE_VIEW_POSITION = "CHANGE_VIEW_POSITION";
```

Figure 10 – Action types variables set in dotlet.

The *reducer* takes an action and the previous state to then update the application with a new state. In dotlet, reducers were collected in a separate source file. For example, to change the sequence state in the store, dotlet resorts to the reducer function presented in figure 11.

```
case CHANGE_SEQUENCE: {
    let newState = Object.assign({}, state);
    let s1, s2;
    let seqtype;
    let seq = action.sequence;
    if (action.seqn === 1) {
        newState.s1 = seq;
        newState.s1Type = action.seqtype;
    } else {
        newState.s2 = seq;
        newState.s2Type = action.seqtype;
    }
    seq = undefined;  // free space
    newState.i = 0; newState.j = 0;
    let ls1 = newState.s1.length;
    let ls2 = newState.s2.length;
    let addToState = updateScores({s1: newState.s1, s2:newState.s2});
    Object.assign(newState, addToState);
    return newState;
}
```

Figure 11 – Dotlet's reducer function to change the sequence state present in the store.

This function creates a "*newState*" variable which contains the *Object.assign()* method that is used to take the new values from the state and assign it to the new target "{ }". It further checks for which sequence number the change is intended, by resorting to "*action.seqn*", and then saving the new values in the "*newState*" variable. Then, the scores are updated according to the new sequence alignment and the "*newState*" is returned.

3.6 LocalStorage

LocalStorage is part of the HTML web storage spectrum, where web applications store data locally, within the user's browser [25]. This storage is designated per domain (e.g., google.com) and protocol (e.g., https). This allows better data management than using cookies, while also allowing for a larger amount of information to be stored. It also does not have any expiration date, allowing the user to come back any time and still have the data stored.

In order to manage the data present in the localStorage, the developer must use a set of built-in functions. To save an item to the localStorage, one must resort to the function "*localStorage.setItem()*". To retrieve an item, the developer must use the function "*localStorage.getItem()*".

To delete an item or every data for that specific website one must use "*localStorage.removeItem()*" and "*localStorage.clear()*", respectively.

3.7 Interaction Git / Github

To receive better insights about the project we decided to resort to git, a version control system designed to handle projects with speed and efficiency. Directly connected to git we have github, the web application where we shared this project and its constant updates [17]. This project is defined as open to the community, meaning anyone can contribute to the improvement of the platform, increasing its value to the scientific community.

Github projects normally are divided in branches where we have the master (main) branch that is the most up-to-date branch, with new branches created where one changes the project code in order to update the master branch state. After we are done coding we open a pull request, which is a way to submit our updates to the master branch where code reviewers can give inputs about one's code.

One who desires to contribute to this project must follow a few steps. Since our project is shared in github, a prospective developer must first clone our project to his or her machine. Then, since the project has a master branch, one must create a new branch to improve the web application. Updating any project within the master branch is considered dangerous because it's harder to track the multiple changes and reverse them if anything is wrongfully changed.

In order to publish each update, we followed a specific guideline. As said above, firstly we created a new branch. This is possible by using the integrated terminal Visual Studio Code has, along with the built-in git commands. The command used for this is "*git checkout -b new_branch_name*". The command "*git checkout*" is used to change between branches, and "*-b*" to create a new branch.

After the developer coded what one foresees as ideal, one submits the changes in the custom branch, so it is saved within the github environment. This is done by adding every file to the git memory with "*git add .*", the dot meaning we want to add every file that was changed or created. Then, one does "*git commit -m"This Is a simple message that explains what was changed""* to store the changes in the github branch, and then one uses "*git push*" to finally submit the changes and present them to the community.

When there is nothing else to add in that branch, one opens a PR (short for pull request), which is a request to update the master branch with the changes done in the custom branch. The PR is

reviewed, and upon approval it is merged with the master branch. Afterwards, the custom branch can be deleted.

3.8 Webpack

Webpack is a module bundler that builds JavaScript applications [26]. When webpack processes the application, a file is created in the root folder. By default the output folder is *./dist* and the default output file is *./dist/main.js*. From this *dist* folder one can host the web application in their machine, launching it in the localhost URL while being in a development mode. Webpack builds the app and NodeJS launches it. If one desires to launch it in production mode the procedure is the same within the server host, but it will be deployed under a designated URL (e.g., https://dotlet.vital-it.ch/).

Coding any application while having it presented in the localhost website allows for a smoother development and correction of errors.

3.9 Electron

Electron is a popular framework that builds desktop applications using HTML, CSS and JavaScript. This tool is known to be behind many popular platforms, including Visual Studio Code. It uses Chromium, an open-source browser project with which google chrome and other web browsers are supported by.

Electron builds cross-platform applications, meaning that one can use the desktop application in any operating system (OS). This is achieved by packaging and building the files with, e.g., electron builder, a npm package.

4 Dotlet development

Dotlet is a platform that was first introduced in 1998, to be used in practical lessons at the Institute of Biochemistry in Switzerland. The program was written by Marco Pagni and Thomas Junier from Vital-IT, with the intention to fulfil the gap of the inexistence of a software that compares sequences by the diagonal plot method. It was developed with the programming language called java, with the intent to be platform-independent, to be run on different operating systems, and also to be run in a web browser. With the continuous technology improvement, web browsers stopped supporting java, which lead to the software becoming obsolete.

In 2016, Julien Delafontaine from Vital-IT presented a new version of dotlet. This version was developed in React and presented only as a web application. This led to a gap in the educational

system, as it's preferable to have a desktop application due to the many challenges present in day-to-day life in school.

Our objective was to improve the already existing web application, along with the development of a desktop application that replicates that same web application. In order to share the state of development, we resorted to github, and within it we divided the work in branches and then submitted them through different pull requests.

4.1 Enhancement of the web page design

The design enhancement of the web application was done throughout the entire development of the application. The purpose of this goal is to make dotlet user-friendlier and to fix previous layout issues.

The first pull request was entirely focused on this goal. Changes were made to the layout component where we added a class to a main tag, in order to be able to style it in the css file. The properties added serve the objective of making the webpage homologous, by changing the sidebars, as we see in figure 12.

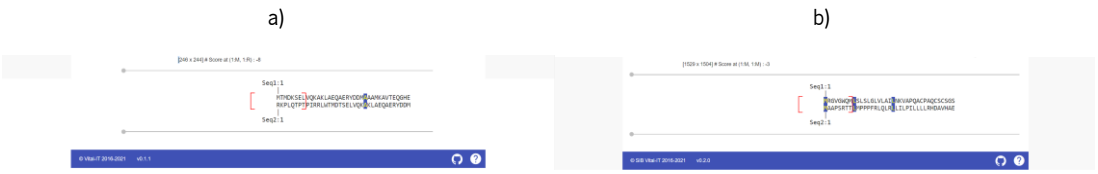

Figure 12 – Layout comparison of the sidebar from the previous (a) and newer (b) version of dotlet.

Further layout improvements were made in order to make the web application responsive. A responsive web application means it accommodates different screen sizes, being particularly important in the scientific community since many people uses smaller screen machines to fit better in the laboratory environment. This was accomplished by resorting to media queries, where we designated what properties must change at a certain monitor width.

Changes to the mid panel, which contains the dotter panel and density panel, and to the two sequences panel were also made with the intent to improve the users' experience. The width of these panels was increased in order to take advantage of the space available and allow the responsiveness to be enhanced. Within the dotter panel, the dot plot Y axis label had a 180 degree turn in order to make it more legible. Also, the minimap constant was increased, resulting in a clearer pixel coordinates display. In the two sequences panel, the grey lines were also extended to allow a more precise selection of the pixel coordinates one intends to select. These changes are presented in figure 13.

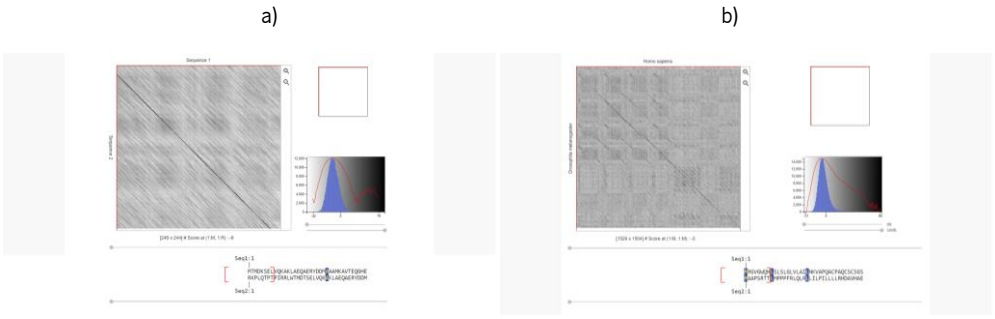

Figure 13 – Previous dotlet layout (a) compared with the updated dotlet layout (b) of the mid panel, that is composed with the dotter and density panel, and the two sequences panel.

To improve the user's understanding of the web application and since they are more interesting due to the existence of repeats, we decided to change the default sequences present in the "*sequence 1*" and "*sequence 2*" slots. In the "*sequence 1*" input we used a homo sapiens sequence, which we can find in the sequences database UniProt under the code "*O94813*". For the "*sequence 2*" input we used a *Drosophila melanogaster* sequence, which we can find in UniProt with the code "*P24014*".

For better readability we decided to change two aspects in the web application. Firstly, we replaced the previous font-family used in the sequence input for a more widespread and monospaced one, called "Courier New". Then, as seen in figure 13, we changed the Y axis of the dotter panel, rotating it 180 degrees by resorting to the "*rotate(180deg)*" css function.

We also added a percentage that accompanies the greyscale component sliders (fig. 14). To calculate the percentage display, we used a built-in object called "*Math*" and its function "Math.*round()*", with the local state of the minimum and maximum bound being multiplied with the subtraction of the width with the maximum bound.

```
let minPercentage = Math.round(this.state.minBound * (100/255))+'%';
let maxPercentage = Math.round(this.state.maxBound * (100/255))+'%';
```

Figure 14 – Variables that are used for the percentage display in the greyscale component.

To display the minimum percentage we make use of a "*textarea*" tag, as we can see in figure 15. A similar tag is used for the maximum percentage.

```
<textarea className={s.sliderPercentage} value={minPercentage} onChange={this.onChangeMinBound} disabled/>
```

Figure 15 – HTML tags for the minimum percentage display in the greyscale component.

4.2 Dependencies issues

Due to the nature of dotlet being launched in 2016 and the technology evolution, the npm packages selected to this web application were deprecated (outdated). This issue led to some key errors in

the webpack build of the application that interrupted it entirely, leaving our application unable to be deployed.

With this issue in mind, updates to a few packages were done. We firstly updated mocha, a framework that runs on node.js, responsible for testing web applications. This testing is a way to replicate users' actions and test if one gets the response they were expecting. If the test passes it means we got the expected response, if not one needs to either check if the test is being done right or something needs to change in the application code.

Then we decided to exclude a deprecated package, which is node-sass, that is responsible for compiling the stylesheet preprocessor files, scss, to css [27]. We replaced this package with another, gulp-sass, which is a plugin for gulp [28].

Lastly, we updated the material-ui, a library that allows its users to import and use different components, so one can be more efficient in the development of a web application [29]. At first we tried to update it to the most recent version, which is v4.12.3, but it was incompatible with the React version dotlet is using. Therefore, we decided to update it under the same major version (v0), to 0.20.2.

Even though we updated some of the packages to fix the build, we decided to remove the public folder (where the webpack build is placed) from the *gitignore* file, a file which git explicitly ignores and usually contains sensitive information.

4.3 Load sequences through an external URL

To input sequences with an URL we decided that only with UniProt links this would be possible, since we have previously worked with this database and because of the different types of information any link can have.

This database stores each sequence in *fasta* files, under a UniProt identifier. To fetch the *fasta* file information we wrote an async/await function. This function fetches (GET request) the *fasta* file information, consisting in a *try* and *catch* block around the asynchronous function, as we see in figure 16.

```
async fetchFasta(url) {
    // GET request using fetch with async/await
    try {
        if (validURL(url) == true) {
            const response = await fetch(url);
            const data = await response.text();
            const data2 = validateFasta(data);
            return data2;
        }
    }
    catch (err) {
        throw err;
    }
}
```

Figure 16 – Async/await function to fetch the fasta file contained in the URL that is passed as an argument.

Our asynchronous function consists of two blocks, the *try* and *catch* blocks. The *try* block contains an if statement where we check if the URL passed in the argument is valid (fig. 17), by resorting to a regular expression, where we first define the search pattern to check if the URL contains the same pattern. We then test the URL to see if it starts with the common protocol, returning a Boolean value (true or false).

```
function validURL(str) {
    var pattern = new RegExp('^(https?:\\/\\/)'); // protocol
    return !!pattern.test(str); // return true or false
}
```

Figure 17 – Function that validates if the URL starts with a web protocol (http or https).

If the return value from the validation of the URL is true, then we fetch the *fasta* file resorting to a GET request, which returns a promise that resolves to the response to that request. Once we have the response, we store the file's text in a const variable to then validate it and return its value. To validate the *fasta* file we wrote a function (fig. 18) that firstly checks if the file contains any value, then we remove any whitespaces from both ends of the string, split the lines and check, in first line (position zero), for the common header a *fasta* file presents. If there is a header, then we remove it and continue with the next line (first position). We then remove the newlines and any whitespaces, joining the array into a single string. Lastly, if the *fasta* file is empty then it returns false, and by resorting to regular expressions we check if the string belongs either to a protein or DNA, to then return the sequence.

```
function validateFasta(fasta) {

    if (!fasta) { // checks if there is something first of all
        return false;
    }

    // immediately remove trailing spaces
    fasta = fasta.trim();

    // split on newlines
    var lines = fasta.split('\n');

    // check for header
    if (fasta[0] == '>') {
        // remove one line, starting at the first position
        lines.splice(0, 1);
    }

    // join the array back into a single string without newlines and
    // trailing or leading spaces
    fasta = lines.join('').trim();

    if (!fasta) {
        return false;
    }

    // note that the empty string is caught above
    // allow for Selenocysteine (U)
    if (/^[ACDEFGHIKLMNPQRSTUVWY\s]+$/i.test(fasta) == true) {
        return (fasta);
    };
}
```

Figure 18 – Function that validates if the text contained in the fasta file belongs either to a protein or DNA.

The second block of the *fetchFasta* (fig. 16) function is the *catch* block, to handle the URLs that are rejected from the *try* block. In this case we throw the error as being the failed URL, which will maintain the rejected URL in the input slot until the user changes the URL into a valid one, or by copy-pasting the intended sequence.

Upon fetching the sequence, we wrote a function to replace the value of the sequences input. To do this, we resorted to an async/await function as we see in figure 19.

```
onChangeSeq1 = (e) => {
    let s1 = e.target.value;
    if (validURL(s1) == true) {
        const getData = async () => {
            const sequence = await this.fetchFasta(e.target.value);
            s1 = sequence;
            s1Type = guessSequenceType(s1, 200);
            this.setState({ s1 });
            store.dispatch(changeSequence(1, s1, s1Type));
        };
        getData();
    }
    let s1Type = guessSequenceType(s1, 200);
    this.setState({ s1 });
    s1 = formatSeq(s1);
    let isValid = validators.isValidInputSequence(s1, s1Type);
    if (isValid.valid) {
        store.dispatch(changeSequence(1, s1, s1Type));
    }
    if (!isValid.valid && !validURL(s1)){
        store.dispatch(openToast("Invalid "+ s1Type +" sequence character '"+ isValid.wrongChar +"'"));
    };
};
```

Figure 19 – Function to replace the input URL with the sequence contained in the fasta file.

Firstly, we validate if the URL that was placed in the sequence input location. Then, we created a function with the asynchronous method, where we create a variable that stores the value that is returned from the *fetchFasta* function (fig. 16) and utilize previously written functions to determine the sequence type (fig. 20) and change the sequence. The latter one replaces the sequence in the redux store. By setting the state, we are replacing the URL with the sequence contained in the *fasta* file. Then, we have to run the function in order to make it work as expected, since we are resorting to an asynchronous method.

```
function guessSequenceType(seq, nchars=200) {
    let L = seq.length;
    // The longest known protein, Titin, has up to 33K aminoacids.
    if (L === 0 || L > 40000) {
        return DNA;
    }
    // Check the first N characters. If they are all ATGCU, conclude it is DNA.
    // Check up to the size of a long protein (~1000).
    // The original code went through the whole sequence and said it is DNA if more than 80% is ATGCU.
    let nucleotides = new Set(['A','T','G','C','U']);
    let N = Math.min(nchars, L);
    for (let i=0; i<N; i++) {
        if (! (nucleotides.has(seq[i]))) {
            return PROTEIN;
        }
    }
    return DNA;
}
```

Figure 20 – Function that determines the type of the sequence (protein or DNA).

The function presented in figure 20 takes the input sequence and determines if this is a DNA or a protein sequence. It takes the length of the sequence and the number of testing characters, so

it is time efficient. If the sequence presents only ATGCU nucleotides in the first 200 testing characters, we conclude it is a DNA sequence. If this is not the case, the sequence is a protein.

4.4 Save the list of loaded sequences

To create a list of saved input sequences for the user to later change which ones to compare, we decided to use the *localStorage* since it stores data across browser sessions with no storage time limitation. In our web application we needed four types of data management. The first, to save the sequences, the second to get the sequences, the third to remove a specific sequence from *localStorage*, and the last one to remove every sequence from the *localStorage*. For the first type we wrote a function that saves the input and default sequences (fig. 21).

```
setDatastorage() {
    if (Object.keys(localStorage).includes(this.state.seqName)) {
        alert("Sequence name already taken.")
    }
    if (Object.values(localStorage).includes(this.state.activeSequence === 1 ? store.getState().s1 : store.getState().s2)) {
        alert("The sequence you're trying to save already exists.")
    }
    else {
        localStorage.setItem(this.state.seqName, this.state.activeSequence === 1 ? store.getState().s1 : store.getState().s2)
        if (this.state.activeSequence === 1) {
            this.onChangeSeq1custom(store.getState().s1)
        }
        if (this.state.activeSequence === 2) {
            this.onChangeSeq2custom(store.getState().s2)
        }
        alert("The sequence " + this.state.seqName + " has been added.")
        this.forceUpdate()
    }
}
```

Figure 21 – Function to save the input and the default sequences into the localStorage.

In this function we check if a name is already taken within the *localStorage*, and also if a sequence already exists. If any of these already exist in the *localStorage*, an alert will be presented to the user so one is aware and then is able to change the sequence itself or the sequence's name. If neither of these scenarios occur, then the sequence is saved in the *localStorage*, and presented to the user in a table that will be presented further on. It also automatically changes the sequences presented to the user in the dotter panel.

After the user saves the sequences, they must be retrieved from the *localStorage*. To do this, we wrote another function, presented in figure 22.

```
getDatastorage() {
    var archive = [],
    keys = Object.keys(localStorage),
    i = 0, key;

    for (; key = keys[i]; i++) {
        archive.push( 'Sequence name: ' + key + ' \n ' + localStorage.getItem(key) + '\n');
    }

    var mappedArchive = archive.map((item, i) => {
        var values = Object.values(localStorage)[i]
        return (
        <div className={s.storageDiv}>
            <button className={s.changeSequencesButton} onClick={() => this.onChangeSeq1custom(values)} data-title={"Set sequence 1"}>Sequence 1</button>
            <button className={s.changeSequencesButton} onClick={() => this.onChangeSeq2custom(values)} data-title={"Set sequence 2"}>Sequence 2</button>
            <a data-title="Remove sequence">
                <img className={s.removeSequence} onClick={() => this.removeDatastorage(keys[i])} src={require("../../public/images/remove_icon.svg")}/>
            </a>
            <li className={s.savedSequences}>{item}</li>
        </div>
        )
    });

    return mappedArchive;
}
```

Figure 22 – Function to retrieve the saved sequences present in the *localStorage*.

In this function we store a list in a variable named "*archive*", where we will save each sequence, and the *localStorage* keys in another variable named "*keys*". The *localStorage* acts like a dictionary, with keys and values, the keys being the sequence names and the values the sequence itself. We iterate through the *localStorage* in order to get every sequence information, by using the "*getItem*" method, and then save this information in the archive.

Following the archive storage of the sequences, we iterate through the archive using *map* in order to go through each element and store the new array created by it, in the "*mappedArchive*" variable. Then, connected to each *localStorage* key, we get its value and store it in a variable. Lastly we return a set of HTML tags from where one can select the sequence to be aligned, remove a certain sequence from the *localStorage*, and view all the sequences the user has previously saved.

In figure 19 we also have the "*onClick()*" JavaScript event relative to the replacement of the sequences. The function behind this event is presented in figure 23, where it takes the selected sequence as an argument and then sets the state to present that sequence. There is a similar function for sequence 2.

```
onChangeSeq1custom(values) {
    let s1 = values;
    if (validURL(s1) == true) {
        const getData = async () => {
            const sequence = await this.fetchFasta(values);
            s1 = sequence;
            s1Type = guessSequenceType(s1, 200);
            this.setState({ s1 });
            store.dispatch(changeSequence(1, s1, s1Type));
        };
        getData();
    }
    let s1Type = guessSequenceType(s1, 200);
    this.setState({ s1 });
    s1 = formatSeq(s1);
    let isValid = validators.isValidInputSequence(s1, s1Type);
    if (isValid.valid) {
        store.dispatch(changeSequence(1, s1, s1Type));
    }
    if (!isValid.valid && !validURL(s1)){
        store.dispatch(openToast("Invalid "+ s1Type +" sequence character '"+ isValid.wrongChar +"'"));
    };
};
```

Figure 23 – Function to replace the aligned sequence with the selected one.

The removal of a sequence is done with a simple function, displayed in figure 24. It gets the argument (key) from the selected sequence and compares it with the sequences present in the state, which are the ones currently in the alignment. If they are the same sequence, then we replace the sequence in the alignment with the default one. Then we apply the "*removeItem*" method to remove that specific sequence from the *localStorage*.

```
removeDatastorage(key) {
    if (localStorage.getItem(key) == store.getState().s1) {
        this.onChangeSeq1custom("MRGVGWQMLSLSLGLVLAILNKVAPQACPAQCSCSGSTVDCHGLALRSVPRNIPRNTERLDLNGNNITRIT
    }
    if (localStorage.getItem(key) == store.getState().s2) {
        this.onChangeSeq2custom("MAAPSRTTLMPPPFRLQLRLLILPILLLLRHDAVHAEPYSGGFGSSAVSSGGLGSVGIHIPGGGVGVITEA
    }
    localStorage.removeItem(key)
    this.forceUpdate()
}
```

Figure 24 – Function to remove the selected sequence from the localStorage.

To remove all the sequences from the *localStorage* we used a different method, called "*clear*" (fig. 25). Although this method removes every key from the storage, as said above, we force the default sequences to be added upon each page re-render, which is what "*forceUpdate*" does. Upon the removal of all the sequences, we replace each sequence that was in the alignment with the default ones.

```
removeAlldatastorage() {
    localStorage.clear()
    this.onChangeSeq1custom("MRGVGWQMLSLSLGLVLAILNKVAPQACPAQCSCSGSTVDCHGLAL
    this.onChangeSeq2custom("MAAPSRTTLMPPPFRLQLRLLILPILLLLRHDAVHAEPYSGGFGSS
    this.forceUpdate()
}
```

Figure 25 – Function to remove all the saved sequences from the localStorage.

After writing functions that deal with the sequence storage, we had to display these sequences. For that we first added two variables, "*openStorage*" "*seqName*" to the local state as seen in figure 26, to have default values that later will be replaced.

```
this.state = {
    open: false,
    openStorage: false,
    seqName: 'Undefined name',
    activeSequence: 1,
    s1: storeState.s1,
    s2: storeState.s2,
    windowSize: storeState.windowSize,
    scoringMatrix: storeState.scoringMatrix,
};
```

Figure 26 – Component local state composed with default values.

The "*openStorage*" is used to have the sequence storage container closed when the page is loaded. To open it, we wrote a function that sets the state to the opposite value (fig. 27). With this, each time the storage button is clicked, it will always change to the opposite value, either opening or closing it.

```
openSeqarea() {
    this.setState({
        openStorage: ! (this.state.openStorage),
    });
}
```

Figure 27 – Function that changes the default value of the "openStorage" variable.

Upon opening the storage area, the sequences are displayed accompanied by each button to select a certain sequence to be aligned with another, to remove a specific sequence, and also to remove all the sequences the user had previously saved. These are rendered resorting to the HTML tags presented in figure 28.

```
<div style={{display: this.state.openStorage ? 'block' : 'none'}} className={s.sequencesDisplayContainer}>
    <ul className={s.sequencesDisplay}>{this.getDatastorage()}</ul>
    <button className={s.storageButton}
        onClick={() => this.removeAlldatastorage()}
        ><img className={s.removeAllSequences} src={require("../../public/images/trash_icon.svg")}/>
            Remove all sequences
    </button>
</div>
```

Figure 28 – HTML tags to display the sequence storage.

The button that opens this storage area is defined by resorting to the material-ui components named "*Toolbar*", the "*ToolbarGroup*", and "*RaisedButton*". These will essentially create a button with a defined label that is binded to the "*openSeqarea*" function and to the "*openStorage*" state, as seen in fig. 29.

```
<ToolbarGroup>
    <RaisedButton onClick={this.openSeqarea.bind(this)}
        secondary={this.state.openStorage} label="Saved Sequences"/>
</ToolbarGroup>
```

Figure 29 – Material-ui components' tags that create the button and bind it to the "openSeqarea" function.

To save the sequences we decided that one should be saved with a custom name, e.g. Homo Sapiens, chosen by the user. Firstly we wrote a function that, according to the name the user wrote, sets its state, changing the local state variable (fig. 30).

```
onChangeSeqName = (e) => {
    let seqName = e.target.innerText;
    this.setState({ seqName });
};
```

Figure 30 – Function that sets the state for the input sequence name.

To display the area where the user writes the sequence name, we first used a "*textarea*" HTML tag. Since it was not working as intended, due to an error with the sequence name display as a placeholder, and was taking a lot of time to figure out the issue, we decided to use a "*div*" tag and make it act as a "*textarea*" tag, since a "*div*" tag is highly customizable (fig. 31). We used the "*contentEditable*" along with "*onInput*" properties to achieve this. We also present to the user the aligned sequence names in the sequence name input location, for them to better understand which sequence they are replacing.

```
<div contentEditable='true' className={s.divtextarea}
    placeholder={"Sequence name: "}
    onInput={this.onChangeSeqName}
    style={{fontFamily: 'Courier New'}}>
        {this.state.activeSequence === 1 ? sequenceNames.getSeq1Name() : sequenceNames.getSeq2Name()}
</div>
```

Figure 31 – HTML "div" tag for the user to input a sequence name.

To have the currently aligned sequence's name displayed in the sequence name input location, we used a function from the dotter panel component. This was done with "*const sequenceNames = new DotterPanel()*". Then we are able to use the function that grabs the sequence names by using *"sequenceNames.getSeq1Name()*" for example, as seen in figure 31.

The input sequence area is presented to the user by using a "*textarea*" html tag (fig. 32), that is common to each sequence input, either to sequence 1 or 2. This is possible by resorting to the local state, where we check which form is activated (since we have a separated form to each sequence).

```
<textarea className={s.textarea} rows={3} ref={(c) => this._textArea = c}
    value={this.state.activeSequence === 1 ? this.state.s1 : this.state.s2}
    placeholder={this.state.activeSequence === 1 ? 'Sequence 1:' : 'Sequence 2:'}
    onChange={this.state.activeSequence === 1 ? this.onChangeSeq1 : this.onChangeSeq2}
    style={{fontFamily: 'Courier New'}}
/>
```

Figure 32 – HTML "textarea" tag for the user to input the sequences.

To save the sequence, the user is presented with a button (fig. 33). Upon a user click to the button, it calls the "*setDatastorage*" function explained above (fig. 21) and saves the sequence in the *localStorage*. It is then presented to the user in the saved sequences table.

```
<button className={s.storageButton}
    onClick={() => this.setDatastorage()}
    ><img className={s.saveSequences} src={require("../../public/images/save_icon.svg")}/>
        Save Sequence
</button>
```

Figure 33 – HTML "button" tag for the user to save a sequence.

Upon any render of the web application, we check for the default sequences within the *localStorage*. This is done with an "*if*" statement (fig. 34), where we check if there are equal or less than one key (sequences number of entries), and if the keys (sequence's names) with the specific default names are present in the *localStorage*. If any of these are confirmed, then the sequence missing is added.

```
if (Object.keys(localStorage).length <= 1 ||
    !Object.keys(localStorage).includes(localStorage.getItem("Homo sapiens")) ||
    !Object.keys(localStorage).includes(localStorage.getItem("Drosophila melanogaster"))) {
    localStorage.setItem("Homo sapiens", "MRGVGWQMLSLSLGLVLAILNKVAPQACPAQCSCSGSTVDCHGLALRSVPRNIPRNTERLDLNGNNI
    localStorage.setItem("Drosophila melanogaster", "MAAPSRTTLMPPPFRLQLRLLILPILLLLRHDAVHAEPYSGGFGSSAVSSGGLGSV
}
```

Figure 34 – "If" statement to check for the default sequences within localStorage.

Upon setting a sequence storage with *localStorage*, we were able to do a small yet important change in the dotter panel. As we can see in figure 15, we have different names in the Y and X axis. We changed the default names so the user easily identifies which sequence is present in either axis, and in the alignment. We achieved this by writing two functions (fig. 35), one for sequence 1 and for sequence 2.

```
getSeq1Name() {
    var archive = [],
    keys = Object.keys(localStorage),
    i = 0, key;

    for (; key = keys[i]; i++) {
        archive.push( 'Sequence Name: ' + key + ' \n ' + localStorage.getItem(key) + '\n');
    }

    var mappedArchive = archive.map((item, i) => {
        var values = Object.values(localStorage)[i]
        for (; values.includes(store.getState().s1); i++) {
            return Object.keys(localStorage).find(key => localStorage[key] === values);
        }
    });

    if (!Object.values(localStorage).includes(this.state.activeSequence === 1 ? store.getState().s1 : store.getState().s1)) {
        return "Sequence 1"
    }

    return mappedArchive;
}
```

Figure 35 – Function that searches the sequence name within the localStorage to be presented in the dotter panel.

This function searches for the keys (names) of each sequence, in the *localStorage*. It gets the state of the sequence that's currently in the alignment and checks for its name within the *localStorage*, returning it. If the sequence that is currently being aligned is not present in the *localStorage*, then it will present the default name, which is "*Sequence 1*". The function for the second sequence is very similar to the one present in figure 35, only changing the "*s1*" to "*s2*" and the default name "*Sequence 1*" to "*Sequence 2*".

This sequence alignment naming is presented to the user by resorting to the functions that are explained above (fig. 36).

```
<div className={s.legendX}>{this.getSeq1Name()}</div>
<div>
    <div className={s.legendY}><div style={{transform: "rotate(180deg)"}}>{this.getSeq2Name()}</div></div>
```

Figure 36 – HTML tags to present the sequences name in the dotter panel.

4.5 Create a stand-alone version that may run offline for educational purposes

Developing a desktop application for dotlet, that replicates its web application environment, was an important goal of the dissertation, so it could be used without internet connection. In order to achieve this, we considered resorting to docker, a platform to containerize applications. This ultimately allows for applications to be stored in a container which can then be shared with the community. Despite it being a good option, it requires internet access to open the containers. Therefore, we decided to use electron, a package which can turn any web application into a desktop application, thus only requiring internet access for the developers. We believed this package was easier to setup and to share with the educational community, if there is no internet access.

We firstly read electron documentation to understand how to set up the package within the dotlet project. Then, we installed the *npm* packages "*electron*", "*electron-builder*" and "*electron-packager*". The first one is to build the project in order for it to be built by the following two, with which we build the desktop application for the different OS.

To set it up, we added a few lines in the "*package.json*" file to connect the package with the project files.

```
"main": "public/electron-starter.js",
```

Figure 37 – Route used for the electron setup.

In figure 37 we have the connection to the electron main file, where we have the main process code that creates the environment of the desktop application (fig. 38).

```
function createWindow () {
  // Create the browser window.
  mainWindow = new BrowserWindow({width: 1300, height: 800})

  // and load the index.html of the app.
  mainWindow.loadFile(isDev ? 'http://localhost:3000' : `public/index.html`);

  // Open the DevTools.
  //mainWindow.webContents.openDevTools()

  // Emitted when the window is closed.
  mainWindow.on('closed', function () {
    // Dereference the window object, usually you would store windows
    // in an array if your app supports multi windows, this is the time
    // when you should delete the corresponding element.
    mainWindow = null
  })
}
```

Figure 38 – Function to open the desktop application.

The function above creates the browser window that presents to the user the web application in the localhost. It also presents the user with the desktop window where it takes the main file from the public folder. This file is responsible for connecting the files that the webpack built to be launched in production (fig. 39).

```
<html class="no-js" lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="x-ua-compatible" content="ie=edge">
    <title>Dotlet JS</title>
    <meta name="description" content="Dotlet JS">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Roboto:300,400,400italic,500,500italic,700,700italic">
    <link rel="stylesheet" href="https://fonts.googleapis.com/icon?family=Material+Icons">
    <link rel="stylesheet" href="https://cdn.rawgit.com/tleunen/react-mdl/master/extra/material.min.css">
    <link rel="stylesheet" href="https://cdn.rawgit.com/isagalaev/highlight.js/master/src/styles/default.css">
    <link rel="apple-touch-icon" href="/apple-touch-icon.png">
  </head>
  <body>
    <div id="container"></div>
    <script src="https://cdn.rawgit.com/tleunen/react-mdl/master/extra/material.min.js"></script>
    <script src="./dist/main.js?36efd1e993a1000a18e3"></script>
  </body>
</html>
```

Figure 39 – ".public/index.html " file that connects the project files for production.

For the app to be launched there are a few lines of code that are required (fig. 40), to make sure that the app quits after the windows close, or that it opens when there are no windows with the app already open.

```
app.on('ready', createWindow)

// Quit when all windows are closed.
app.on('window-all-closed', function () {
  // On OS X it is common for applications and their menu bar
  // to stay active until the user quits explicitly with Cmd + Q
  if (process.platform !== 'darwin') {
    app.quit()
  }
})

app.on('activate', function () {
  // On OS X it's common to re-create a window in the app when the
  // dock icon is clicked and there are no other windows open.
  if (mainWindow === null) {
    createWindow()
  }
})
```

Figure 40 – Default setup for the electron app to be opened and closed.

To display the electron app in the browser or in the desktop application, we wrote a few scripts (fig. 41). The "*electron*" script only open dotlet in the browser, whereas the "*electron-dev*" open dotlet in both the browser and in the desktop application. The "*pack*" is used to generate the package directory without packaging it, with its purpose being for testing. The "*dist*" script is used to package the application in a distributable format, with the application installer file. The "*package*" scripts make use of the "*electron-packager*", distributing the application to either the designated operating systems with the specific architecture, or to all the OS and respective architecture available.

```
"electron": "electron .",
"electron-dev": "concurrently \"BROWSER=none npm run start\" \"wait-on http://localhost:3000 && electron .\"",
"pack": "electron-builder --dir",
"dist": "electron-builder",
"package-mac": "electron-packager . --overwrite --platform=darwin --arch=x64 --icon=build/icon.icns --prune=true --out=release-builds",
"package-win": "electron-packager . dotlet --overwrite --asar=true --platform=win32 --arch=ia32 --icon=build/icon.ico --prune=true --out=release-builds --version-string.Co
"package-linux": "electron-packager . dotlet --overwrite --asar=true --platform=linux --arch=x64 --icon=assets/icons/png/1024x1024.png --prune=true --out=release-builds",
"package-all": "electron-packager . --overwrite --all --icon=build/icon.icns --prune=true --out=release-builds"
```

Figure 41 – Scripts used for the development and distribution of dotlet as a desktop application.

All of the scripts above can be run by resorting to the console and typing, e.g., "*npm run dist*". Upon running the previous example, a folder is created in the root directory with the desktop application and its dependencies within it.

To setup *electron-builder* and *electron-packager*, such as the build configuration and target formats for macOS, linux and windows (fig. 42). The macOS target format is *dmg*, for linux is *deb*, and for windows it is *nsis*.

```
"build": {
  "extends": null,
  "appId": "com.example.app",
  "directories": {
    "buildResources": "build"
  },
  "files": [
    "node_modules/**/*",
    "public/**/*"
  ],
  "dmg": {
    "contents": [
      {
        "x": 110,
        "y": 150
      },
      {
        "x": 240,
        "y": 150,
        "type": "link",
        "path": "/Applications"
      }
    ]
  },
  "linux": {
    "target": [
      "AppImage",
      "deb"
    ]
  },
  "win": {
    "target": "NSIS",
    "icon": "build/icon.ico"
  },
  "nsis": {
    "installerIcon": "build/icon.ico",
    "installerHeaderIcon": "build/icon.ico",
    "deleteAppDataOnUninstall": true
  }
}
```

Figure 42 – Scripts used to build dotlet as a desktop application.

After setting up the application and launching it in the browser, it was working as intended, but when we launched it as a desktop application we got a 404 error. This happens because, even though React has provided the files correctly, it automatically redirects the "*public/index.html*" file to a single "*public/*". Since the "*index.html*" file connects all the files of the application, it breaks when the redirect happens.

In the 404 error page we have a link to take us to the home page, and upon clicking in the link, it opens the dotlet page as intended. So, as a work around, we decided to write the function present in figure 43.

```
// Function called as a work around, to make electronjs work as intended
autoClick() {
  window.onload = function(){
    document.getElementById('homePage').click();
  }
}
```

Figure 43 – Function to automatically click in the homepage upon error page load.

This function is intended to automatically click in the home page link, which takes us to the dotlet page. Upon the error page load, we get the id of the home page link (fig. 44) and use the "*click()*" JavaScript event on it.

```
<Link to="/" id={'homePage'} onClick={this.autoClick()}>home page</Link> to choose a new direction.
```

Figure 44 – Homepage link tag with the embedded automatic click function.

In the figure above, we present the link to take the user to the home page, while also passing the automatic click function to it.

5 Workflow

Dotlet was developed to use scoring matrices on aligned sequences that are submitted by the user. Despite this application being user-friendly, one could first be introduced to it in order for it to be easier and faster to achieve the intended results. In this chapter we are going to present the updated application by showing possible workflows the user could perform.

5.1 Dotlet as a web application

The newest version of dotlet, developed in this project, was a major update to the previous version, presenting to the user new features and improving one's work with the application. We can view this newest version in figure 45.

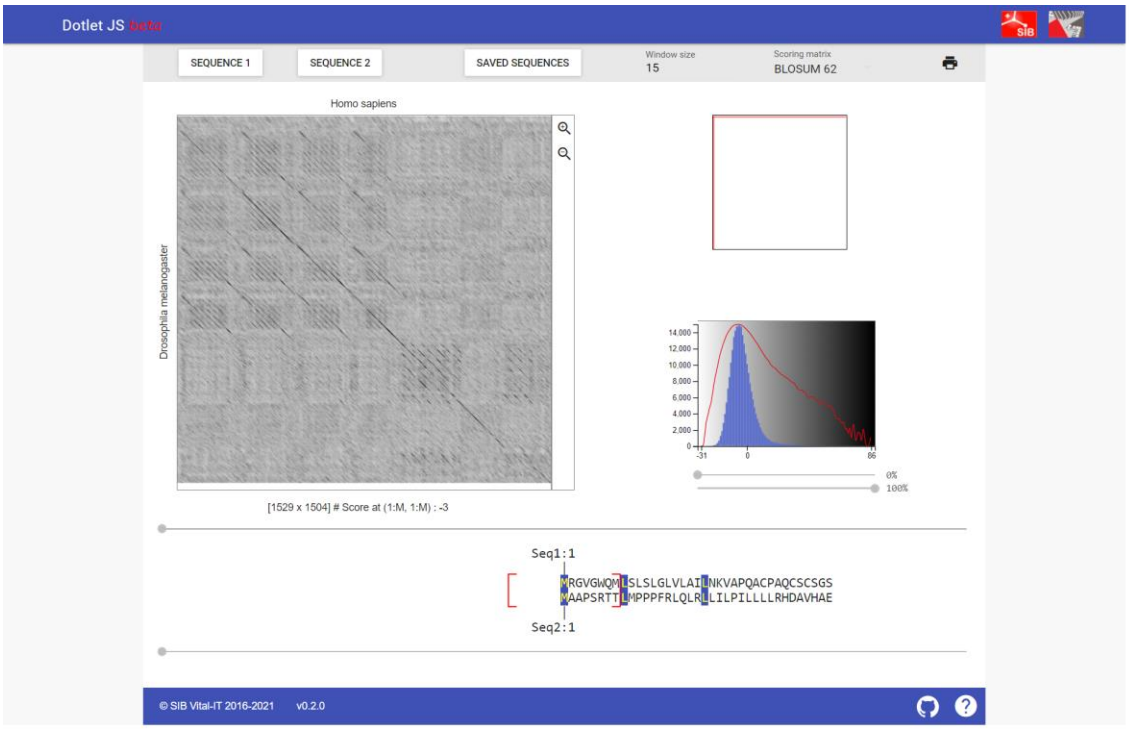

Figure 45 – The newest dotlet version, as a web application.

Upon loading the website, the user is presented with the header (fig. 46). This contains the title of the application on the left corner, and the Swiss Institute of Bioinformatics (SIB) logo along with the Vital-IT logo on the right side. Clicking in any of the logos will take the user to either the SIB or to the Vital-IT website.

Figure 46 – Dotlet's header, presenting the application title and the logos of the associated companies.

Then, in the main layout of the web application, the user can view the input panel, where one views different buttons. Selecting the "*sequence 1*" button will open the sequence input area (fig. 47), where one can view the currently aligned sequence that is displayed in the X axis of the dotter panel. The selected button presents a background red color.

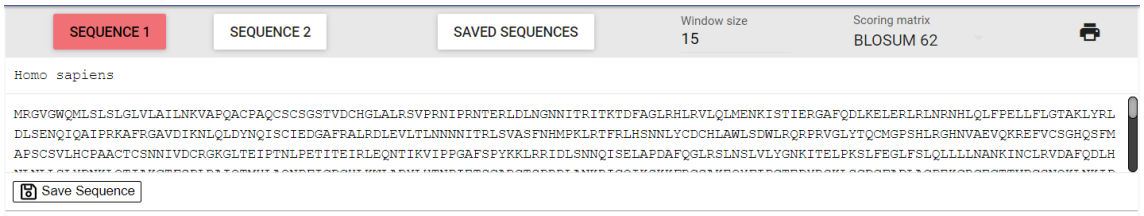

Figure 47 – Sequence 1 input panel.

Within this sequence input panel the user can replace the current sequence with another. Firstly, one should have either the sequence or the link from the UniProt database that contains the sequence fasta file, e.g. "*https://www.uniprot.org/uniprot/E5G0U9.fasta*". Then, in the first text area, the user writes the name for the new sequence. Afterwards, one pastes the link or the

sequence itself in the text area below. Lastly, there is a button in the lowest part of this panel, with which the user can save the inputted sequence in the localStorage. The localStorage is specific to the browser and the user's machine. The "*sequence 2*" button works the same way, as we can see in figure 48.

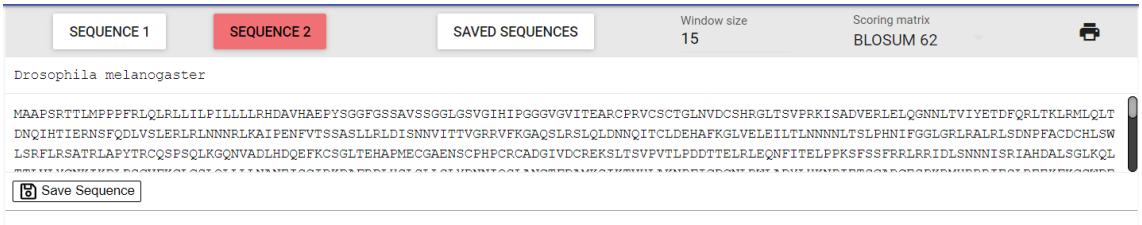

Figure 48 – Sequence 2 input panel.

Upon saving the sequences the user can open the saved sequences panel (fig. 49), where all the sequences are stored. In this panel, one can select which sequence to align in either axis, delete a specific sequence, or delete all the sequences previously saved. The default sequences are impossible to delete.

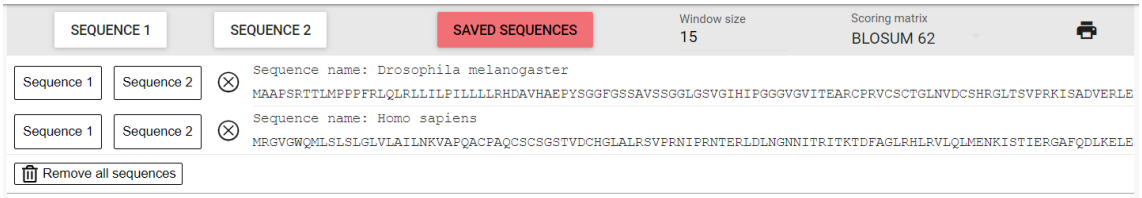

Figure 49 – Saved sequences panel.

To select a sequence the user must click in either the "*sequence 1*" or "*sequence 2*" buttons present in the saved sequences panel, as shown in figure 50. These will set the selected sequence to replace the previous one in the dotter panel.

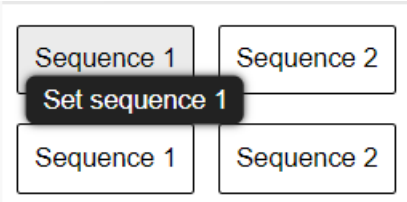

Figure 50 – Set a sequence for the alignment, within the saved sequences panel.

For the user to remove a specific sequence from the *localStorage* one must use the specific button for it, shown in figure 51. This will remove the selected sequence permanently from the storage, until the user re-saves it.

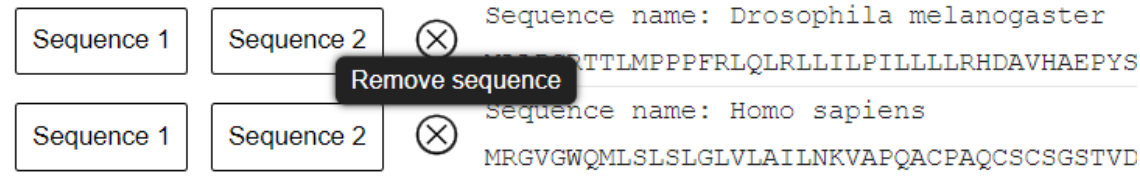

Figure 51 – Button to remove a specific sequence from the localStorage.

For the removal of every sequence, apart from the default ones, the user has to click in the last button present in the saved sequences panel, as shown in figure 49.

Then, in the input panel we have the window size. The user can change its value, resulting in a different view of the dot plot panel, of the two sequences panel, and on the bar chart of the density panel. The default value is "*15*" and we can see how everything is presented in figure 45. In comparison, if we change the window size value to "*1*", then we will have a much different approach to the sequence alignment, as we see in figure 52.

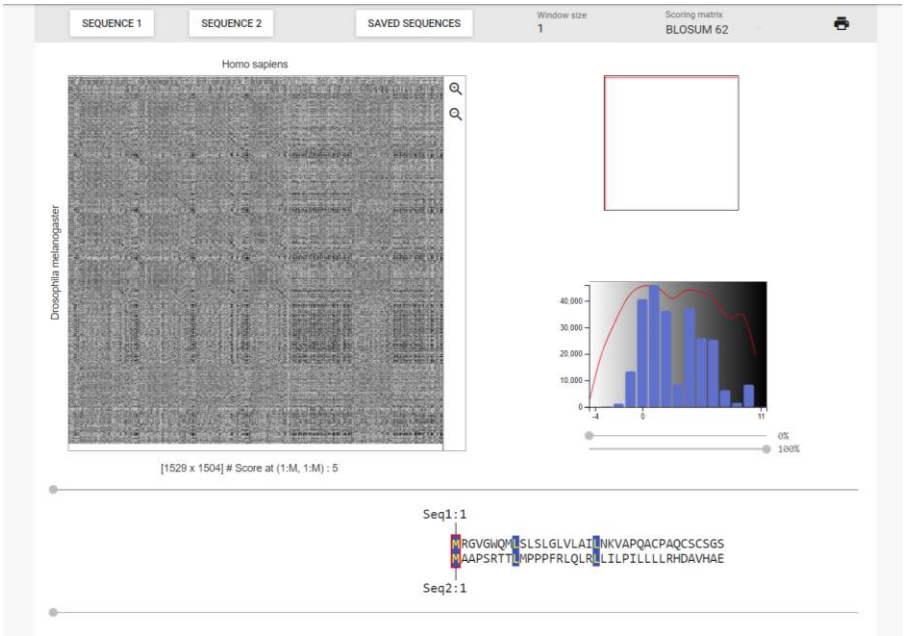

Figure 52 – The sequences alignment with the window size changed to "1".

To change the scoring matrices used in the sequence alignment, the user must resort to the "*Scoring matrix*" expandable button, shown in figure 53.

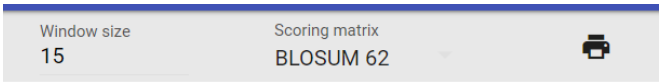

Figure 53 – The scoring matrix button to change the scoring method of the sequence's alignment.

As explained above, we present the user with multiple types of scoring matrices. One must choose which works best to achieve the expected results. Additionally, we present the identity matrix. The menu where the user can select any of these matrices by expanding the "*scoring matrix*" button is shown in figure 54.

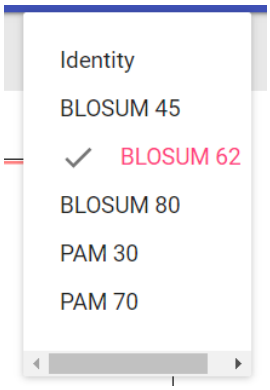

Figure 54 – The scoring matrices available for the sequence's alignment.

Lastly, in the input panel the user is presented with a print button, where one can print the sequence alignment.

After selecting which sequences to compare, the user proceeds to view the dot plot panel, where the sequences are represented (fig. 55). One can click in this panel pixel by pixel, viewing the sequence alignment for that exact location. The selected pixel is represented in the panel with a junction between two red lines.

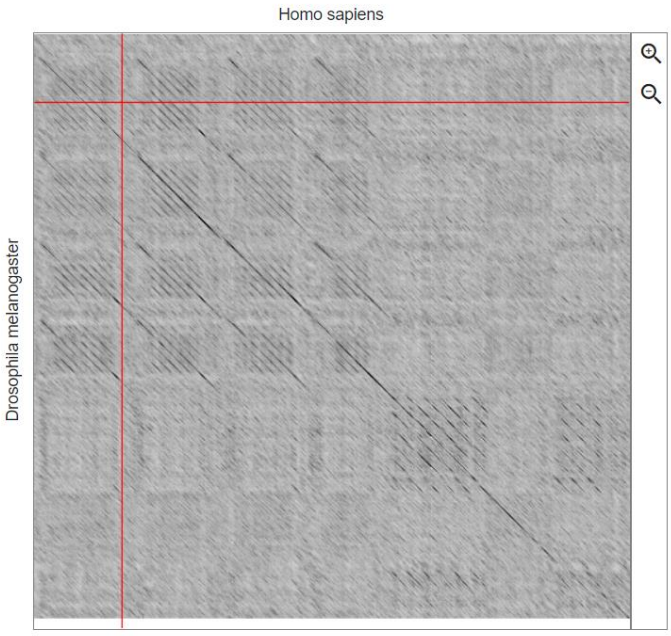

Figure 55 – The dot plot for the sequence's alignment.

In the dotter panel the user can also use the zoom buttons placed in the top right corner. By default, upon zooming in, it zooms to the area of the selected pixel (fig. 56). The zoom also changes the minimap and the density panel. The user can also change the zoom location by dragging or clicking the mouse in the minimap.

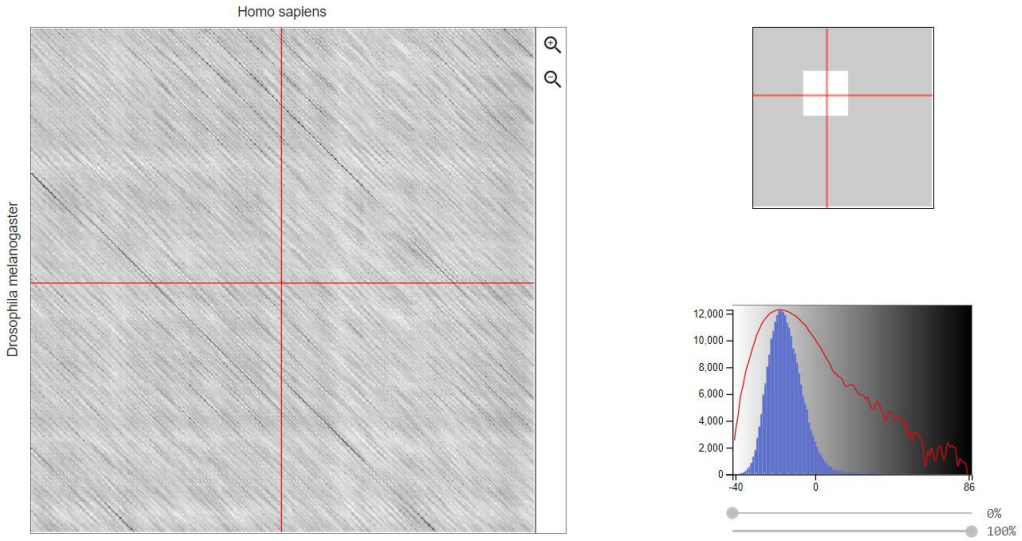

Figure 56 – The dotter panel zoomed in, alongside the minimap and the density panel.

The density panel is used to isolate either the high scores or low scores. The high scores are the matches between the two aligned sequences, and the low scores are the mismatches. Each match and mismatch correspond to a pixel in the dot plot panel, with the existence of a larger number of pixels with low scores and only a few with high scores. To remove the background noise (low scores) in order for the high scores to stand out, the user adjusts the greyscale scrollbars. A representation of presenting mostly high score values is shown in figure 57.

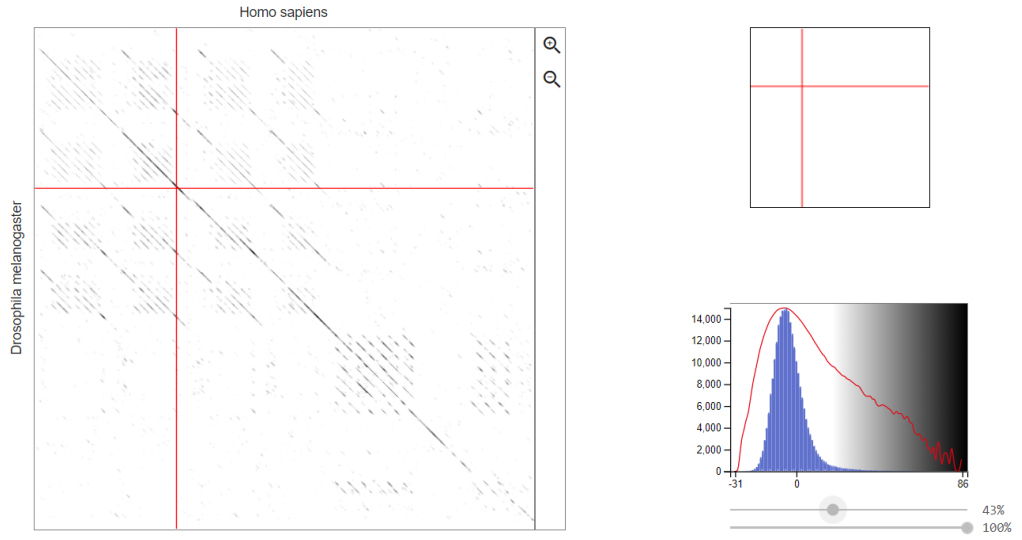

Figure 57 – Adjusting the greyscale in the density panel in order to present mostly high scores.

Under the dotter panel, the user is presented with the two sequences panel where one can change, by adjusting the grey scrollbars, the pixel location in the dotter panel. The user also has the score information for the alignment between the two sequences, in that exact pixel location. (fig. 58).

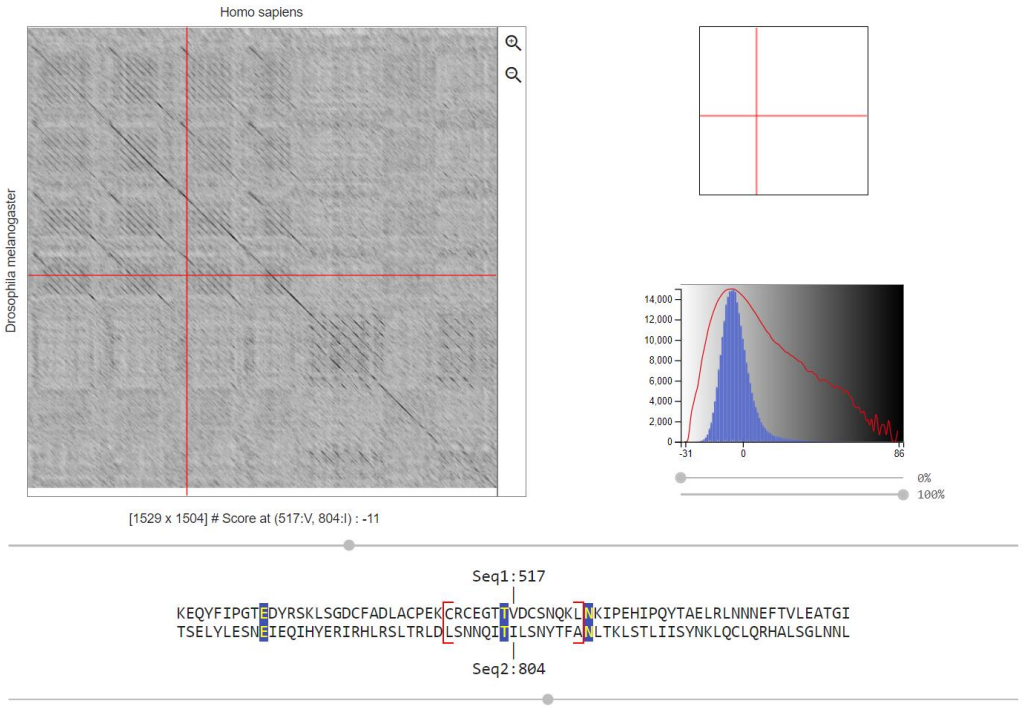

Figure 58 – Adjusting the scrollbar in the two sequences panel to change the pixel location in the dotter panel, accompanied by the alignment score for that exact pixel.

Changing the window size and the scoring matrix will result in different scoring and visualization, for the same sequences in the alignment. In the example displayed in figure 59, we have the window size set to "*5*" and the scoring matrix in use is "*PAM 30*".

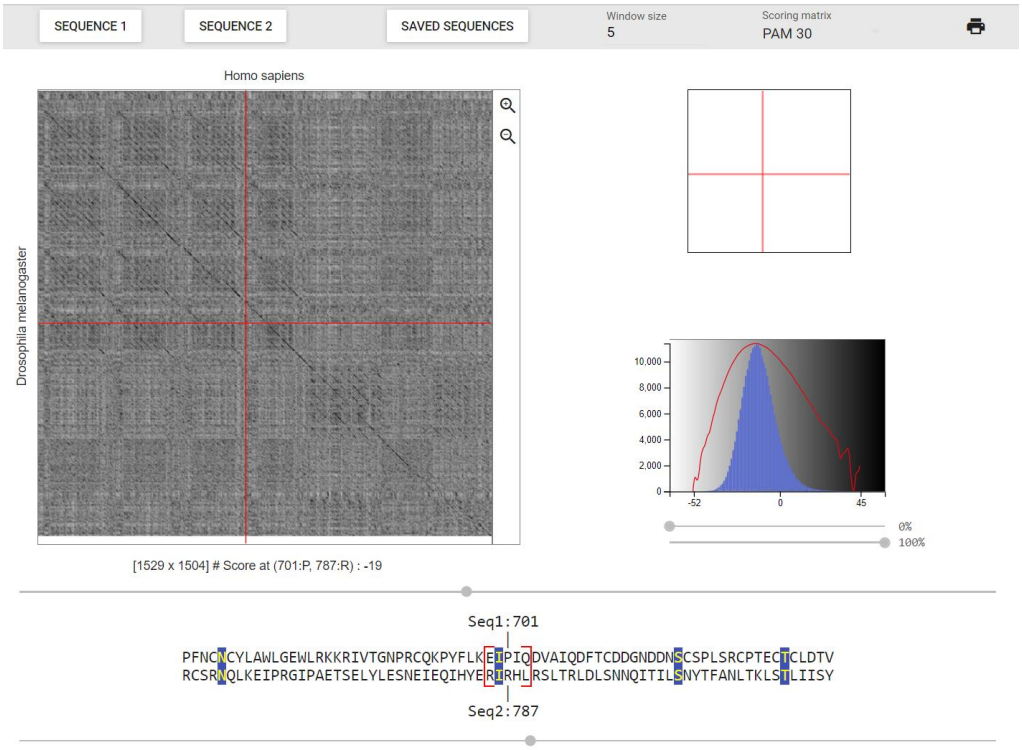

Figure 59 – Changing the window size and scoring matrix for the same sequences in the alignment.

Lastly, the user views the footer of the website, where one has the left section contained with the company the developers are associated with, and the version of the application. In the right section the user has two links. The first will take the user to the github page where the application is shared with the community, and the second will take the user to the documentation page of the web application (fig. 60).

Figure 60 – Dotlet's footer, presenting the associated company, the version, the github link and the documentation link.

5.2 Dotlet as a desktop application

The user might not always have access to the internet, therefore one has to resort to the dotlet's desktop application. In the example shown in figure 61, we have the file containing the desktop application for the windows OS. This folder was generated using the "*electron-builder*", whereas the folders for linux and macOS were generated by resorting to "*electron-packager*".

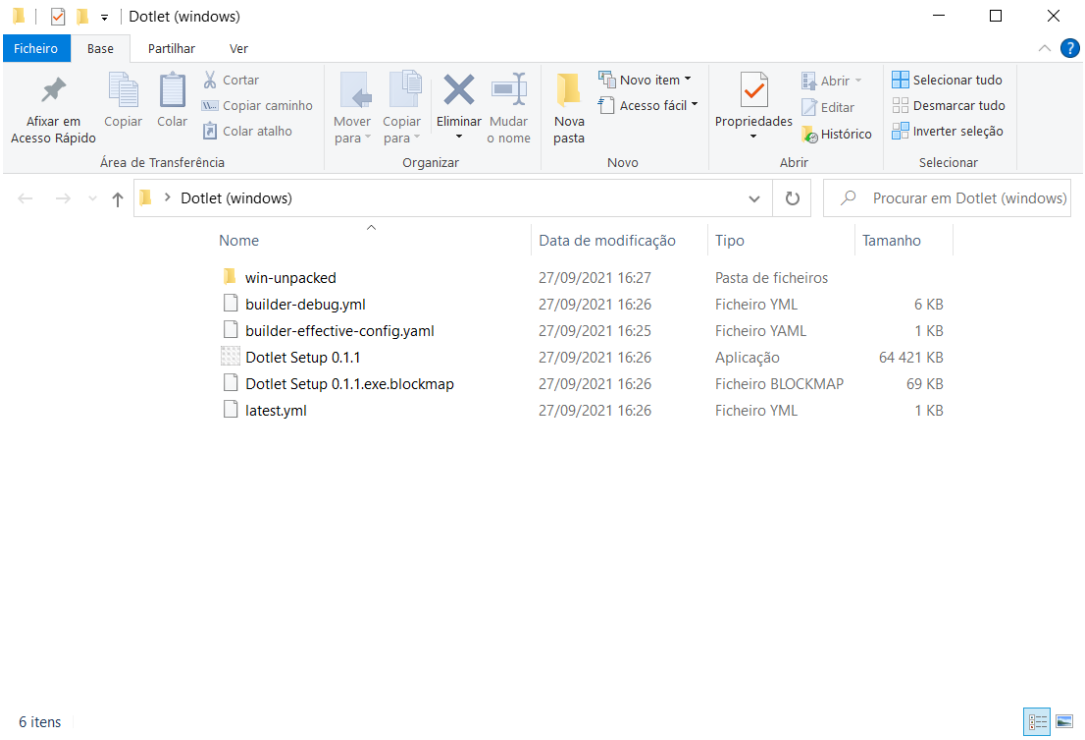

Figure 61 – Dotlet's desktop application main folder.

This folder contains the application installer, named "*Dotlet Setup 0.1.1*". Upon opening it, it installs the application automatically in the current OS. If the user prefers to open the application without installing it in one's machine, then one has to open the *"win-unpacked*" folder, shown in figure 61.

The "*win-unpacked*" folder is where the application information is stored, comprising multiple files for this purpose (fig. 62). For the user to launch the desktop application without running the

installer, one simply has to run the "*dotlet*" file present in this folder, which is an executable file that only launches the application without installing it.

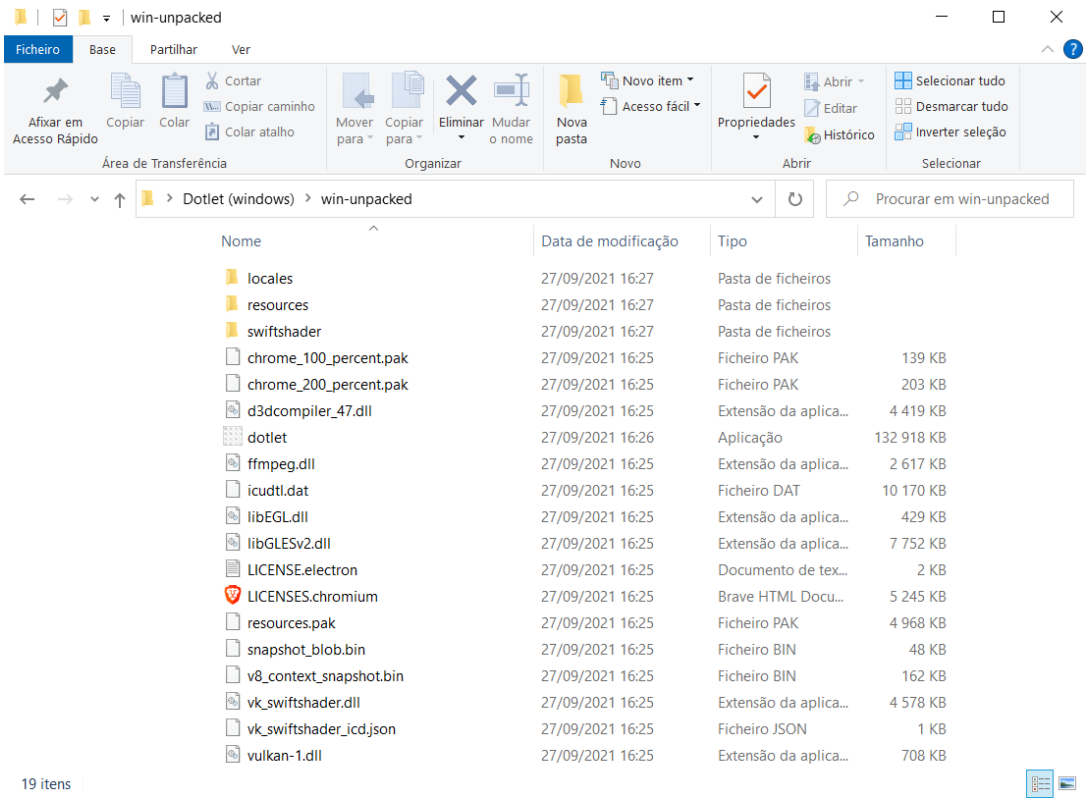

Figure 62 – Dotlet's desktop application main folder.

The files built for linux and macOS, with the different architectures, are built with "*electron-packager*". A folder is generated containing the dotlet desktop application for the different OS and architectures (fig. 63).

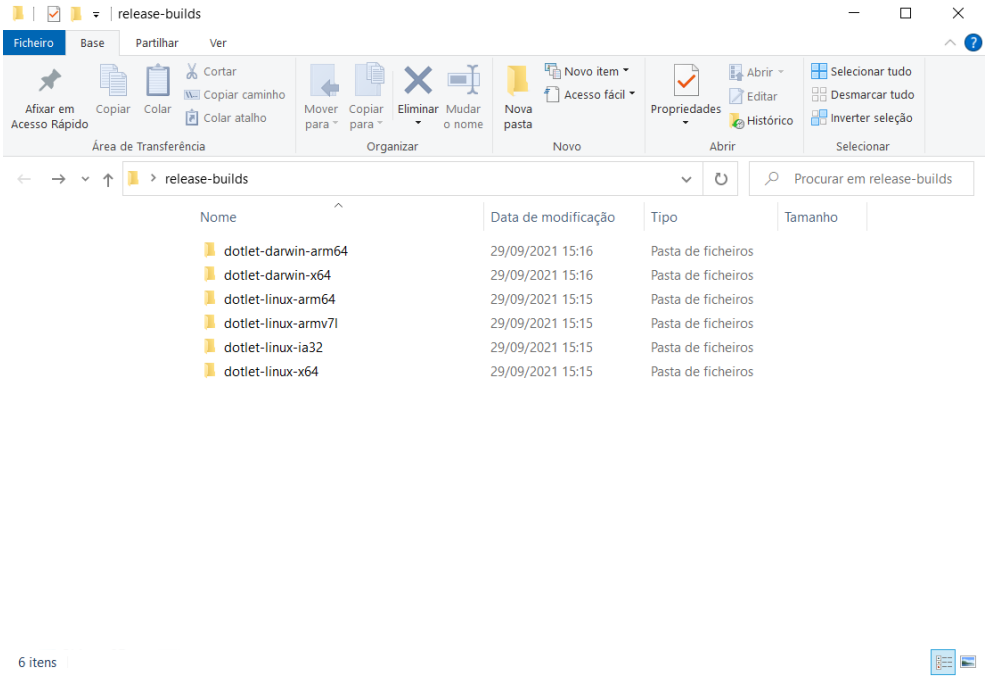

Figure 63 – Folders containing dotlet's desktop application, for the either linux and macOS.

Upon launching the application, the user is presented with a replicated version of the web application (fig. 64). This makes it easier for the user since both the workflow of the web and the desktop application are the same.

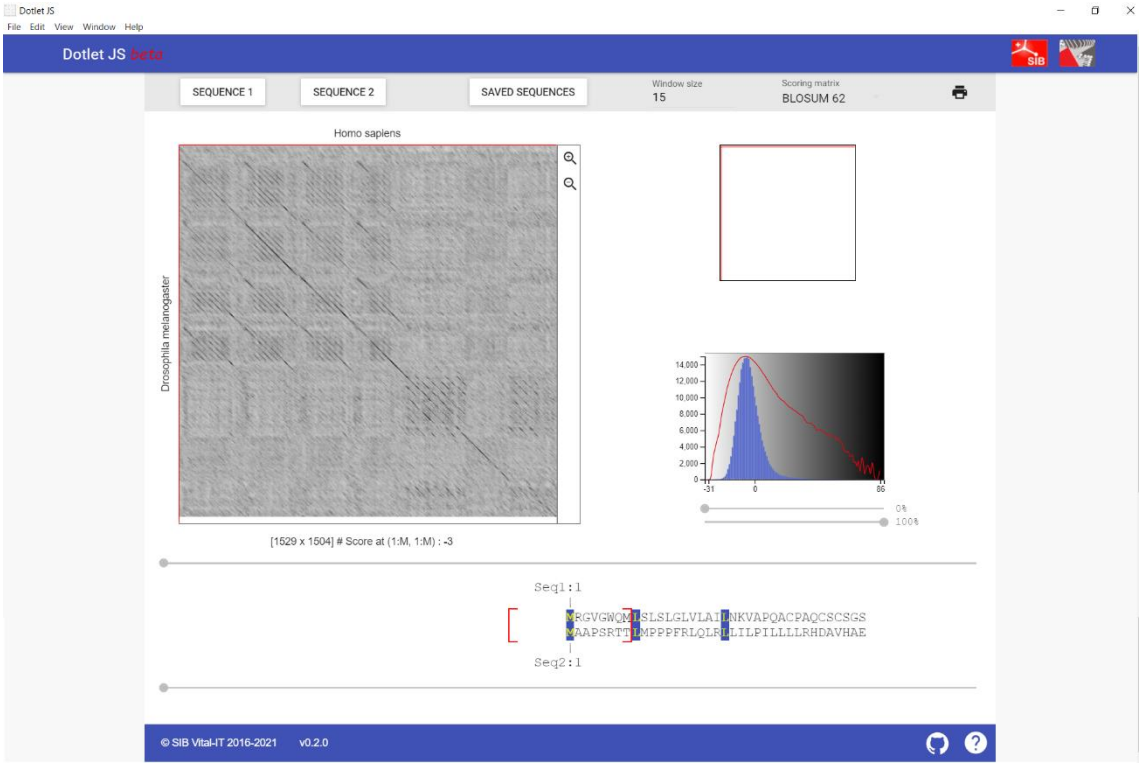

Figure 64 – Dotlet's desktop application in a windows OS.

With this desktop application, dotlet improves the user experience simply due to the fact that one is not required to be connected to the internet to align sequences.

6 Conclusion

This project consisted of multiple goals that were defined to achieve the ultimate objective of improving the dotlet application. For the project to be successful bibliographic search was done in order to contextualize the dot plot method and the different types of scoring matrices. Also, the bibliographic search was done to gather information about the different types of technologies we utilized in the project.

Before the first stage of developing the dotlet application we chose which technologies provided the necessary tools. To do this we resorted to the bibliographic search as well as our previous knowledge about these technologies. It was also taken into consideration the previous source code and which technologies were used in order to develop that version of dotlet.

Upon starting the dotlet development, the primary focus was to fix certain display issues the previous version presented. This included the lack of responsive behavior, which is important since the scientific community could be using tablets, and also enlarging the usable space in the application to make it user friendlier.

Upon making these changes we proceeded to fix dependencies issues that were occurring because of the updates certain packages suffered. These issues were preventing the build of the application, which even upon updating some of the dependencies is presenting errors in some machines. Despite efforts to fix this issue we found it overly complicated and since the build was being done by the developers and the server, we decided to continue the application development.

The next step was to implement a fetch feature in the application. We resorted to the *fetch()* method in JavaScript, to request the server information contained in the URL. After retrieving the information from the URL, which is a *fasta* file from the UniProt database, a verification takes place to validate the information present in the *fasta* file. It recognizes the common *fasta* file header, removes it, and presents the user with the intended sequence.

Another goal of this project was to allow the user to save the inputted sequences. This was done next, by resorting to the *localStorage* property. Upon pasting the sequence in the input panel, the user uses a button to save the sequence in the *localStorage*, which will then be presented in another panel where the saved sequences are displayed. This panel also presents buttons to select which sequence to align, to delete that specific sequence, and to delete every sequence present in the *localStorage*. If any of the deleted sequences are present in the alignment, then the default sequence for the "Sequence 1" or "Sequence 2" slot substitutes it. By default, two sequences are always present in the *localStorage*, which are the sequences the user is shown when entering the web application. If the user tries to save sequences with a custom name that is already applied to another sequence previously saved in the *localStorage*, then one is shown a message informing that. Also, if one tries to save a sequence that is already present in the *localStorage* a message is presented to the user informing that the sequence already exists. Upon saving a sequence a message is shown informing the sequence was successfully saved.

The final stage of this dissertation lies in the development of the stand-alone application of dotlet. This was achieved resorting to the *npm* packages *electron*, *electron-packager*, and *electron-builder*. *Electron* builds a replicated version of a web application as a desktop application, while *electron-builder* and *electron-packager* takes the built files and creates the folders with the executable files which allows the user to launch the desktop application.

This project is now concluded with the achievement of every established goal, where improvements can be made to the application, by applying different technologies or adding features to the existing application.

6.1 Possible tasks in the future

Regarding possible improvements to the dotlet application, multiple tasks could be done. In order to reach a greater number of people, older features that dotlet's Java version presented could be added, such as:

- Comparing DNA (translated in three frames) vs protein (e.g., identify exons in a genomic sequence);
- Single DNA comparison (DNA vs cDNA).

Other possible implementations to the dotlet application could be:

- Allow users to input sequences by submitting a *fasta* file present one's computer. The filename extensions could be *.fasta*, *.fna*, *.ffn*, *.frn*, *.fa*;
- Automatically retrieve the name of the sequence through the header of the *fasta* file;
- Allow more databases URL's, containing *fasta* files, to be used.

References

1. Junier, T., Pagni, M.: Dotlet: Diagonal plots in a Web browser, www.isrec., (2000).
2. Gibbs, A.J., Mcintyre, G.A.: The Diagram, a Method for Comparing Sequences: Its Use with Amino Acid and Nucleotide Sequences. Eur. J. Biochem. 16, 1–11 (1970).
3. Devereux, J., Haeberli, P., Smithies, O.: A comprehensive set of sequence analysis programs for the VAX. Nucleic Acids Res. (1984).
4. Sonnhammer, E.L.L., Durbin, R.: A dot-matrix program with dynamic threshold control suited for genomic DNA and protein sequence analysis (Reprinted from Gene Combis, vol 167, pg GC1-GC10, 1996). Gene. 167, Gc1–Gc10 (1995).
5. Gibbs, A.J., Dale, M.B., Kinns, H.R., Mac Kenzie, H.G.: The transition matrix method for comparing sequences; Its use in describing and classifying proteins by their amino acid sequences. Syst. Biol. (1971).
6. Fitch, W.M.: Locating gaps in amino acid sequences to optimize the homology between two proteins. Biochem. Genet. (1969).
7. Arnold, K., Gosling, J., Holmes, D.: The Java programing language. (2013).
8. Keller, M., Nussbaumer, M.: Cascading Style Sheets: A novel approach towards productive styling with today's standards. In: WWW'09 - Proceedings of the 18th International World Wide Web Conference (2009).
9. Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for JavaScript. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (2009).
10. Combe, T., Martin, A., Di Pietro, R.: To Docker or Not to Docker: A Security Perspective. IEEE Cloud Comput. (2016).
11. Cotroneo, D., Natella, R., Pietrantuono, R., Russo, S.: Software aging analysis of the linux operating system. In: Proceedings - International Symposium on Software Reliability Engineering, ISSRE (2010).
12. Kredpattanakul, K., Limpiyakorn, Y.: Transforming javascript-based web application to cross-platform desktop with electron. Lect. Notes Electr. Eng. 514, 571–579 (2019).
13. Mount, D.W.: Comparison of the PAM and BLOSUM amino acid substitution matrices. Cold Spring Harb. Protoc. (2008).
14. Wheeler, D.: Selecting the Right Protein-Scoring Matrix. Curr. Protoc. Bioinforma. (2003).
15. Gackenheimer, C., Gackenheimer, C.: What Is React? In: Introduction to React (2015).

16. Zayour, I., Hamdar, A.: A qualitative study on debugging under an enterprise IDE. Inf. Softw. Technol. 70, 130–139 (2016).
17. Blischak, J.D., Davenport, E.R., Wilson, G.: A Quick Introduction to Version Control with Git and GitHub. PLoS Comput. Biol. 12, 1–18 (2016).
18. Tilkov, S., Vinoski, S.: Node.js: Using JavaScript to build high-performance network programs. IEEE Internet Comput. 14, 80–83 (2010).
19. Duldulao, D.B., Cabagnot, R.J.L.: Practical Enterprise React. (2021).
20. T.N.Sharma, Priyanka Bhardwaj, Manish Bhardwaj: Differences between HTML and HTML 5. Int. J. Comput. Eng. Res. 2, 1430–1437 (2012).
21. Genevès, P., Layaïda, N., Quint, V.: On the analysis of cascading style sheets. WWW'12 - Proc. 21st Annu. Conf. World Wide Web. 809–818 (2012).
22. The, J., Guide, D.: JavaScript - The Definitive Guide , 5th Edition. (2006).
23. Ong, S.P., Cholia, S., Jain, A., Brafman, M., Gunter, D., Ceder, G., Persson, K.A.: The Materials Application Programming Interface (API): A simple, flexible and efficient API for materials data based on REpresentational State Transfer (REST) principles. Comput. Mater. Sci. 97, 209–215 (2015).
24. Jordan, H., Schneider, N., Barth, L., Caspers, M., Peters, C., Specht, D., B¨uhring, T.: Rich Internet Applications w/HTML and Javascript. Rich Internet Appl. w/HTML Javascript. 25 (2017).
25. Casario, M., Elst, P., Brown, C., Wormser, N.: HTML5 Local Storage Solution 11-1 : Understanding Occasionally- Connected Applications. HTML5 Solut. Essent. Tech. HTML5 Dev. 281–303 (2011).
26. Clow, M.: Introducing Webpack. Angular 5 Proj. 133–137 (2018).
27. Mardan, A.: Applying Stylus, Less, and Sass. Pro Express.js. 181–183 (2014).
28. Maynard, T.: Getting Started with Gulp – Second Edition. (2017).
29. Boduch, A.: React Material-UI Cookbook: Build captivating user experiences using React and Material-UI. (2019).