Pepfeature: A Python package that generates useful features for linear B-cell epitope prediction purposes.

Student Name	Essa Umar Khan
University	Aston University
Module Code	CS3010
Module Name	Individual Project
Student Number	170077653
Candidate No.	976199
Contact Email	khane2@aston.ac.uk
Supervisor	Dr. Felipe Campelo
Contact Email	f.campelo@aston.ac.uk

Submission Date	02/05/21

Abstract

Pepfeature, a Python package has been developed. It consists of functions for calculating eight distinct sequence-based families of features for a vector of peptides (input as part of a Pandas data frame). The set of features calculated are namely: Amino acid frequencies, Conjoint Triads, K-mer frequencies, AA descriptors, Amino acid composition, Sequence entropy, Molecular Weight and Number of Atoms. This package was designed and implemented in a way so that it can integrate with the main client's prediction pipeline but is flexible enough for general use by other researchers developing epitope prediction tools. Moreover, it is designed/structured in a manner to ease future additions of feature calculation capabilities to the package.

Pepfeature is a refactoring of the feature calculation capabilities in the R package *Epitopes* – a tool currently used by the main client. It has outperformed *Epitopes* in benchmarks by a comfortable margin, which makes it an asset for the main client's predictive pipeline.

How to access the deliverable and the source code?

It can be done through:

https://github.com/essakh/pepfeature

Moreover, the documentation on GitHub provides instructions on how to install and use the package. Installing the package is essentially running the pip command:

pip install Pepfeature

Table of contents

1. Introduction

1.1 What is computational epitope prediction and why is it important?

Linear B-cell epitopes are short strings of amino acids within a continuous stretch of a protein sequence (Wang et al., 2011); these are located on the surface of an antigen, they trigger an immune response and are recognized by specific antibodies, and for the purpose of removing the antigen from the body bind to (Wang and Pai, 2014).

There exists motivation for the identification of linear B-cell epitopes as they provide the underpinning knowledge for the development of epitope-based vaccines, diagnostic tests, and various disease prevention techniques (Potocnakova, Bhide and Pulzova, 2016). Fortunately, the increasing availability of verified epitope databases has paved the way for the utilization of computational techniques for linear B-cell epitope prediction (ibid.). Compared to experimental identification, computerized techniques for prediction are more efficient due to lower demand of resources, time and effort (EL-Manzalawy and Honavar, 2010). Computational prediction serves as a prioritization strategy for laboratory validation - it is not a substitute for laboratory identification but rather a pre-filtering tool to facilitate more efficient investigation of the most promising targets.

Computational methods for prediction consist (but not solely) of algorithms that predict linear B-cell epitopes based on data derived from the protein sequence of the antigen (Sanchez-Trincado, Gomez-Perosanz and Reche, 2017). Feature calculation/extraction is an important part of the development of good prediction pipelines for this specific data, i.e., the generation of the most informative features that can help an algorithm such as a classifier, correctly identify the most promising regions from what would be a linear B-cell epitope in a protein sequence.

The focus of this project is the development of a Python package that consists of routines to calculate features on epitope data sets for linear B-cell epitope prediction purposes.

1.2 Contributions

The main contributions of this project are summarised below:

• A publicly available Python package named Pepfeature consisting of feature calculation routines for epitope data, namely: amino acid (AA) frequencies (20 features), Conjoint Triads (343 features), K-mer frequencies (400+ features), AA descriptors (66 features), AA Composition (9 features), Sequence Entropy (1 feature), Molecular Weight (1 feature) and Number of Atoms (5 features). This package was designed and implemented in a way so that it can integrate with the main client's classification pipeline but is flexible enough for wider general use by the scientific community. Moreover, it is designed/structured in a manner to ease future additions of feature calculation capabilities to the package.

• Systematic evaluation of performance of the implemented routines in Pepfeature with that of the existing R package solution (named '*Epitopes*'), covering the aspect of runtime and accuracy.

1.3 Context and Clients

The justification for the development of these feature extraction capabilities in Python is based on the context in which the package has been developed in. This is elaborated as follows:

The main client of this project is Jodie Ashford - a researcher in the field of machine learning approaches for epitope prediction. She uses a classification pipeline (see Figure 1) for her research in which currently the 'Model Fitting' and Prediction Block use Python tools and the preparation for these i.e., the data retrieval, consolidation and 'Feature Calculation' block of the pipeline utilize tools coded in the R programming language. Of focus is the 'Feature Calculation' block, which is currently facilitated by the usage of a package coded in R named '*Epitopes*'.

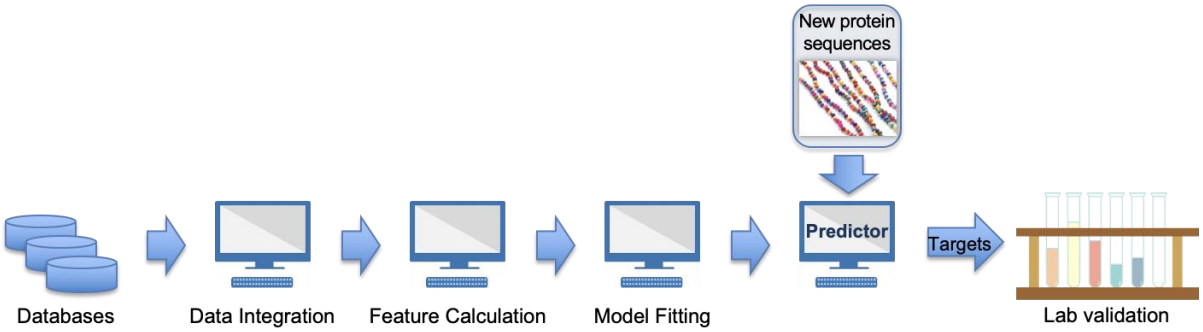

Figure 1 : Main client's Machine Learning Epitope Prediction Pipeline [Diagram courtesy of Dr. Campelo (2020)]

The Python package deliverable of this project is part of a collective effort to transform the main client's current mixed-languages prediction pipeline into a full Python pipeline. This project contributes to this by the refactoring of code from the R *Epitopes* package to a novel Python package, based on first principles of what those features are intended to calculate rather than a simple reproduction of the existing R implementation. Therefore, the deliverable of this project is the substitute Python solution to execute the 'Feature Calculation' block of the prediction pipeline. Likewise, there are efforts by other people on developing Python solutions for other different blocks of this pipeline.

Besides catering towards the needs of the main client and having the package locally available for use by her, the package is also available publicly to the wider Python and scientific community. Moreover, Dr. Campelo acts as a secondary client who has a great interest in the main client's research and will be representing the wider research community in Epitope Prediction – to whom the public Package will also be useful. Dr. Campelo is also the supervisor of this final year project and Jodie's PhD research.

1.4 Justification for Development

Although the R package suffices the main client's needs for the 'Feature Calculation' block of her pipeline, but a Python package alternative holds the potential to bring multiple improvements in many aspects to this process. The main client focused benefits are elaborated as follows:

For the client, the utilization of the envisioned Python solution is in practice more convenient and less complex to work with than using *Epitopes* in her current pipeline as it eliminates the need to switch from an R package for the pipeline's 'Feature Calculation' block to a Python package for the

‘Model Fitting’ block. The benefit of this is that the updated pipeline will be executable completely from within a single deployment environment.

Moreover, there is another student developing a Python solution for the ‘Data Integration’ block (Figure 1) - which directly precedes the ‘Feature Calculation’ block - as part of a collective effort to transform Jodie’s pipeline into a pure Python pipeline; in such an outcome the Pepfeature package integrated in this envisioned Pythonic pipeline will result in seamless compatibility between the inputs and outputs of the blocks directly upstream and downstream respectively.

All the aforementioned justifications are based on the intuitive expectation that the least variance there is in the programming languages utilized to code the modules in a prediction pipeline than the less complex development, collaboration, version control, package control, distribution and testing become. There is great motivation for diminishing complexity, as it saves resources and time.

Another motivation for the development of the Python package alternative to carry out the feature calculation in the prediction pipeline, is that results from a series of R vs Python speed benchmark tests conducted by Korstanje (2020) suggest that Python may carry out the feature calculation tasks in less time; for the end user this provides the benefit of training their models faster, as Machine Learning pipelines can be used iteratively to continuously improve the model (Kunft et al., 2019).

Additional non-client specific motivation for the development of Pepfeature is that it is a consolidated, publicly available Python package to calculate protein-based features; the utility of this is already excellent and a contribution to the diverse Python ecosystem of scientific libraries that makes it a preferred tool for data mining.

1.5 Report Structure

The remainder of the report is organised in the following manner (each section is coupled with evaluations and reflections where appropriate):

• Section 2 consists of an overview of the key research conducted in preparation and during the development of Pepfeature and further necessary theory that this deliverable relied on.

• Section 3 provides the approach that was taken to manage this project and interact with the clients.

• Section 4 mainly expounds how the macro requirements were formulated and what design decisions were taken to cover these requirements in the deliverable—

• Section 5 details the strategy used to verify the results produced by Pepfeature’s routines, how the functionality of the routines was tested to be working, and noteworthy findings whilst testing have been detailed.

• Section 6 presents details of how the package has been maintained after delivery.

• Section 7 presents the results of Pepfeature’s benchmark conducted against the in-house R package.

• Section 8 provides the feedback of the clients on the deliverable.

• Section 9 produces the conclusion of the work, summarizing what has been achieved and what was covered in this report.

2. Background Research

2.1 Existing Solutions

In this section the research and analysis of relevant existing solutions are presented.

2.1.1 The Model 'R' Package: Epitopes

'Epitopes' is a prototype R package for feature extraction and processing of epitope data from the Immune Epitope Database. The developers of *Epitopes* include this project's supervisor/secondary-client Dr. Felipe Campelo and the main client, Jodie Ashford (Campelo, Ashford and Lobo, 2021).

In Section 1.3 it was expounded that Pepfeature is a replacement for *Epitopes* to facilitate the main client's prediction pipeline. The epitope features this R package calculates are the same features that Pepfeature is aimed to calculate. These features are detailed in Section 2.2 of the report. Thus, Pepfeature can be thought of as a partial refactoring of the *Epitopes* R package to a novel Python package; a partial port only as *Epitopes* provides additional processing routines that facilitate the 'Data Integration' block of the current prediction pipeline, which are outside the scope of requirement for this project.

This package was not specifically made for the client's pipeline - rather she adopted around it. Pepfeature on the other hand, is specifically made to integrate with the main-client's pipeline.

The source code[1] of this package was not public during most of the development of this project and was only partially released in early 2021 and thus Pepfeature's routines' development was instead purely based on first principles of what those features are intended to calculate. This was an intentional development decision, so that Pepfeature's code would not be influenced by specific design decisions of the *Epitopes* package, thereby increasing the likelihood of developing a novel way of calculating those features which may potentially be more efficient; and to ensure that its distinctions will make it a worthwhile alternative of a package to the scientific community to whom it will be made available.

2.1.2 Pfeature Python Package

According to Pande et al. (2019) Pfeature is a collection of routines for computing a wide range of protein and peptide features. This tool can calculate more features than this project's deliverable (Pepfeature). Its developers consist of multiple scientists.

Pfeature can be used in two main ways:

[1] Epitopes source code: https://github.com/fcampelo/epitopes

1) A webserver that provides the Pfeature's functionality via a web interface[2]

2) A Python package[3]

A detailed study of this package's code was not conducted for the same reasons stated for why it was not done for the *Epitopes* package. This decision was taken to ensure that Pepfeature's code development would not be influenced by Pfeature's design decisions, to increase the likelihood of coming up with novel ideas and to ensure that it is a distinctive worth wile alternative of a package to the scientific community to whom it will be made available.

Pfeature is a direct competitor to Pepfeature but what distinguishes Pepfeature from this package is that it is an in-house built package that will be customized to the client's prediction pipeline, but at the same time have a level of flexibility so that it could be utilized by the wider scientific community as well; the features that Pepfeature calculates are focused/niched for epitope prediction whereas Pfeature is capable of calculating an enormous amount of features that are relevant for proteins and peptides but not necessarily relevant for epitope prediction (ibid.). Moreover, Pepfeature will be expandable enough to allow others to add more features if they desire.

2.2 Feature Extraction Capabilities to Develop

In total Pepfeature can calculate eight groups of sequence-based features. In this section the underpinning theoretical concepts of the feature extraction routines that are implemented in Pepfeature are expounded. This understanding of how to derive the features laid down the foundations for formulating the logical steps taken during their programmatic algorithmic implementation in Python code. A description of how to calculate each of the eight group of features is given.

Note that proteins/peptides are composed of 20 natural AAs. In this project these 20 natural AAs are referred to as by their respective one-letter symbol: A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y (Amino Acids Reference Charts, 2021).

2.2.1 Proportion of Individual AAs in Sequence

These are the simplest features to calculate and are essentially the frequencies of AAs in a sequence. It will be calculated as the proportion (out of 1) of all 20 natural AAs each, in a peptide. Thus, this feature calculation gives 20 newly derived features for each peptide (Jespersen et al., 2019). The calculation of the feature can be expressed as follows:

A peptide P composed of L number of AAs can be expressed as:

$$P = AA_1AA_2\ AA_3 \dots AA_L$$

[2] Available from URL: https://webs.iiitd.edu.in/raghava/pfeature/

[3] Available from URL: https://github.com/raghavagps/Pfeature

The AA frequencies features AAF for a particular peptide e.g., P, can be expressed as:

$$AAF = \{f_1, f_2, f_3, \dots, f_{20}\}$$

where $f_i = \frac{t_i}{L}$; t_i is the number of AAs of type i in the peptide and where i corresponds to the 20 natural AAs ($i = 1, 2, 3 \dots 20$) (Pande et al., 2019).

2.2.2 *k*-mer Composition

k-mers in this project's context are k-length contiguous combinations of subsequences of AA letter(s) in a peptide (Paull et al., 2019).

As a visual example, all the possible *k*-mers of a sequence are shown in Table 1.

Table 1: k-mers, where k is equal to 1 to 5 for the sequence: DVHIE

k	k-mers
1	D, V, H, I, E
2	DV, VH, HI, IE
3	DVH, VHI, HIE
4	DVHI, VHIE
5	DVHIE

The process of k-mers where k=3 can be expressed as follows:

A peptide P composed of L number of AAs can be expressed as:

$$P = AA_1AA_2\ AA_3 \dots AA_L$$

Next, successively sliding windows with continuous three AA sequences gives:
$AA_1AA_2AA_3$, $AA_2AA_3AA_4$, $AA_3AA_4AA_5$, … , $AA_{L-k-1}AA_{L-k}\ AA_{L-k+1}$.

Generally, P will have $L - k + 1$ k-mers and n^k is the count of total possible k-mers, where n is the number of possible monomers. For example, since there are 20 valid AA letters, there are 400 (20^2) possible 2-letter combinations, 8000 (20^3) 3-letter combinations, etc.

The k-mer composition features KCF that represent a particular peptide with $k > 0$, can be expressed as:

$$KCF = \{f_1, f_2, \dots, f_{n^k-2}, f_{n^k-1}, f_{n^k}\}$$

where $f_i = \frac{c(s_i,P)}{|mers_k(P)|}$; $mers_k(P)$ denotes the set of all different k-mers in P and $|mers_k(P)|$ is the count of elements in this set (i.e. the count of k-mers in P); for the sequence s_i of the i-th type, where $s_i \in mers_k(P)$. Let $c(s_i, P)$ be the count of positions in P where s_i occurs; i corresponds to the i-th k-mers in $mer_k(P)$.

All other possible k-mer sequences (amounting to n^k) that $\notin mer_k(P)$ are by defaulted to have zero as their frequency value within P as their feature value.

2.2.3 Conjoint Triad Frequencies

According to the method of Shen et al. (2007) Conjoint Triad Frequency features (CTFF) are based on structural neighbours, where AAs are assigned to one of 7 classes based on their physiochemical properties. The classification of the 20 AAs is shown in Table 2.

Table 2: Classification of AAs

Group	AA belonging to the Group
0	A, G, V
1	C
2	F, I, L, P
3	M, S, T, Y
4	H, N, Q, W
5	K, R
6	D, E

To calculate the CTFF of a peptide, first represent each AA in the peptide by its group value. For an example, see Table 3.

Table 3: A Peptide and its Equivalent Group Value Representation

Peptide	Group Value
MVRKGEKKKAKP	305506555052

Secondly, calculate the frequencies of each 3-number subsequence. By this way, a peptide is represented by 343 (7 × 7 × 7) calculated new features (Yang, Xia and Gui, 2010). This can be expressed as follows, where a peptide P composed of L number of AAs is given as:

$$P = AA_1AA_2\ AA_3 \ldots AA_L$$

For the next step, successively sliding windows with continuous three AA sequences gives: $AA_1AA_2AA_3$, $AA_2AA_3AA_4$, $AA_3AA_4AA_5$, … , $AA_{L-2}, AA_{L-1}\ AA_L$.

The CTFF that represent a particular peptide $CTFF$, can be expressed as:

$$CTFF = \{f_1, f_2, f_3 \ldots, f_{343}\}$$

Where $f_1 = \frac{c_i}{L-2}$, and c_i is the count of positions in P where the i-th triad type of all contiguous three AAs occur and where values for i correspond to the i-th triads (i= 1, 2, 3, ⋯, 343).

2.2.4 Sequence Entropy

To have a measure of the complexity of a peptide at its AA composition level its entropy is calculated. Calculating the entropy for a given peptide results in a single feature can be expressed as follows, where a peptide P composed of L number of AAs is given as:

$$P = AA_1AA_2\ AA_3 \ldots AA_L$$

And the Sequence Entropy feature for P is calculated using the following function $\mathrm{E(P)}$:

$$\mathrm{E(P)} = -\sum_{i=1}^{20} P(AA_i)\log_2(P(AA_i)),$$

where $P(AA_i)$ is the proportion of occurrence of the i-th AA in P and where values for i correspond to the i-th AAs (i= 1, 2, 3, ⋯, 20) (Pande et al., 2019).

2.2.5 AA Composition

According to Osorio, Rondón-Villarreal and Torres (2015) AA Composition features (AACF) for a peptide are calculations of the number of AAs of a particular class; the 20 natural Aas are classified based on their physiochemical characteristics shown in Table 4.

Table 4: Classification of Aas

Class	**Aas belonging to class**
Tiny	“A”, “C”, “G”, “S”, “T”
Small	“A”, “B”, “C”, “D”, “G”, “N”, “P”, “S”, “T”, “V”
Aliphatic	“A”, “I”, “L”, “V”
Aromatic	“F”, “H”, “W”, “Y”
NonPolar	“A”, “C”, “F”, “G”, “I”, “L”, “M”, “P”, “V”, “W”, “Y”
Polar	“D”, “E”, “H”, “K”, “N”, “Q”, “R”, “S”, “T”, “Z”
Charged	“B”, “D”, “E”, “H”, “K”, “R”, “Z”
Basic	“H”, “K", "R"
Acidic	"B", "D", "E", "Z"

The process of calculating AACF is easiest to be understood with an example. The AA sequence P is defined as:

$P =$ LLLLLLLLDVHIESG

The AACF would be the proportions shown in Table 5.

Table 5: AACF of the AA sequence: LLLLLLLLDVHIESG (read table left to right)

Class	***P*’s AA characters belonging to the respective class. (*Char*)**	**Total number of *Char* belonging to the respective class divided by *P*’s length.** *(These are the AACF for P)*
Tiny	G, S	2/15
Small	D, V, G, S	4/15
Aliphatic	L, L, L, L, L, L, L, L, V, I	10/15

Aromatic	-	0/15
NonPolar	I, G	2/15
Polar	D, H, E, S	4/15
Charged	D, E, H	3/15
Basic	H	1/15
Acidic	D, E, H	3/15

2.2.6 Number of Atoms

Number of atoms features (NOAF) for a peptide are the total number of each type of atom in the sequence – essentially a weighted sum of the number of atoms in each of the constituent AA characters of the peptide.

Table 6: Table showing the number of each type of atom (C, H, O, N, S) for each AA (Biological Magnetic Resonance Data Bank: Chemical Shift Statistics table, 2021)

AA	C	H	O	N	S
A	3	7	1	2	0
C	3	7	1	2	1
D	4	7	1	4	0
E	5	9	1	4	0
F	9	11	1	2	0
G	2	5	1	2	0
H	6	9	3	2	0
I	6	13	1	2	0
K	6	14	2	2	0
L	6	13	1	2	0
M	5	11	1	2	1
N	4	8	2	3	0
P	5	9	1	2	0
Q	5	10	2	3	0
R	6	14	4	2	0
S	3	7	1	3	0
T	4	9	1	3	0
V	5	11	1	2	0
W	11	12	2	2	0
Y	9	11	1	3	0

Table 6 shows the number of each type of atom (C, H, O, N, S) for each AA. Thus, for every given peptide five features are generated corresponding with each type of atom.

The process of calculating NOAF is illustrated with an example:

The AA sequence P is defined as:

$P =$ LLDVVHI

Table 7 demonstrates the calculation of 1/5 of features, that is the number of atom 'C' in P. The feature i.e., the weighted sum is 28. This calculation would be done for each of the 4 remaining atoms (H, O, N, S) to determine the remaining 4 NOAF for P.

Table 7: Example of NOAF calculation for one type of Atom (C) for the sequence LLDVVHI (read table left to right).

AA characters in P (Char)	**Total occurrence of Char in P (O)**	**Atom 'C' value of Char (nC)** *[values taken from table 6]*	**O * nC**
L	2	6	(2*6=) 12
D	1	4	(1*4=) 4
V	2	5	(2*5=) 10
H	1	6	(1*6=) 6
I	1	6	(1*6=) 6
		Weighted sum	38

2.2.7 Molecular Weight

The Molecular Weight feature (MWF) for a peptide can be calculated as the weighted sum of each constituent AAs multiplied by its weight, as given in Table 8:

Table 8: Molecular Weight of the 20 natural AAs (Amino Acids Reference Charts, 2021).

AA	**Weight**
A	89.09
C	121.16
D	133.10
E	147.13
F	165.19
G	75.07
H	155.16
I	131.17
K	146.19
L	131.17
M	149.21
N	132.12
P	115.13
Q	146.15
R	174.20
S	105.09
T	119.12
V	117.15
W	204.22
Y	181.19

The process of calculating MWF is illustrated with an example:

The AA sequence P is defined as:

P = LLDVVHI

Table 9 demonstrates the calculation of MWF for a given Peptide.

Table 9: Example of MWF calculation for the sequence LLDVVHI (read table left to right).

AA characters in P (Char)	**Total occurrence of Char in P (O)**	**Weight of Char (W)**	**O * W**
L	2	131.17	(2*131.17=) 262.34
D	1	133.10	(1*133.10=) 133.1
V	2	117.15	(2*117.15=) 234.3
H	1	155.16	(1*155.16=) 155.16
I	1	131.17	(1*131.17=) 131.17
		Weighted sum	916.07

2.2.8 AA Descriptors

According to Osorio, Rondón-Villarreal and Torres (2015) the AA Descriptor Features (AADF) for a given peptide involve 66 physiochemical descriptors based on AA properties. The descriptors and the corresponding weights they each assign to the 20 natural AAs can be found on the GitHub repository[4] of Pepfeature, and references for these descriptors from where they are taken are available in the documentation[5] of the R package '*Peptides'*.

For a given peptide the calculation of the feature for a descriptor can be expressed as:

$$average = \frac{weighted\ sum}{peptide\ length} = feature\ for\ a\ descriptor$$

In all 66 descriptors' cases what needs to be calculated is the $average$ of the weights of the constituent AA values; this is equivalent to calculating a $weighted\ sum$ with weights given by the descriptors divided by $peptide\ length$ – the length of the peptide string.

As an example, the process of calculating one descriptor's feature (out of 66 features) is illustrated:

[4] Download link to excel file containing all descriptors: https://github.com/essakh/pepfeature/blob/master/pepfeature/data/AAdescriptors.xlsx

[5] https://rdrr.io/cran/Peptides/man/aaDescriptors.html

The AA sequence P is defined as:

P = LLDVVHI

Table 10 demonstrates the calculation of the feature (given by the 'average' in the table) for descriptor pp1.

Table 10: Example of calculating an AA Descriptor feature for the sequence LLDVVHI (read table left to right).

AA characters in *P* (*Char*)	**Total occurrence of *Char* in *P* (O)**	**Weights for each *Char* given by descriptor pp1 (W)**	**O * W**
L	2	-0.9	(2*-0.9=) -1.8
D	1	1	(1*1=) 1
V	2	-1	(2*-1=) -2
H	1	0.67	(1*0.67=) 0.67
I	1	-0.94	(1*-0.94=) -0.94
		Weighted sum	-3.07
		Average	(-3.07/7=) 0.44

3. Project Management and Preparation

3.1 Interactions With Clients

Communication with the main client throughout the development of Pepfeature was solely through email correspondence and with the secondary client through both email and Microsoft Teams meetings. The meetings with Dr. Campelo were audio recorded as well as extensive notes were taken, which helped in recalling information in instances where breaks were taken from working on the final year project. These meetings mainly took the format of a session where I would prepare questions before hand and ask them or engage in consultations with him.

Since the secondary client, Dr. Campelo, is also the supervisor of both Jodie (the main client) and myself, he had a comprehensive awareness of the details and progress of both of our projects and therefore was able to a sufficient extend provide guidance in matters pertaining to Jodie's requirements. For instance, the information and clarifications pertaining to the necessary details regarding Jodie's pipeline were relayed to me by Dr. Campelo – it was more convenient and

efficient to obtain such information through him, as due to his dual role as a supervisor he was able to contextualize and tailor the complex information to my needs.

However, the main client was of course consulted on distinct occasions to obtain feedback regarding the deliverable and whether it met her expectations. Besides the initial contact at the start of the project, the main client was consulted after the very first feature calculation algorithm was coded. In this instance, the main client provided feedback that solidified my understanding of her expectations; as this early implementation was approved with only some pointers to improvement, it gave assurance that the requirements had been interpreted correctly and paved the way for how to approach the rest of the implementations of feature calculation capabilities.

Moreover, in Section 2.1.1 it was expounded that both the clients were involved in the development of the R package – the existing solution in use by the main client –of which Pepfeature is a partial port of. Due to this the clients were naturally already aware of their desired requirements for a Python package replacement, viz. Pepfeature, that delivers practically the same feature extraction functionalities as their own developed *epitopes* R package. Thus, a systematic and intricate requirement elicitation prior to the development of the package was not needed – the initial requirements were handed over and settled over a span of a few email communications between myself and the clients.

Any further changes and expansion to these initial requirements were carried out with the consultation of the secondary client as his guidance was certain to be compatible with the main client needs, and it was more convenient and faster to consult him regarding these matters during our bi-weekly one-to-one supervisory meetings pertaining to the progress of the project.

Furthermore, once the deliverable was judged to be complete and ready to be shown to the clients for their judgement, it was requested from both to test it out, and to gather feedback in a systematic manner, both were asked to answer a set of objective questions through email.

3.2 Contingency Plan and Risk Mitigation

A contingency plan to take account of possible future event or circumstance that could have unfavourable affects and hinderance on the optimum completion of the project was formulated at the commencement of the project. None, of the risks materialised throughout the course of the project. The plan is shown in Table 11.

Table 11: Contingency plan.

Risk	**Mitigation**
Supervisor has to take time off (e.g., due to becoming ill).	Seek support from the university and I shall keep working on my own discretion aiming to get as much of the goals done as possible.
I have to take time off (e.g., due to becoming ill).	The nature of the project allows for many ‘finishing’ points; I shall readjust my goals to aim for a ‘finishing’ point that is within my capacity with the guidance of the advisor.

Data sets get lost completely.	Use a dataset easily available on the internet instead.
Main client (PhD student) becomes unavailable for contacting (e.g., due to becoming ill).	I will contact the advisor who is probably aware of what the client would require at each stage of the project anyways for further guidance.
I lose my progress on the project.	Stick to a plan that requires frequent backups on to the cloud of the progress made – I will be using git for all my files, so there will be incremental backups.
Due to hardware and software issues, I become unable to progress further on the project.	I am expected to mainly only make use of a laptop to complete the project. In the unlikely scenario that my laptop of use for example breaks then I have a backup laptop ready from which I can continue the project promptly.

3.3 Work Schedule

The schedule/timetable shown in Figure 2 was formulated at the commencement of the final year project. Practically the project itself only loosely followed these stages within the stated time frames.

Around a month was dedicated and set aside at the beginning of the project to learn Python. What was learnt during this time were the basics of Python's use and syntax and familiarization with the Pandas and NumPy libraries as these were certainly going to be used for the development of the deliverable. Sufficient skills were learned in this field before proceeding with interactions with the main client and formulation of the requirements.

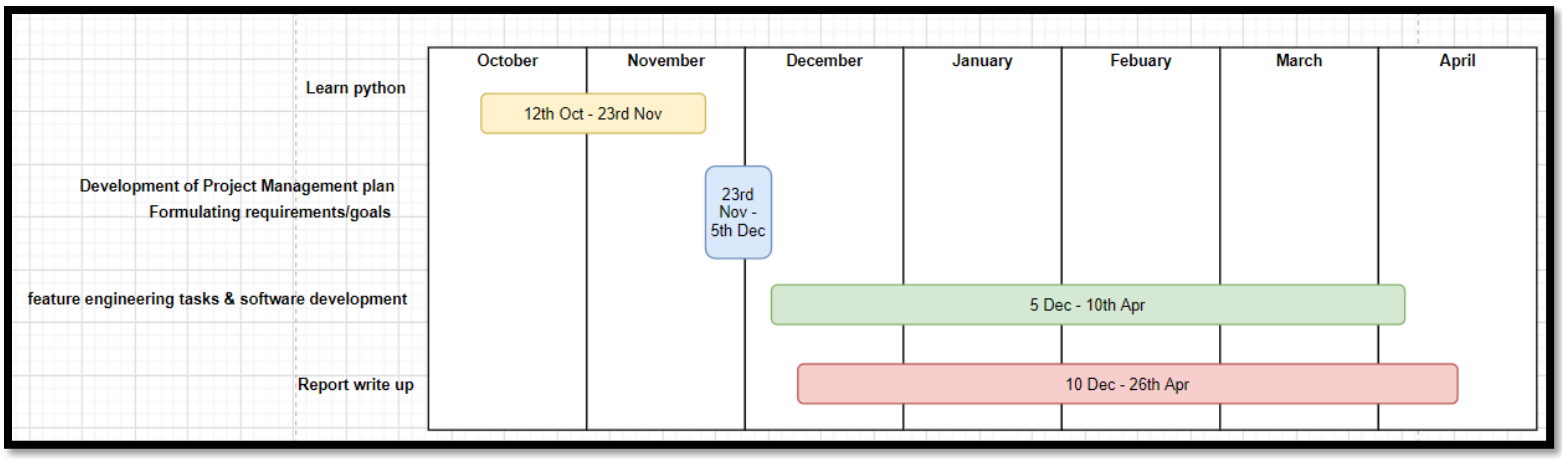

Figure 2: Work Schedule for the entire final year project

3.4 Software Methodology

Analysis, design, implementation, testing and maintenance are the industry recognised stages that the course of development traversed through leading to the final deliverable.

It was decided that an appropriate project development approach would be a sprint based one, Agile, based on bi-weekly sprints.

The project was research-guided such that sufficient information regarding each epitope feature calculation functionality without the biological theoretical information associated with them were explained to me by the supervisor; such that the development of the algorithms to calculate those features could be done purely using first principles.

Moreover, the project being research-guided also meant that explanations of the features to calculate were disclosed to me in increments in the order of priority of development – only gaining the next set of feature calculation functionality details if I confirmed the functioning of the current feature algorithm being worked on as working.

This approach had several benefits. Firstly, since the functionalities to implement are complex and I lacked the domain knowledge and experience in the field of feature calculation, it ensured that I did not overwhelm myself and took an approach to developing the package in a manner such that in the case if time ran out for the implementation of all calculation aimed for, then the deliverable would still have a viable wrapping up point as a consolidated Python package – the only difference being that this would be capable of calculating fewer features than was originally desired.

Moreover, the supervisor was the main developer of the R package *Epitopes*, therefore through his personal domain knowledge he laid out a road map for me to complete the course of feature functionality development in the most coherent and efficient manner; for example, the features ‘k-mer frequencies’ and ‘conjoint triad frequencies’ essentially have the same programmatic principles in their algorithms, therefore it would only be efficient to develop these one after the other rather than have a delay between their implementation by diverting focus on the other complex features and consequently having to waste time re-discovering or re-learning the methods of development.

3.5 Documenting Tasks

Trello[6] boards were used in this project to document progress and keep track of tasks mainly related to the deliverable. Cards (representing tasks or progress notes) were grouped together under a timeframe of 2 week; more specifically, in line with the Agile approach, for these 2-week sprints there would be 3 boards titled: 'To Do', 'Doing' and 'Done' representing each state of the cards (tasks) in terms of progress.

The supervisor would add cards that contain the details of the next feature calculation capability to implement and other relevant miscellaneous stuff, in a separate board titled 'To do (at some point)' and from here these cards would be dragged and dropped into the latest 'To Do' board of the current 2-week sprint. See Figure 2 for a visual of the Trello Board set-up.

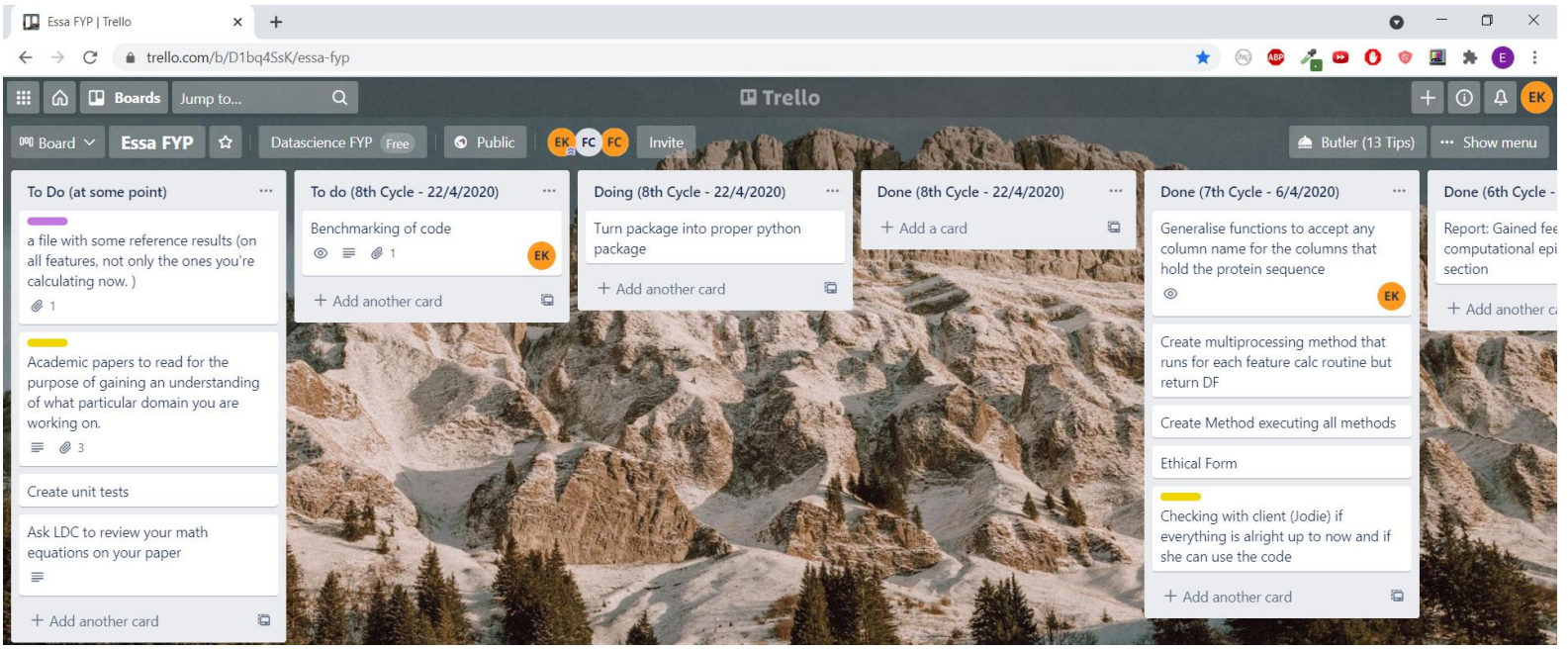

Figure 3: A screenshot of the Trello board for this project showing the set-up of the boards and cards use.

The benefits of the Trello board are organisation related and it can help show the progress of the project to stakeholders as it is essentially a project diary. The Trello board was generally updated at least bi-weekly, primarily following the arranged bi-weekly supervisor meetings.

[6] The Trello board of this project can be accessed at: https://trello.com/b/D1bq4SsK/essa-fyp

3.6 Code Repository

GitHub[7] was used for version control and as the codebase. It is also from where people can read the documentation, which has been explained further in section 4.2.2 of this report.

4. Design and Implementation

4.1 Set Requirements and Formulation

The initial requirements for the Python package Pepfeature were mutually agreed upon with the clients in the form of a MoSCoW[8] hierarchy. These initial requirements just gave a macro-overview of what was required as the clients intended to give me the freedom to choose the Pythonic tools and approach to take to meet these requirements as long as it conforms to be a self-contained Python package in the end. Therefore, during development, through consultations with the secondary client certain design decision – micro requirements – were formulated for the ultimate purpose of achieving these macro requirements stated in Table 12 – these design decisions and the rationale behind them are expounded upon in Section 4.2.

Table 12 presents the macro requirements for the Python Package, along with the priority with what these should be approached with, captured in the 'MoSCoW Rank' column. This priority was set and informed by the clients based on their personal experience of what feature calculation capabilities would likely be the easiest to implement.

From Table 12, the requirements with IDs 1, 3, 4 are necessary to facilitate the user's access to the implementations of the requirements pertaining to the feature calculation capabilities, viz. IDs 2, 5, and are ranked as 'Must Have' – meaning without these foundational requirements the user will be unable to make a convenient use of the feature calculation algorithms that this Package envisions to provide.

'Should have' requirements do weigh as highly as must haves, although they do not necessarily require completion within the current timeframe –but their inclusion in the current timeframe would be excellent.

Furthermore, the reason why the requirement with ID 2 is a 'Must Have' and the one with ID 5 is a 'Should have' is for the purpose of setting a threshold of what inclusion of feature-calculation capabilities would be considered a minimum viable deliverable that has at least some useful and worthwhile routines for the clients to benefit from. Meaning, if the deliverable only ended up covering the requirement with ID 2 (and all the other 'Must Have' requirements) then it would still be deemed a useful product in line with what was envisioned, and such a deliverable can be

[7] The GitHub repository can be accessed at: https://github.com/essakh/pepfeature

[8] MoSCoW Analysis is a prioritization technique that is common for projects such as this, that follow an agile approach with a fixed deadline. The aim is to have emphasis on prioritising the implementation of critical requirements (IIBA, 2009).

wrapped up to be a Python package that calculates some useful epitope prediction related features.

These requirements have been expanded upon to categorise them into functional and non-functional requirements.

Table 12: Pepfeature's Macro Requirements

ID	Title	Functional (F) or Non-functional (NF)	Description	MoSCoW Rank
1.	Read the standardised windowed data (input) coming from the Data Integration block.	F	Python package should facilitate functionality that takes in the input of a (Pandas) data frame.	Must Have
2.	Calculate the frequency-related features	F	Python package should facilitate functionality that calculates the following features on input data: • AA proportion • K-mer frequencies • Conjoint Triads	Must Have
3.	Return a data frame (output) containing the input observations + calculated features with columns in a specific format.	F	Python package should facilitate functionality that outputs the initially input data frame with the features calculated appended as columns.	Must Have
4.	Column names must follow the convention "feat_XYZ" for calculated features	NF	Python package should facilitate functionality that calculates the following features on input data: Column names of the features calculated and appended to inputted Data frame must be named in the format feat_XYZ.	Must Have
5.	Calculate additional features (See 'Description' column for this row.)	F	Python package should facilitate functionality that calculates the following features on input data:	Should Have

			• AA descriptors • Sequence Entropy • Molecular Weight • Number of Atoms (C, H, O, N, S) • Frequency of AA types	
6.	Be able to process up to 10 million observations.	F	Python package should facilitate functionality that allows such memory efficient processing that 10 million observations could be processed in one go.	Could Have
7.	Parallelise calculations on multiple cores.	F	Python package should facilitate functionality that allows multi-processing of the feature calculation functionalities.	Could Have
8.	data-pre-processing of invalid AA sequences	F	Python package should facilitate functionality that does data pre-processing prior to feature calculation: If an AA sequence contains an invalid AA code (B, J, X or Z), the invalid AAs are removed prior to the calculation of any feature.	Could Have

In Table 12, the requirements ranked with a priority of 'Could Have' in the 'MoSCoW rank' column were decided to be only implemented if there is enough resources (time) to do so. These requirements are not essential for the project to be deemed a success.

Besides the functional and non-functional deliverable related requirements in Table 12 another optional requirement set by the secondary client was to benchmark the deliverable's calculation speeds against R package *Epitopes* – this was to elucidate if Pepfeature performs better than this current R package solution used by the main client. Albeit the deliverable performing better than *Epitopes* is not a factor of success, however, it will be an indicator that to what extent the package is useful to the main client.

4.2 Implementation

A Python package named Pepfeature has been developed. Firstly, the main technologies used are expounded followed by an exposition of the underpinning design decisions to meet the macro requirements stated in Table 12. Their relevance to the original requirements and an evaluation of their implementation, along with reflections where appropriate have also been provided in this

section.

4.2.1 Main Python Packages Used

In Pepfeature, Pandas and NumPy are the core packages powering the actual data manipulation and processing of the input date sets for the purpose of feature calculation. Specifically, Pandas facilitated in-memory manipulation of the large datasets through a simple interface and the exportation results to a CSV file. Moreover, the fast column and row selection and the visualization of results with output support that Pandas provides made it very intuitive to code with and provided a visual way to see the results of the feature extraction algorithms, speeding up debugging and development overall.

Pandas utilizes NumPy arrays under the hood for efficient, and fast data manipulation as NumPy has bindings to C (The pandas development team, 2021). There are parts in the implementation of Pepfeature where the NumPy package for its efficient arrays have been used standalone due to their great efficiency in terms of performance and mainly to make use of the optimized built in additional functions that they provide.

Moreover, Python comes with the standard library that provides a wide range of packages that are standardized solutions for many problems that occur in everyday programming; the significant package used from here is 'multiprocessing' for the purpose of parallelising calculations on multiple cores.

4.2.2 Availability on PyPi

Pepfeature is available for download and use from GitHub as a project, and it is also hosted at the Python Packaging Index[9] (PyPI) to download as a consolidated package. This means that the Package is *pip* installable from the console by entering:

pip install Pepfeature

This was not a requirement from either of the clients – the Package being installable from GitHub was sufficient as per them. This is an example where the original expectation of the work is exceeded.

It being installable through PyPi has the main benefit of easy availability and accessibility when users are wanting to use Pepfeature in their own project. Thus, Pepfeature is very easy to install, delete and update to the latest version all using a package installer such as Pip. Additionally, Pip also manages the package's dependencies – ensuring their availability and compatibility upon installing Pepfeature.

Moreover, it also abstracts away the complex code from the user and leaves them with an API to interface with – making it Pythonic - in line with what Python users are generally used to with scientific packages such as Pandas and NumPy.

[9] Pepfeature's PyPi page: https://pypi.org/project/Pepfeature/

Before proceeding with hosting it on PyPi, the secondary client was consulted and gave the advice that as this is a scientific package, therefore the veracity of the outputs produced by the package have to be flawless. Only when it was ensured that the features produced by Pepfeature were accurate (see Section 4.3 for details regarding the verification of the results) was the package hosted on PyPi.

Pepfeature's availability on PyPi contributes to it being a 'consolidated self-contained Python package' - which was an expectation of the clients.

4.2.3 Documentation & installation

On Pepfeature's GitHub repository page's README.md file display window, a comprehensive documentation is provided covering what the Package is, installation instructions, examples of use, API overview and documentation of the API functions and respective parameters. The counterpart package *Epitopes* (R package) does not provide such extensive documentation. Moreover, it was not part of the clients' requirements to provide such comprehensive documentation along with Pepfeature, therefore this an example of where the deliverable has exceeded the Clients expectations.

4.2.4 Package Structure

The file structure of the Pepfeature package is shown in Figure 4. Here the *.py* files with names that start with an underscore character are private modules, viz. *'__init__.py'*, *'_test.py'* and *'_utils.py'* in accordance with the *Python Pep 8 style guide*[10]; these modules are intended to be relevant only to the developers of the package and are thus abstracted away from normal users and nor form part of the public API.

[10] This way was chosen as in practise there is no Pythonic way to denote private modules and enforce such restrictions during compile time. However, there are common conventions to denote modules as private such as by starting their names with an '_' (Python.org, 2013).

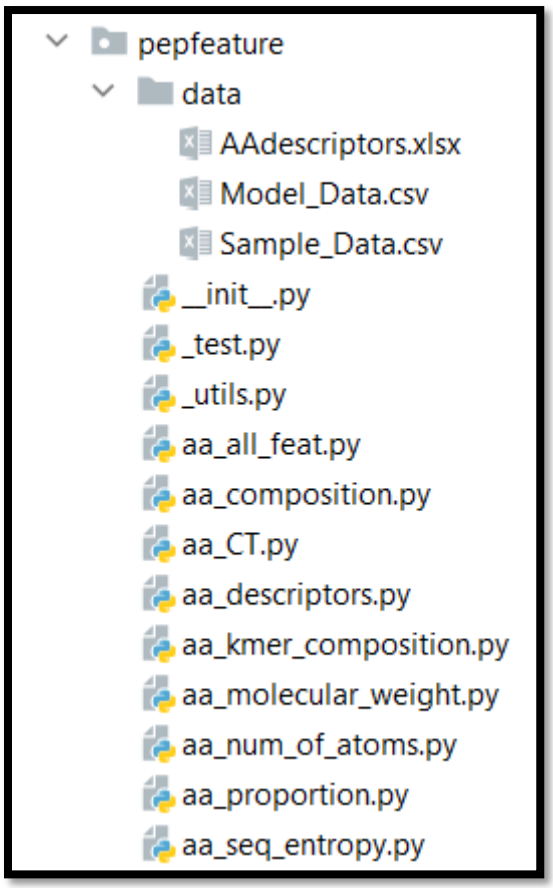

Figure 4: File Structure of Package Pepfeature.

The rest of the modules in the package are public and form part of the API. They are feature calculation modules and have a name starting with “aa”, followed by an indication of what epitope feature(s) (out of the eight possible groups of features) its constituent *‘_algorithm’* function calculates, this is expounded upon in the next Section 4.2.5. The modules with a name starting with “aa” will be referred to as “public” modules for the remained of the report.

Moreover, in Figure 4 it shows that the package also contains a sub directory named *‘data’*. This consists of the excel file *‘AAdescriptors.xlsx’* that holds data pertaining to the individual weights for each AA on each AA descriptor – this file is used in the feature calculation algorithm (i.e., the function named *_algorithm*) held in the *‘aa_descriptors.py’* module. The other two CSV files in the *‘data’* folder are used in the module *‘_test.py’* and their purpose has been explained in Section 5 of this report.

Please refer to either Pepfeature’s documentation, its docstrings in the source code or Table 15 in section 4.2.8 of this report for details of what feature-calculation capabilities are in each of the public modules.

4.2.5 The Functions in Modules

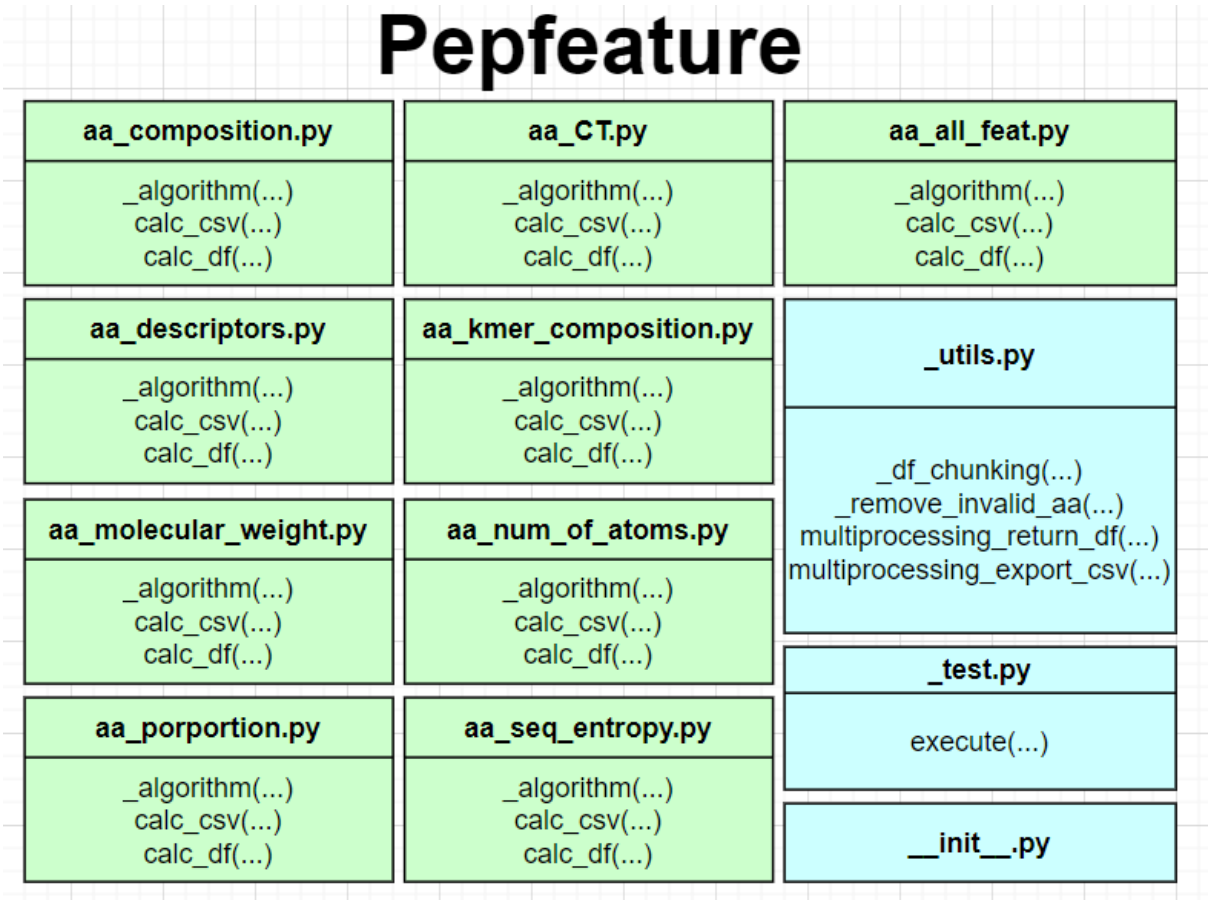

Figure 5: High level overview of Pepfeature's modules and the functions within them. Green rectangles represent public modules and blue rectangles respresent private modules. Only functions that start with the underscore character ('_') in their names are private.

Please refer to either Pepfeature's documentation (on GitHub) or its source code's docstrings for the details omitted (such a parameters) of the functions mentioned in this report.

The Public Modules' Functions

All public modules shown in Figure 5 (green rectangles) contain a private[11] function named *_algorithm*. The *_algorithm* function in each public module contains the code to calculate the feature(s) corresponding with what is indicated as the module name it is contained within; for example, for the module *aa_CT.py,* its *_algorithm* function contains code to calculate conjoint triad frequencies features only. However, the module *aa_all_feat.py* is the odd one out and its *_algorithm* function calculates all the eight groups of features on the input in one go.

[11] In practise there is no Pythonic way to denote private functions and enforce such restrictions during compile time. However, there are common conventions to denote private functions such as starting their names with an '_' (Python.org, 2013).

Each *_algorithm* function in Pepfeature returns a Pandas data frame object with the features calculated appended as columns. The *_algorithm* functions are wrapped by other functions; these wrapper functions are either for the purpose of formulating the public API or to add multiprocessing functionality and/or to add exportation of the data frame produced as a CSV functionality, this is elaborated below.

In these public modules there are two additional functions, viz. *calc_csv* and *calc_df* which are wrapper functions intended to be part of the public API for the users, but the *_algorithm* function is not and is private – only to be used internally within the module.

_util.py's Functions

Contained in the module *_util.py* the function *_remove_invalid_aa* is a helper function for the wrapper-functions' *multiprocessing_return_df* and *multiprocessing_export_csv, and* the function *_df_chunking* is a helper used in *multiprocessing_export_csv* only.

The wrapper functions, *calc_df* and *calc_csv*, present in each of the public modules wrap these functions, viz. 'multiprocessing_return_df' and 'multiprocessing_export_csv' respectively to formulate the API.

Here, the purpose of the wrapper-function *multiprocessing_return_df* is to take in as a parameter the *_algorithm* functions that exist across each of the public modules, and in turn it is executed in this wrapper function with the option of multiprocessing functionality. These functions return the input data frame but with the features calculated as appended.

Similarly, the other wrapper-function, *multiprocessing_export_csv,* takes in as a parameter the *_algorithm* functions that exist across each of the public modules, and in turn the supplied *_algorithm* function is executed in this wrapper function with the option of multiprocessing functionality and instead exports the data frame produced as a *CSV* file saved into a stated location (also passed as parameter to *multiprocessing_export_csv*).

4.2.6 Maintainability and Expansion Potential

The helper and wrapper functions contained within *_utils.py* discussed in the previous section contribute towards the maintainability of the code. The wrapper functions serve as an interface to add functionality to the core code. The core code is contained in the *_algorithm* functions across each of the public modules; here, the wrapper functions add multiprocessing and *CSV* file exportation functionalities. The benefit of this is that the wrapper functions save the developer from the hassle and the mistakes that can slip through by modifying the codes back and forth had these functionalities been implemented without wrapper functions across the 9 public modules in each *_algorithm* function. Moreover, due to the use of these wrapper and helper functions the package, Pepfeature, is given a relatively easily comprehensible structure which contributes to maintainability.

Furthermore, all the modules and their constituent functions are annotated formally using docstrings which also increases maintainability.

The aim of this effort towards maintainability is to make the inclusion of future additions of other feature calculation capabilities straight forward by others. Albeit, this is not a requirement of the clients, nor has extensive effort been put in to facilitate formal protocols for open-source development of the package and in creating the specific kind of documentation that goes along with it. But the current maintainable and structured package lays down the foundations for this,

and users are expected to be able to informally expand the package (through the accessible source code on the GitHub repository) with more feature calculation capabilities beyond the current eight group of features calculated if they wish to do so. This is an example where the original expectation of the deliverable is exceeded. Nevertheless, for a future version of Pepfeature a guide should be provided detailing how to expand the package for the convenience of users.

4.2.7 API Overview

The aim was to make the API intuitive and Pythonic (like the API of the Pandas and NumPy package) – which successfully has been achieved; this can be gauged from Figure 6 which displays the structure of the different interfacing options the user has when writing a line of code using the imported Pepfeature package. The API that the user interfaces is intuitive and simple. Use case examples of the API are given in the examples.py file, available in the GitHub repository[12].

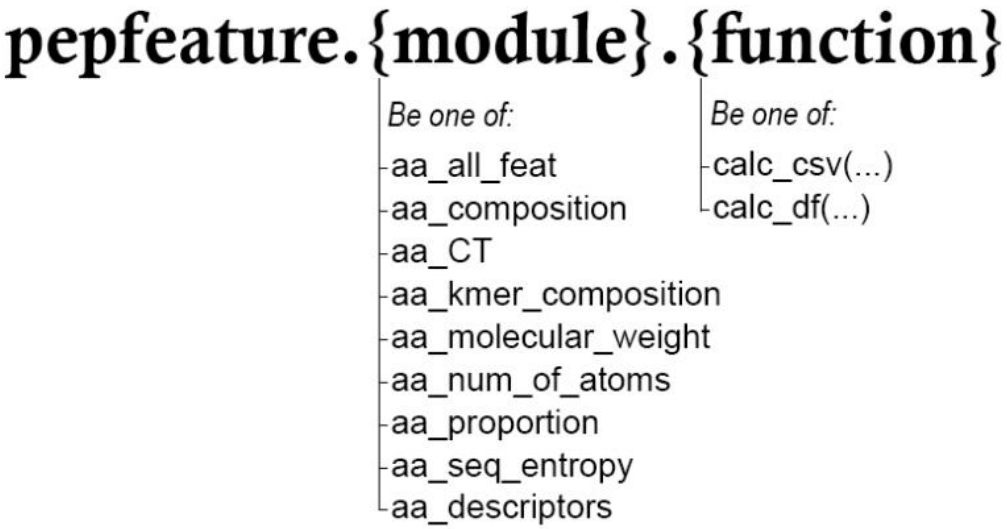

Figure 6: Pepfeature's Public API overview

[12] Available at: https://github.com/essakh/pepfeature/blob/master/examples.py

Table 13: Subset of '*Table 12: Pepfeature's Macro Requirements'* showing requirement with ID 1

ID	Title	Functional (F) or Non-functional (NF)	Description	MoSCoW Rank
1	Read the standardised windowed data (input) coming from the Data Integration block.	F	Python package should facilitate functionality that takes in the input of a (Pandas) Data frame.	Must Have

The requirement with *ID 1* has been reproduced in this section (see Table 13) for easy reference. The API that the user interfaces to use Pepfeature covers this requirement. Essentially, both the API interfacing functions across the public modules, viz. *calc_csv* and *calc_df* accept a data frame to process through their *'dataframe'* parameter. As an example, this can be seen in Figure 7 that shows a function definition of *calc_df* along with its docstring.

```
def calc_df(dataframe: object, Ncores: int = 1, aa_column: str = 'Info_window_seq'):
    """
    Calculates Frequency of AA types for given amino acid sequences

    For each sequence calculates nine features corresponding to the percentage of each Amino Acid type in the sequences

    Results appended as a new columns named feat_perc_{group-value} e.g.  feat_Perc_Tiny, feat_Perc_Small etc.

    :param dataframe: A pandas DataFrame that contains a column/feature that is composed of purely Amino-Acid sequences (pepides).
    :param Ncores: Number of cores to use. default=1
    :param aa_column: Name of column in dataframe consisting of Amino-Acid sequences to process. Default='Info_window_seq'
    :return: Pandas DataFrame

    """
```

Figure 7: Part of the function definition of the function calc_df within aa_composition.py

A point to note is that all *calc_df* and *calc_csv* functions across all public modules accept an argument *'aa_column'* (see Figure 7 for a visual of this). This argument refers to the name of the column in the input data frame that holds the AA sequences to process. Previously, since this is a Python package built around the main-client's prediction pipeline and their column composed of the AA sequences to process was expected to be named "Info_window_seq", so this name was hardcoded in the routines of the package. But after consultations with the secondary client, it was decided that for the package to be more generalised for the rest of the scientific community, the *'aa_column'* argument was added. Moreover, for the convenience of the main client this argument was defaulted to "Info_window_seq".

4.2.8 Feature Calculation Capabilities

In Pepfeature each feature-calculation capability has its own public module named after it, and contained within every such module is a function named '*_algorithm*' which contains the code to calculate that specific group of features. For reference, Table 15 shows which public module corresponds with what feature calculation capability and which explanation it corresponds to from section 2.2 of this report (part of the background research section).

Table 14: Showing which Module in the package corresponds with what feature calculation capability explanation in this Report.

Package's module name	**Section of report expounding the feature and its derivation**
aa_proportion.py	2.2.1 Proportion of Individual AAs in sequence
aa_kmer_composition.py	2.2.2 k-mer Composition
aa_CT.py	2.2.3 Conjoint Triad Frequencies
aa_seq_entropy.py	2.2.4 Sequence Entropy
aa_composition.py	2.2.5 AA Composition
aa_num_of_atoms.py	2.2.6 Number of atoms
aa_molecular_weight.py	2.2.7 Molecular Weight
aa_descriptors.py	2.2.8 AA descriptors

The requirement with ID 2 and 5 has been reproduced in this section (see Table 15) for easy reference. Pepfeature successfully calculate all of the eight different groups of features required from the clients, stated in requirement ID 2 and 5 (see Table 15).

Table 15: Subset of *'Table 12: Pepfeature's Macro Requirements'* showing requirements with ID 2 and 5

ID	**Title**	**Functional (F) or Non-functional (NF)**	**Description**	**MoSCoW Rank**
2	Calculate the frequency-related features	F	Python package should facilitate functionality that calculates the following features on input data: • AA proportion • K-mer frequencies • Conjoint Triads	Must Have

5	Calculate additional features (See 'Description' column for this row.)	F	Python package should facilitate functionality that calculates the following features on input data: • AA descriptors • Sequence Entropy • Molecular Weight • Number of Atoms (C, H, O, N, S) • Frequency of AA types	Should Have

Furthermore, based on discussions with the secondary client it was decided to implement a separate module named *aa_feat_all.py*, and its constituent *_algorithm* function acts as the envelope function that executes all the functions to calculate the eight features in one go, concatenate the results together and output them as a single data frame. This is also an example where the original expectation of the deliverable is exceeded.

4.2.9 The API Functions 'calc_csv' and 'calc_df'

Table 16: A Subset of *'Table 12: Pepfeature's Macro Requirements'* showing requirements with ID 3, 4, 6, 7 and 8.

ID	Title	Functional (F) or Non-functional (NF)	Description	MoSCoW Rank
3	Return a data frame (output) containing the input observations + calculated features with columns in a specific format.	F	Python package should facilitate functionality that outputs the initially input Data frame with the features calculated appended as columns.	Must Have
4	Column names must follow the convention "feat_XYZ" for	NF	Python package should facilitate functionality that calculates the following features on input data: Column names of the features calculated and	Must Have

	calculated features.		appended to inputted Data frame must be named in the format feat_XYZ.	
6	Be able to process up to 10 million observations.	F	Python package should facilitate functionality that allows such memory efficient processing that 10 million observations could be processed in one go.	Could Have
7	Parallelise calculations on multiple cores.	F	Python package should facilitate functionality that allows multi-processing of the feature calculation functionalities.	Could Have
8	Data-pre-processing of invalid AA sequences.	F	Python package should facilitate functionality that does data pre-processing prior to feature calculation: If an AA sequence contains an invalid AA code (B, J, X or Z), the invalid AAs are removed prior to the calculation of any feature.	Could Have

The requirement with *ID* 3, 4, 6, 7 and 8 have been reproduced in this section (see Table 16) for easy reference.

As expounded in section 4.2.5, the functions that the user calls to interface with the package are either *calc_csv* or *calc_df*. *Calc_df* simply returns a Pandas data frame containing the calculated features appended as new columns in the specified format as in requirement ID 4 (Table 16). Thus, the requirements with ID 3 and 4 have been covered.

Under the hood both *calc_csv* and *calc_df* come with multiprocessing functionality and they accept a parameter *'Ncores'* to set the CPU cores to use for this. Thus, the requirement with ID 7 shown in Table 16 has also been covered.

To cover the requirement with ID 6 in the deliverable, it was decided after consultations with the secondary client to develop functionality where the input data frame is processed in chunks and then exported as a *CSV* file as it would be impractical for the average user to load up the entire data frame of 10 million observations into RAM for processing.

Thus, the functions *calc_csv* were created across the public modules to do this. *Calc_csv* under-the-hood is intended to be a memory efficient way of calculating the features, as they are calculated on a single chunk of the input data frame (of *'chunksize'* number of rows – passed as a parameter) at a time and a chunk is deleted from memory – freeing up space - once it has been processed and exported as a CSV. Moreover, it is theorized that if this functionality is paired with the Pandas' *read_csv* function's *'chunksize'* argument – which allows to specify how many rows of data to read into local memory at a time – and these chunks of the larger data frame are input into *calc_csv* (which also in turn has an appropriate *'chunksize'* as its parameter set) then further significant memory efficient performance is expected, however this hypothesis remains to be tested in the future.

Unfortunately, due to a lack of time there has been no attempt made to process a data set of 10 million observations and this should be done at some point. Albeit theoretically this is expected to be possible with Pepfeature, and therefore the requirement with ID 6 (Table 16) is deemed to be covered.

Lastly, a pre-processing step involving the removal of invalid AA characters from the sequences takes place prior to actually calculating the features. This is done in the helper function *_remove_invalid_aa* within the module *_utils*. This covers the requirement with ID 8 (Table 16). However, there lies an inefficiency issue with this function:

The execution of *calc_df* and *calc_csv* in the module *aa_all_feat.py*, under-the-hood, themselves execute the function *_remove_invalid_aa,* but an issue is that they also traverse through all other *calc_df* functions in Pepfeature – executing each of them – and each one of these in turn also execute the *_remove_invalid_aa* function; consequently, the input data frame is looped over multiple times redundantly in an attempt to pre-process. For further efficiency improvements these unnecessarily attempts at pre-processing should be fixed in a future version of Pepfeature. Albeit even with this flaw Pepfeature has shown better performance when benchmarked against the *Epitopes* R package (this is expounded in section 7).

5. Testing

The Pepfeature package contains a sub directory named 'data' (see Figure 4, section 4.2.4 for a visual illustration). This contains two CSV files, viz. Model_Data.csv and Sample_Data.csv. Model_Data.csv is a data set that consists of a column composed of AA sequences, plus all the features calculated on them by the R package *Epitopes*, appended as columns; this model data set – containing accurate feature calculations – was provided by the developers of the *Epitopes* R package to use for verifying the veracity of the results produced by the Pepfeature package.

Sample_Data.csv consists of a single column containing the very same AA sequences as Model_Data.csv. This data set is input into the feature calculation routines of Pepfeature as a data frame to be processed, and in return the same data frame with the features calculated, appended as columns would be outputted.

For the majority of the duration of development of Pepfeature, whenever a newly developed feature calculation capability would need testing (a variation of) Sample_Data.csv would be converted into a Pandas data frame and input into the corresponding *_algorithm* function and following this, the results produced – i.e. the features calculated – would be manually checked against (a variation of) Model_Data.csv for correctness. Since, this was a manual check by the

eye, only a few of the results calculated would be matched against the model data set. If there would be a correct match, then that specific feature calculation capability of Pepfeature would be assumed to produce the correct results and thus be working as intended.

However, as the complexity of the package grew manual checks were not viable anymore and thus it was decided that in an automated manner all the 8 group of features calculated on Sample_Data.csv to be compared with Model_Data.csv for accurateness. The module *_test.py* was created for this. This module is intended to be used by developers whenever they want to ascertain that the main functionality of the Package works; it is especially useful to run after updating the package, to ensure that feature calculation capabilities have not broken.

The module *_test.py* consists of one function, '*execute(Ncores, save_folder)*', to execute the tests; where the parameter *Ncores* is the cores to use for the computation (multiprocessing) and *save_folder* the path to export the resultant CSV to. This function tests three things:

1. For the purpose of generating the features on Sample_Data.csv, the test ultimately under-the-hood executes each of the *_algorithm* functions in Pepfeature through the execution of *calc_df*, the one in *aa_all_feat.py*; thus, if any of these functions (and the functions that these in turn execute themselves under-the-hood) were programmatically (code-wise) broken an error would be raised – and if not they are code-wise sound. However, this is not why *_test.py* was created.
2. Next, the first intended test (Test 1) occurs where the resultant Pandas data frame with the features calculated are then compared with the values in Model_Data.csv for accuracy. Due to a difference of precision in the results produced by both packages, the testing strategy necessitated to round up Pepfeature's results to 3 decimal places before matching and comparing them with Model_Data.csv.
3. Finally Test 2 occurs, the aa_all_feat.*calc_csv*() function with the Sample_Data.csv's derived data frame passed as a parameter is executed. Under the hood, this requires executing each of the *calc_df* functions in the other public modules, but if the unintended test in point one of this list was successful then these would be confirmed as working already. Therefore, this test is to solely check that the *calc_csv* in *aa_all_feat.py* works as intended, this also involves manually checking that the CSV is created in the correct *'save_folder'* location and the results in the CSV are matching with Model_Data.csv at a glance.

From the aforementioned list, from point (3) it is clear that these tests do not test the working of the API function *calc_csv* belonging to the other eight public modules. Moreover, this automated test is not exhaustively testing every parameter. For example, the set *'chunksize'* parameter of *aa_all_feat.calc_csv(…)* is not tested for any other realistic test case.

In future versions a testing strategy should be formulated with the aim of exhaustively testing every interface API function to ensure correctness along with every parameter, such as by using black box testing techniques. Additionally, the testing capability in the module *_test.py* should be converted into a formal unit test, and more unit tests should be created that test other aspects of the system as well; this will also contribute towards facilitating the ease of expansion of the package by others.

The results of Tests 1 and 2 are printed to console; Figure 8 shows an example of the results of Test 1 in a case where all features produced matched with those in Model_Data.csv. It should be noted that the features calculated by Pepfeature are 100% correct.

But in a case where there are non-match cases, the test prints out the column names where there is a mismatch, and these column names can be used to trace where the errors are coming from

(i.e., which of the *_algorithm* function across the public modules is making the erroneous calculations).

```
-------------TEST 1----------------------------------------
Result of test of creating DF (on Sample_Data.csv) consisting ALL feature calculated (using aa_all_feat.calc_df()):
-----------------------------------------------------
845 Matched features with Model_Data.csv.

0 Umatched features with Model_Data.csv.
Thus, 100% accurate results produced by this package.
------------------TEST 2-----------------------------------
For testing sake: re-creating DF and exporting as CSV (using aa_all_feat.calc_csv()) ...
CSV created, please check the CSV that has been created in set folder location for accuracy
--------------------Test ENDED---------------------------------
```

Figure 8: Output of _test.py's execute(…) function to console in the case where there is a 100% match of features between those generated by Pepfeature and those in Model_Data.csv.

Moreover, anything that ***_test.py's*** *execute* function does not test, has been loosely tested manually for common use cases throughout development stages and it can be said with reasonable confidence that all functions of the package work for at least common use cases. The confidence has increased after both clients tested the package for normal use with no issues.

5.2 Pepfeature Uncovering Flaws in Epitopes (R package)

There are two instances where the development of Pepfeature uncovered minor flaws in the *Epitopes* package; these flaws were reported to the package maintainer (Dr. Campelo) who took note of them for potential correction in future versions of the package. This is an instance where the original expectation of the deliverable has been exceeded. These flaws are explained as follows:

Firstly, during the development of Pepfeature an inconsistency was realised in some of the column names of the data set that hold the features generated by *Epitopes* (the Model_Data.csv file). Instances where features would be a proportion (out of 1) and not a percent (out of 100) the column names would start with "feat_**Perc**" – the name was not representative of the feature values. Such an inconsistency was not present in Pepfeature's generated features – their corresponding feature's column names would start with "feat_**Prop**". Therefore, wherever there was an inconsistency in the names of the columns in Model_Data.csv this was manually renamed and corrected for the purpose of running the tests in *_test.py* module, as it matches values by column names.

Secondly, in Pepfeature a pre-processing step of the AA sequences in the input data frame takes place, which removes invalid characters from each AA sequence before commencing the calculation of the feature(s). It had to be tested whether Pepfeature also generates correct results if the input data frame includes invalid AA sequences.

Therefore, in one instance invalid AA sequences were added to Sample_Data.csv and likewise these same AA sequences and their calculated features (by *Epitopes*) were handed over by Dr. Campelo to add to Model_Data.csv.

```
-------------TEST 1-----------------------------------------
Result of test of creating DF (on Sample_Data.csv) consisting ALL feature calculated (using aa_all_feat.calc_df()):
------------------------------------------------------------
845 Matched features with Model_Data.csv.

80 Umatched features with Model_Data.csv.
Names of unmatched Model_Data.csv features: ['feat_seq_entropy', 'feat_Prop_Tiny', 'feat_Prop_Small', 'feat_Prop_Aliphatic', 'feat_Prop_Aromatic', 'feat_Prop_NonPolar', 'feat_Prop_F
-------------------TEST 2-------------------------------------
For testing sake: re-creating DF and exporting as CSV (using aa_all_feat.calc_csv()) ...
CSV created, please check the CSV that has been created in set folder location for accuracy
--------------------Test ENDED---------------------------------
```

Figure 9: Output of _test.py's execute(…) function to console in the case where calculations on invalid AA sequences were matched between those generated by Pepfeature and those in Model_Data.csv

When the tests in *_test.py* was executed a mismatch of these results occurred between both. Upon a thorough investigation it was concluded that the fault lies in the features calculated of these invalid AA sequences by the *Epitopes* R package. Thus, the development of Pepfeature lead to the uncovering of erroneous results produced by *Epitopes*. Figure 9 shows the test results in the instance where this mismatch occurred.

6. Maintenance

There have been instances of modification of the Package after delivery to the client to correct faults. One example of this is that soon after Pepfeature was deemed to be complete and was hosted on *PyPi*, the main client was asked to test out the Package. However, after trying to run the deliverable the client notified about it not working on her end and certain package dependency issues occurring. This issue was investigated and later found out to be due to incompatible versions of the dependencies on her machine. This issue was resolved by specifying the required dependencies and version in the *setup.py*[13] file, in the package directory; this now ensures that whenever the package is installed using *pip* any required dependencies are installed to the correct version.

Moreover, on the Pepfeature's public documentation (on GitHub) it has been requested from users to report any bugs or glitches they find and welcomes feedback. This opens the potential for further opportunities for the package's maintenance.

7. Benchmark: Pepfeature vs. Epitopes (R package)

The strategy for benchmarking was to calculate all features on five data sets consisting of a column composed of AA sequences containing 10, 50, 100, 500 and 1000 observations. These were fed into the respective umbrella functions from both Pepfeature and the R package three times to calculate all the eight features and timed for the execution to complete. Both of the

[13] Setup.py file can be found here: https://github.com/essakh/pepfeature/blob/master/setup.py

benchmarking scripts[14] for each of the packages along with the datasets can be found in the 'benchmark' directory on Pepfeature's GitHub repository[15].

Both benchmarking scripts were executed under the same conditions. The system used was 'iMac 3.6 GHz Quad-Core Intel Core i7, 16Gb RAM running Mac OS 11.2.3' and the Python script was running under Python 3.7.4 and the R script under version 4.0.5; Pepfeature version 1.0.8 was used.

The results of the benchmark are reproduced in Table 18 and 19.

Table 17: Benchmark results of Python package Pepfeature.

AA observations in Data frame (size)	**Minimum execution time out of 3 runs (seconds)**	**Mean execution time of 3 runs (seconds)**	**Maximum execution time out of 3 runs (seconds)**
10	1.751	2.243	3.200
50	3.561	3.578	3.595
100	5.428	5.542	5.612
500	20.897	21.341	21.888
1000	39.967	40.582	41.515

Table 18: Benchmark results of R package Epitopes.

AA observations in Data frame (size)	**Minimum execution time out of 3 runs (seconds)**	**Mean execution time of 3 runs (seconds)**	**Maximum execution time out of 3 runs (seconds)**
10	0.793	0.797	0.805
50	3.270	3.320	3.340
100	6.380	6.600	6.760
500	31.700	34.200	36.100
1000	69.100	76.600	88.600

A graph plotting size of the dataset vs. minimum time is plotted for visual comparison in Figure 10. Minimum time has been deemed more appropriate to plot on the assumption that the minimum execution time gives a lower bound for how fast the machine can run the functions and that higher values in the results are most probably not caused by variability in the programming language's speed, but by other active processes on the machine interfering with the timing (docs.python.org, n.d.).

[14] The R package's benchmarking script was provided courtesy of Dr. Campelo.

[15] Available at: https://github.com/essakh/pepfeature/tree/master/benchmark

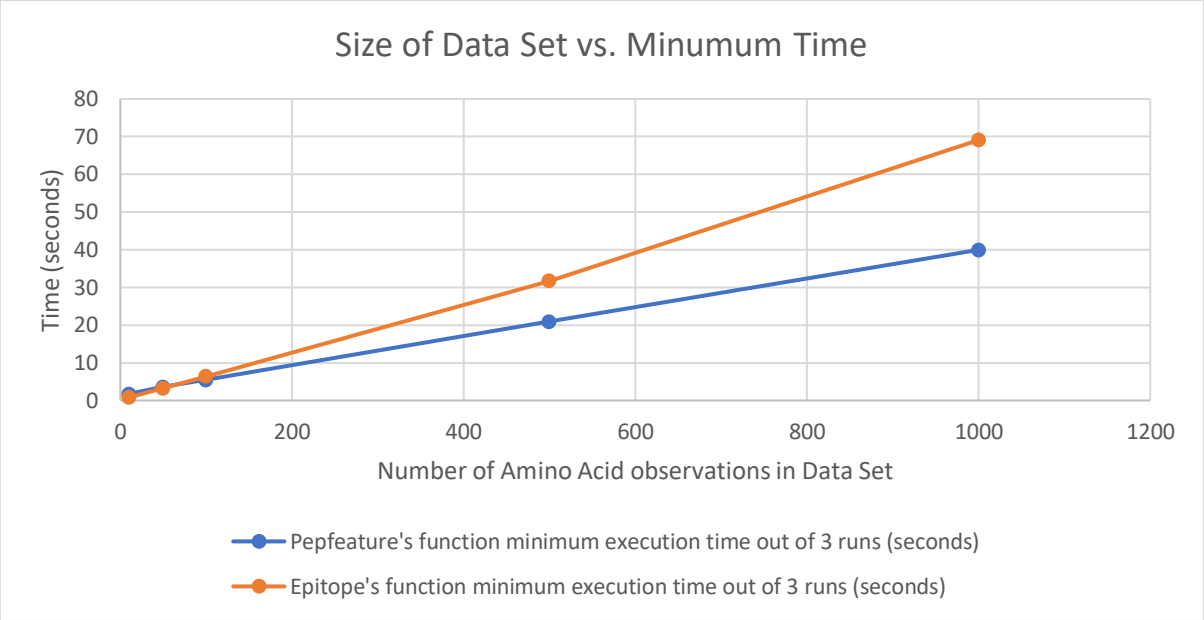

Figure 10: Graph of Size of Data Set vs. Minimum Time

From the results in Table 18, 19 and the trend illustrated in Figure 10, it can be concluded that Pepfeature is roughly 50% more time efficient when calculating all of the eight features on the largest data set (1000 observations), and it is reasonable to expect this trend to continue even for larger data sets.

In Figure 10 it can be seen that there is a smaller time efficiency difference for the smaller data sets (10, 50, 100 observations), this is very likely due to the inefficient runtime overhead in Pepfeature caused by the usage of the dependency – a package – 'pkg_resources' in module *aa_descriptors.py*; this dependency unnecessary builds a working set of *all* installed packages (taking up extra time during runtime), even though only the packages that are in use are needed to be loaded up (setuptools.readthedocs.io, n.d.). The unnecessary overhead this creates is negligible in practical terms, but it should be resolved in a future version of Pepfeature.

The intended users of this package are likely to use it for feature calculation of very large datasets consisting of thousands of observations and the time efficient improvements over *Epitopes* that this package has displayed is thus excellent. This is an instance where the original expectation of the deliverable is exceeded.

Other useful benchmarks to conduct would be to determine which of the two packages are more memory efficient. Furthermore, to conclusively proof the memory efficient benefits that Pepfeature's *calc_csv* functions are intended to have (as explained in section 4.2.9 of report), a benchmark would need to be conducted.

8. Feedback from Clients

Both the secondary-client and main-client were asked to use version 1.0.8 of Pepfeature. A set of objective questions had been asked by email to gauge both of their feedback. The set of questions asked from both are very similar but have been tailored to their specific interests in the package. These questions are reproduced in Appendix 1.

Here under is a summary of the positive things mentioned about Pepfeature by the main-client, Jodie (2021):

- She appreciated and praised the comprehensive documentation on GitHub. Especially, the graphics that have been provided for the API as they clearly show the functions that are available.
- She personally tested out calculation on a small set of observations and pointed out that the speed of the calculations are “good” and that all features are calculated “correctly” and “quickly”.
- She deemed the API relatively easy to use and fit for purpose.
- She said that the package allows for the calculation of all the desired features.
- She appreciated the modular structure of the package.
- She appreciated how “well documented” the source code is, which made it very easy to read for her.
- She confirmed that the input-output structure of the package is exactly what is needed for its intended use in the prediction pipeline.
- She confirmed the package’s suitability for general use (by others) because in the API functions, *calc_csv* and *calc_df*, as parameters, accept the name of the *‘aa_column’* and the length (k) for the k-mer features, along with the fact that the routines are able to handle inputted AA sequences of varying sizes.
- She is happy with the feature calculation capabilities and the package in general.

Here under is a summary of the positive things mentioned about Pepfeature by the secondary client, Campelo (2021):

- He appreciated that Pepfeature (an in-house built package) is general enough for use by researchers at Aston as well as by other researchers developing epitope prediction tools.
- He stated that that the package seems robust and successfully implements several common sequence-based features that are useful for the prediction of linear epitopes.
- He stated that that the deliverable meets expectations in terms of usability, ease of use and performance.
- He stated that Pepfeature outperforms *Epitopes* and thus will be useful in Jodie’s prediction pipeline.
- He stated that it is flexible enough to allow further development in the future, and potentially add additional features based on other statistical properties of the sequence, as well as biologically based features.

It is noteworthy that the sentiments expressed throughout the preceding Section 4 regarding Pepfeature meeting expectations and requirements have been held by both clients as well.

Moreover, neither client had anything negative to say. However, further potential improvements for the package were suggested by both clients, and are summarised as follows:

- Give the user some feedback to let them know that the features are being calculated. (This has been noted down to add in a future release of Pepfeature – perhaps some sort of progress bar being printed on the console.)
- For the functions *calc_csv*, along with the current save path that is passed in as a parameter, allow for a file name to be passed in too so that you one can choose the name of the output CSV file, rather than it being auto generated. (This has been noted down to add in a future release of Pepfeature.)

- Add automated unit testing to ensure that the calculations remain correct after updates. (Exhaustive unit tests are planned for a future version; the need for this had already been determined by myself, see section 4.4 for more information.)
- Remove any commented-out code blocks in the source code for the purpose of neatening it. (This will be done promptly in a future release.)

9. Conclusion

Pepfeature, an in-house Python package has been successfully developed. It consists of functions for calculating eight sequence-based set of features for linear B-cell epitope prediction purposes. The way it was implemented makes it flexible enough for use not only by the main client at Aston, but potentially by other researchers developing epitope prediction tools.

Objectively and as per the clients, the package has covered all the set requirements and met all expectations. Some of the key areas where Pepfeature has exceeded expectations is that it is hosted on PyPi; the comprehensive documentation availability; its development leading to the unintentional uncovering of flaws in the results produced by *Epitopes* – the R package counterpart of Pepfeature – and that it outperformed in benchmarks against it by a comfortable margin, which makes it an asset for the main client's predictive pipeline.

This report has covered the process of creating Pepfeature, including the preparation, project management, the implementation, and an evaluation of the deliverable.

Appendices

Appendix 1: Questions asked to Clients to gather their feedback

Asked to main client:

Q. Whether the input-output structure of my package is convenient for the intended use in the pipeline, and as a general tool for epitope prediction?
Q. Whether the calculation speed is as expected?
Q. If there are any other features that could be added in future versions?
Q. Any other feedback you wish to share?

Asked to secondary client:

Q. Whether the package is sound as a general tool for epitope prediction?
Q. Whether the calculation speed is as expected?
Q. Do you think that the deliverable has met expectations/not-met expectations and why?
Q. Do you think that there are any other features that could be added in future versions, and by looking at the package it seems easy to add new feature calculation capabilities?
Q. Any other feedback/improvements you wish to share?

References List

Amino Acids Reference Charts. Available at: https://www.sigmaaldrich.com/life-science/metabolomics/learning-center/amino-acid-reference-chart.html (Accessed: 1 May 2021).

Ashford, J. (2021) [Feedback] Email to Essa Khan, 27 April.

Biological Magnetic Resonance Data Bank: Chemical Shift Statistics table. Available at: https://bmrb.pdbj.org/ref_info/aadata.dat (Accessed: 1 May 2021).

Campelo, F. (2020) [Prediction Pipeline Figure] Email to Essa Khan, 25 November.

Campelo, F. (2021) [Feedback] Email to Essa Khan, 30 April.

Campelo, F., Ashford, J. and Lobo, F. (2021). Epitopes package. [online] Github. Available at: https://github.com/fcampelo/epitopes [Accessed 25 Feb. 2021].

docs.python.org. (n.d.). timeit — Measure execution time of small code snippets — Python 3.9.0 documentation. [online] Available at: https://docs.python.org/3/library/timeit.html.

EL-Manzalawy, Y. and Honavar, V. (2010). Recent advances in B-cell epitope prediction methods. Immunome Research, 6(Suppl 2), p.S2.

IIBA (2009). A Guide to Business Analysis Body of Knowledge (BABOK Guide). Toronto, On, Canada International Institute Of Business Analysis.

Jespersen, M.C., Mahajan, S., Peters, B., Nielsen, M. and Marcatili, P. (2019). Antibody Specific B-Cell Epitope Predictions: Leveraging Information From Antibody-Antigen Protein Complexes. Frontiers in Immunology, [online] 10, p.3. Available at: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6399414/ [Accessed 3 Dec. 2019].

Korstanje, J. (2020). Is Python faster than R? [online] Medium. Available at: https://towardsdatascience.com/is-python-faster-than-r-db06c5be5ce8#:~:text=The%20total%20duration%20of%20the [Accessed 13 Jan. 2021].

Kunft, A., Katsifodimos, A., Schelter, S., Breß, S., Rabl, T. and Markl, V. (2019). An intermediate representation for optimizing machine learning pipelines. Proceedings of the VLDB Endowment, 12(11), pp.1553–1567.

Osorio, D., Rondón-Villarreal, P. and Torres, R. (2015). Peptides: A Package for Data Mining of Antimicrobial Peptides. The R Journal, [online] 7(1), p.4. Available at: https://journal.r-project.org/archive/2015/RJ-2015-001/RJ-2015-001.pdf.

Pande, A., Patiyal, S., Lathwal, A., Arora, C., Kaur, D., Dhall, A., Mishra, G., Kaur, H., Sharma, N., Jain, S., Usmani, S.S., Agrawal, P., Kumar, R., Kumar, V. and Raghava, G.P.S. (2019). Computing wide range of protein/peptide features from their sequence and structure.

Paull, M.L., Johnston, T., Ibsen, K.N., Bozekowski, J.D. and Daugherty, P.S. (2019). A general approach for predicting protein epitopes targeted by antibody repertoires using whole proteomes. PLoS ONE, [online] 14(9). Available at: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6730857/ [Accessed 1 May 2021].

Potocnakova, L., Bhide, M. and Pulzova, L.B. (2016). An Introduction to B-Cell Epitope Mapping and In Silico Epitope Prediction. Journal of Immunology Research, [online] 2016, pp.1–11. Available at: https://pubmed.ncbi.nlm.nih.gov/28127568/ [Accessed 21 Oct. 2020].

Python.org. (2013). PEP 8 -- Style Guide for Python Code. [online] Available at: https://www.python.org/dev/peps/pep-0008/.

Sanchez-Trincado, J.L., Gomez-Perosanz, M. and Reche, P.A. (2017). Fundamentals and Methods for T- and B-Cell Epitope Prediction. Journal of Immunology Research, [online] 2017, pp.1–14. Available at: https://www.hindawi.com/journals/jir/2017/2680160/.

setuptools.readthedocs.io. (n.d.). Package Discovery and Resource Access using pkg_resources — setuptools 56.0.0 documentation. [online] Available at: https://setuptools.readthedocs.io/en/latest/pkg_resources.html [Accessed 27 Apr. 2021].

Shen, J., Zhang, J., Luo, X., Zhu, W., Yu, K., Chen, K., Li, Y. and Jiang, H. (2007). Predicting protein-protein interactions based only on sequences information. Proceedings of the National Academy of Sciences of the United States of America, [online] 104(11), pp.4337–4341. Available at: https://pubmed.ncbi.nlm.nih.gov/17360525/ [Accessed 30 Apr. 2021].

The pandas development team (2021) Pandas (1.2.4) [Python Package]. Available at: https://pandas.pydata.org/ [Accessed: 30 April 2021].

Wang, H.-W. and Pai, T.-W. (2014). Machine learning-based methods for prediction of linear B-cell epitopes. Methods in Molecular Biology (Clifton, N.J.), [online] 1184, pp.217–236. Available at: https://pubmed.ncbi.nlm.nih.gov/25048127/.

Wang, H.-W., Lin, Y.-C., Pai, T.-W. and Chang, H.-T. (2011). Prediction of B-cell Linear Epitopes with a Combination of Support Vector Machine Classification and Amino Acid Propensity Identification. [online] Journal of Biomedicine and Biotechnology. Available at: https://www.hindawi.com/journals/bmri/2011/432830/ [Accessed 20 Dec. 2020].

Yang, L., Xia, J.-F. and Gui, J. (2010). Prediction of Protein-Protein Interactions from Protein Sequence Using Local Descriptors. Protein & Peptide Letters, 17(9), pp.1085–1090.