

Master Informatique  
Parcours Bioinformatique et Modélisation

**Project Report**

**Studying the evolutionary history of a B cell lineage**

**Authors:**

Brenda Enriquez Rodriguez

Ekaterina Gaydukova

**Supervisors:**

Juliana Silva Bernardes

Nika Abdollahi

22 may 2022

# Abstract

**Background.** B cells are a type of lymphocyte that are responsible for the humoral immunity component of the adaptive immune system. These white blood cells produce antibodies, which play a key part in immunity response. This mechanism is highly specific for each invader because of a unique B cell receptor (BCR) that can differentiate friendly bacteria from deadly pathogens by their unique part called antigens. The evolutionary changes of BCR create B cells which have a stronger and more specific response to the antigen and these changes might be represented as the BCR lineage tree.

**The main goal** of our project was to develop phylogenetic metrics in order to describe the properties of BCR lineage trees. As a secondary goal, we used these metrics to compare the performance of two algorithms that generate BCR lineage trees and to search the similarities into BCR lineage trees of patients with different haematological diseases.

**Methods.** We used ClonalTree and GCtree algorithms that might reconstruct BCR lineage trees. As a data, we used repertoires of immunoglobulin heavy chains (IgH) coming from two different sources: simulated by GCtree repertoires and another one from patients with haematological disorders (chronic lymphocytic leukaemia and Waldenstrom macroglobulinemia).

**Results.** We developed ten metrics that might be applied in search of the similarities in IgH repertoires of patients with haematological diseases (chronic lymphocytic leukaemia and Waldenstrom macroglobulinemia). Comparisons of BCR lineage trees could be performed for IgH repertoires from the different patients, and for one patient's repertoires in different time points. In providing the meaningful information of the BCR lineage trees, the capability of each metric was dependent on initial data and repertoires' diversity.

**Conclusion.** In our work, we have developed phylogenetic metrics that could have capability in description of BCR lineage trees of patients with haematological diseases and might be sensitive to BCR lineage tree changes over the time.

## Acknowledgements

We are highly indebted to our supervisors, Juliana Silva Bernardes and Nika Abdollahi for their guidance and supervision during this project, as well as for providing necessary information regarding the project and for their kind support and motivation. Constant guidance and willingness to share their vast knowledge made us understand this breathtaking world of B cells with wonderful trees and present this project as it shows.

We also want to recognize the work of all the team who works on BCR intraclonal diversity for making this project possible by providing us data of different BCR lineages.

# Contents

<b>1 Introduction</b>	<b>4</b>
<b>2 Methods</b>	<b>6</b>
2.1 ClonalTree algorithm	6
2.2 Phylogenetic metrics	8
2.3 Data	10
<b>3 Results</b>	<b>11</b>
3.1 Analysis of simulated BCR lineage trees	11
3.2 Analysis of categorised BCR lineage trees	13
3.3 Analysis of uncategorised BCR lineage trees	16
<b>4 Discussion</b>	<b>18</b>
<b>5 Bibliography</b>	<b>19</b>
<b>6 Appendices</b>	<b>20</b>
6.1 Categorised dataset	20
6.2 Visualisation of trees of different categories	21
6.3 Code	25

# 1 Introduction

B cells are the essential part of the adaptive humoral immune system. They are responsible for identifying antigens and mediating the production of antigen-specific immunoglobulin (Ig) directed against invasive pathogens (antibodies) through molecules called B-cell receptors (BCR).

Immunoglobulins are protein heterodimers consisting of two identical H (heavy) and two identical L (light) chains. Each chain is composed of a constant C region and a variable region. Variable regions are encoded by the Variable (V), Diversity (D) and Joining (J) genes, which are rearranged through a genetic mechanism known as V(D)J recombination. This mechanism allows the formation of a BCR repertoire that can theoretically recognise all possible foreign molecules [1].

Constantly changing antigenic landscape implies constant modifications of the immune repertoire. When exposed to an antigen, the naïve B-cell genes encoding immunoglobulin undergo multiple rounds of mutations, known as somatic hypermutations [2] (SHM) and Darwinian selection. This stage of B-cell development is known as affinity maturation, as it gradually increases the affinity of Ig to the pathogen-associated antigen. The naïve B-cell and its variants produced during affinity maturation form the B-cell lineage [1]. Natural selection that occurs during affinity maturation selects B cells with higher affinity for IG antigens. The analysis of mutations within a BCR lineage might be useful to describe the process of clonal development. This knowledge might be important in the processes of developing accurate vaccines, discovering therapeutic monoclonal antibodies, or understanding B-cell tumours [3].

Genetic evolution, such as that observed in affinity maturation, is often studied using phylogenetic inference, a well-known methodology that describes the evolution of related DNA or protein sequences in different species. Theoretically, we can use classical phylogenetic tree reconstruction algorithms for the BCR lineage, replacing species with BCR sequences with different mutations. However, there are several reasons why conventional phylogenetic tree algorithms are not suitable for the reconstruction of BCR lineage trees [4]. Firstly, in a phylogenetic tree the root is usually unknown, but in a BCR lineage tree the root

or BCR sequence of the naive B-cell that gave rise to the lineage can often be predicted. Furthermore, the observed sequences are represented usually only in the leaves of classical phylogenetic tree, and the inner nodes represent the relationships amongst sequences while in a BCR lineage tree the observed BCR sequences can be leaves or internal nodes in the tree because the selection process during affinity maturation tolerates the coexistence of cells with various levels of antigen affinity and therefore different BCR mutations. Moreover, IG sequences are under intense selective pressure, and the neutral evolution assumption is invalid. Finally, in a phylogenetic tree, the context-dependence of SHM violates the assumption that sites evolve independently and identically [2]. Some computational tools have been developed specifically for reconstructing BCR lineages, for example, GCtree [5].

ClonalTree [4] is a new method to reconstruct BCR lineage trees, combining minimum spanning tree (MST) [6] algorithm and genotype abundance to infer maximum parsimony trees. ClonalTree has a lower time complexity compared with other algorithms and great potential for many applications, particularly in clinical settings where time constraint is important.

The aim of the project is to identify certain significant phylogenetic metrics through the ClonalTree algorithm. These metrics might describe the reconstructed trees. As a second objective, we would like to compare the trees obtained with ClonalTree and GCtree in order to identify differences between the resulting tree, and propose possible causes. Finally, we will analyse different IgH repertoires coming from patients with different pathologies that are characterised by significant clonal expansion, leading to potential big trees. Some of the used IgH repertoires obtained from patients with chronic lymphocytic leukaemia and others from patients with Waldenström macroglobulinemia. Those IgH repertoires were collected during routine diagnostic procedures.

## 2 Methods

### 2.1 ClonalTree algorithm

ClonalTree reconstructs BCR lineage trees based on the MST algorithm and genotype abundances. ClonalTree uses a set of observed BCR sequences with the same rearrangement event and looks for a minimum-sized directed tree structure (B-cell lineage) that might represent the affinity maturation process. Nodes represent BCR sequences, and the weight of edges connecting nodes represents the distance between sequences in terms of mutation, insertion, and deletion operations. All observed sequences are reachable from the root, e.g., the naive BCR sequence.

ClonalTree performs a multiple sequence alignment to consider evolutionary events. Next, it computes a similarity-weighted hamming distance between each pair of sequences. This pairwise distance is then used as the weight edge connecting two sequences.

For reconstruction of BCR lineage trees, ClonalTree uses modified Prim's algorithm[7]. Starting at the root, ClonalTree adds all root's neighbours with minimum edge weight to a priority queue. On the next step, it iteratively extracts from the priority queue the node with the lowest edge weight and highest genotype abundance. If no cycle is formed, the node and the edge will be added to the tree. Following this, all the neighbours of the added node with minimum edge weights are included in the priority queue. Process of adding nodes and edges will continue until all nodes will not be covered. ClonalTree adds each node only once at the priority queue. If a set of edges have the same weight, ClonalTree will choose the one that connects nodes with high abundance.

Once the BCR lineage tree has been built, ClonalTree algorithm modifies the obtained tree. The process of editing implements two strategies: add unobserved intermediate nodes to the tree and detach/reattach sub-trees. Unobserved internal nodes might represent unobserved sequences that were not sampled or disappeared during the affinity maturation. In those cases, the evolutionary relationships were also lost. One way to recover them is to analyse the reconstructed tree to identify common ancestors not yet represented. ClonalTree adds an unobserved internal node when it detects a missing common ancestral in the tree. It generally happens when a distance between sister nodes is smaller or equal to the distance for their parent. As the second step of editing the obtained tree, this algorithm can detach a sub-tree

from an internal node by removing its edge and reattaching it to another internal node or leaf. ClonalTree performs this editing operation if it can reduce the depth of the lineage tree by keeping the overall cost. The goal is to try to find the most parsimonious tree.

ClonalTree takes as input a sequence file in fasta format. FASTA format is a text-based format for representing either nucleotide sequences or amino acid (protein) sequences, in which nucleotides or amino acids are represented using single-letter codes. Reconstructed tree returns in a Newick format. Newick tree format is a way of representing graph-theoretical trees with edge lengths using parentheses and commas. The second output file in csv format has information about the branches of the tree and the lengths of these branches. A comma-separated values (CSV) file is a delimited text file that uses a comma to separate values.

In order to represent trees, ClonalTree has a visualisation module. The module requires a sequence file in fasta format, the tree obtained by the algorithm in Newick format and a file with branch lengths of the resulting tree in csv format. This module returns a json file that is used to visualise trees in two possible formats: Circle and Elbow tree (Figure 1). JSON (JavaScript Object Notation) is a file format that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and arrays.

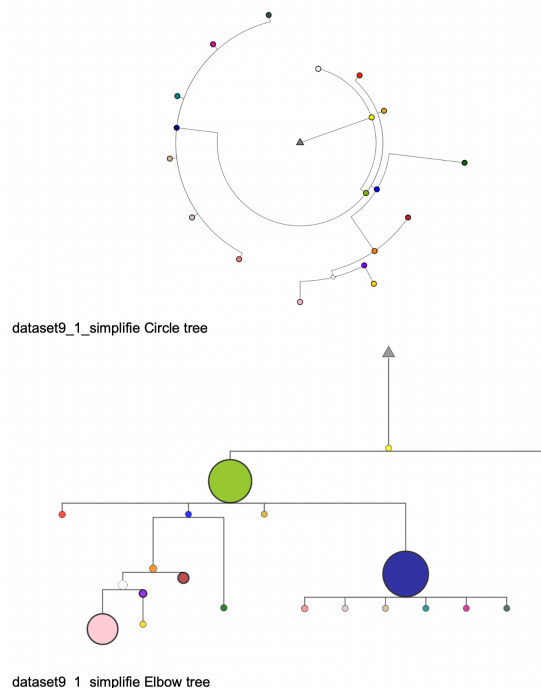


Figure 1: **Visualisation in Circle and Elbow tree format.** Used dataset9\_1\_simplifie.

## 2.2 Phylogenetic metrics

ClonalTree provides the possibility to construct and represent trees for a wide range of B-cell repertoires. In order to be capable of interpreting these trees, various phylogenetic metrics may be applied. During this project, we have developed and implemented in a ClonalTree algorithm calculation of certain metrics that might describe a reconstructed tree (Table 1).

Richness metrics sum up the quantity of phylogenetic differences present in an assemblage, and we can further distinguish metrics according to the type of basic units they sum across. The aim is to understand how much evolutionary history is associated with a set of sequences. We implemented the Phylogenetic diversity metric (PD) [8][9]. PD is a measure of biodiversity based on phylogeny and it was defined as the phylogenetic diversity of a set of species that is equal to the sum of the lengths of all those branches on the tree that span the members of the set (Figure 2). The branch lengths on the tree are informative because they count the relative number of new features arising along that part of the tree.

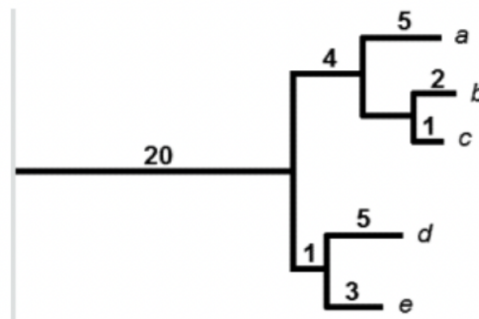


Figure 2: **Phylogenetic tree with PD = 41.** Branch lengths are shown above branches.

The PD is 41 for this set of species  $(20 + 5 + 4 + 2 + 1 + 5 + 1 + 3)$ . [9]

The divergence dimension contains metrics that average the distribution of units extracted from a phylogenetic tree. The aim is to understand how closely related (on average) are sequences within a clone. We implement an average Phylogenetic diversity metric (avPD)[8] that is equal to the sum of the lengths of all those branches on the tree divided by the number of branches of this set.



The positions of abundant clonotypes in the evolutionary tree can be informative. In order to quantify these positions, we have used node depth (D) and node height (H) values. The depth of a node is the number of edges from the node to the tree's root node. A root node will have a depth of 0. The height of a node is the number of edges on the longest path from the node to a leaf. A leaf node will have a height of zero (Figure 3). We create a function that returns depth and height, for the three most abundant clonotypes in BCR lineage trees. If we detect an unobserved internal node that Clonaltree added and is absent in the original sequences file, we will add a note about it in the file with the results.

In addition, we calculate the number of levels in the tree lineage tree and return it as the SizeTree metric.

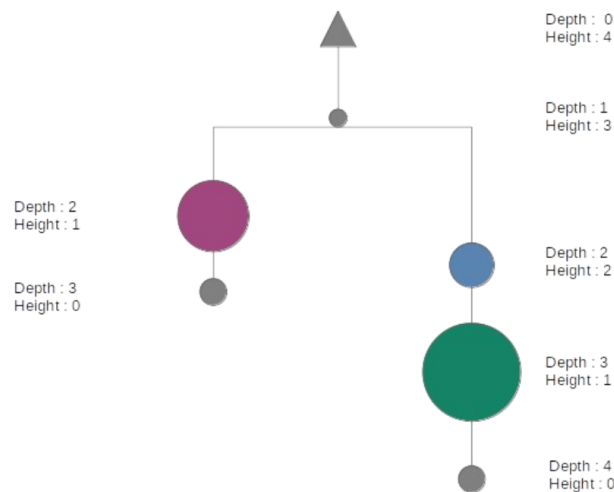
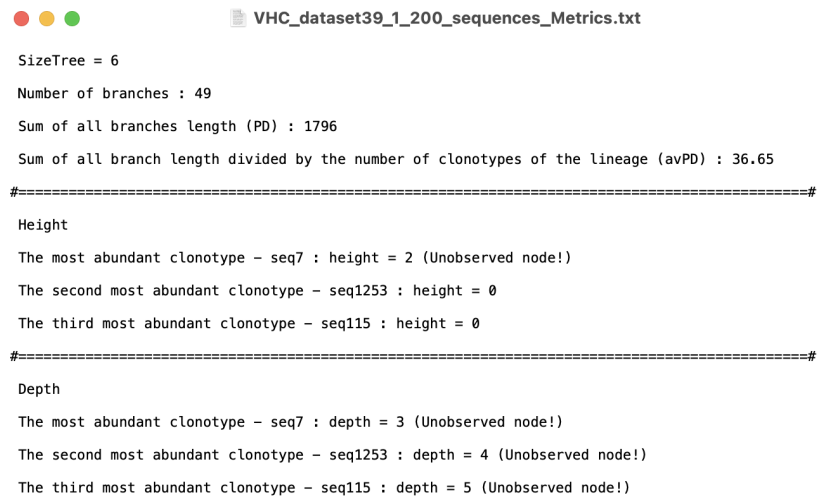


Figure 3: Example of depth and height values for edges in tree.

As a result, all the metrics (Table 4) calculated for a given repertoire are written to a text file and saved with the other files obtained by the ClonalTree algorithm (Figure 4).

Table 1: Metrics implemented in ClonalTree algorithm.

Number of branches	PD	avPD	H1	H2	H3	D1	D2	D3	SizeTree
Number of branches of the tree	Sum of all branches length	Sum of all branches length divided by the number of clonotypes of the lineage	Height for the most abundant clonotype	Height for the second most abundant clonotype	Height for the third most abundant clonotype	Depth for the most abundant clonotype	Depth for the second most abundant clonotype	Depth for the third most abundant clonotype	Depth of the tree



```

VHC_dataset39_1_200_sequences_Metrics.txt

SizeTree = 6
Number of branches : 49
Sum of all branches length (PD) : 1796
Sum of all branch length divided by the number of clonotypes of the lineage (avPD) : 36.65
=====#
Height
The most abundant clonotype - seq7 : height = 2 (Unobserved node!)
The second most abundant clonotype - seq1253 : height = 0
The third most abundant clonotype - seq115 : height = 0
=====#
Depth
The most abundant clonotype - seq7 : depth = 3 (Unobserved node!)
The second most abundant clonotype - seq1253 : depth = 4 (Unobserved node!)
The third most abundant clonotype - seq115 : depth = 5 (Unobserved node!)

```

Figure 4: **Example of a resulting text file that describes all metrics.**

## 2.3 Data

We used four datasets to analyse different BCR lineage trees and their behaviour related to our calculated metrics. The first dataset includes 92 BCR lineage trees that were simulated by GCTree.

The second dataset is composed of IgH repertoires of patients with chronic lymphocytic leukaemia. These repertoires revealed a very high intraclonal diversity on Vidjil (an immune repertoire analysis tool [10]). This dataset contains thirteen repertoires from different patients. These trees were previously analysed and classified into four categories during a descriptive study at the biological haematology laboratory of the Pitié-Salpêtrière Hospital. The complete datasets and their scores are shown in the supplementary information (Table S1). To better understand the meaning of the values in the topology of BCR lineage trees and show the differences between the categories, we visualise these trees in the Elbow and Circle formats. One example of a tree from each category is available in the supplementary information (Figures S1-S4).

The third dataset corresponds to uncategorised repertoires that describe 184 IgH repertoires of patients with chronic lymphocytic leukaemia. The fourth dataset is composed of 40 IgH repertoires of patients with Waldenstrom macroglobulinemia. For the three datasets containing patient repertoires, we constructed lineage trees only for the most abundant clone.

## 3 Results

### 3.1 Analysis of simulated BCR lineage trees

To evaluate ClonalTree performance, we compared trees obtained by ClonalTree with those obtained by GCTree. We possessed 92 repertoires that contained IgH sequences generated by GCTree and the resulting trees for these repertoires. The resulting trees served as the true trees that we hoped to reconstruct using ClonalTree. By using the lineages generated by GCTree, we reconstruct 92 new trees from ClonalTree. Next, we calculated all our metrics for each GCTree tree and ClonalTree tree. As a result, we obtained two sets of score values for each repertoire (184 sets of metrics in total) : one describing the tree by ClonalTree, the other describing the tree by GCTree.

In order to quantify the resemblance of reconstructed ClonalTree' trees from simulated truth trees by GCTree, we created a global score that describes the difference between two trees. This global score is based on Euclidean distance that calculates using each of ten metrics results in the pair:

$$Score = \sqrt{\sum_{i=1}^{10} (metric\_ClonalTree_i - metric\_GCTree_i)^2} \quad (1)$$

We observed that GCTree and ClonalTree reconstructed the tree with the same score in 57 % of cases, but for the remaining pairs, the global score was more than zero (Table 2). The maximum score was seven. Figure 5 shows boxplots of global scores for each pair of metrics values of the 92 BCR lineages trees.

Table 2: Scores for ClonalTree and GCTree BCR lineage trees

Value of score	Number of pairs obtained this scores (total = 92)	%
= 0	52	56.6
> 0	40	43.4
> 2	19	20.6
> 4	4	4.3

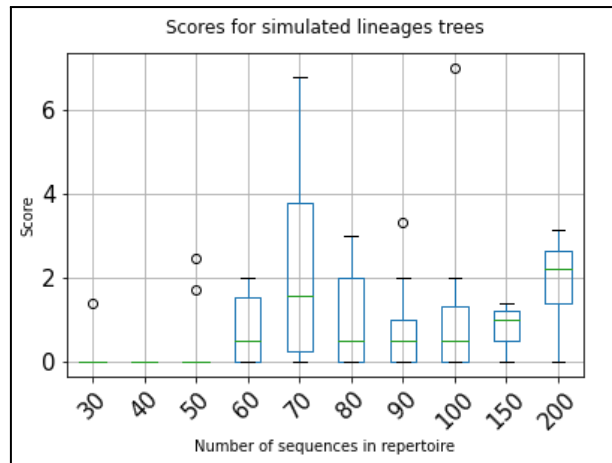


Figure 5: **Comparison of GCtree and ClonalTree by the score between their BCR lineage trees.**  
Box-plots represent the scores for trees with the same number of IgH sequences.

To determine why ClonalTree and GCtree sometimes build different trees, we decided to analyse the pairs of trees with scores greater than zero. We obtained forty cases when the score was more than zero. Figure 6 demonstrates in which metrics the dissimilarities occurred and how many times. The most divergent metrics were PD with mean difference  $2.00 \pm 1.44$  and avPD with mean difference  $0.05 \pm 0.04$ . *SizeTree* was in third place with mean difference  $1.11 \pm 0.33$ . According to these metrics differences, we found that ClonalTree produces a tree with higher number of levels (*SizeTree* metric, on average, one more level), but with lower sum and average value of lengths of all branches of the tree (PD and avPD metrics).

We assume that the differences in these three metrics (PD, avPD and *SizeTree*) are due to the step of the ClonalTree algorithm to reconstruct the tree in order to reduce the number of levels in the tree. ClonalTree might reduce the number of levels of the tree, but this algorithm does not accept this change in order to keep the smallest sum of all branches, which is an indicator of the amount of evolutionary change.

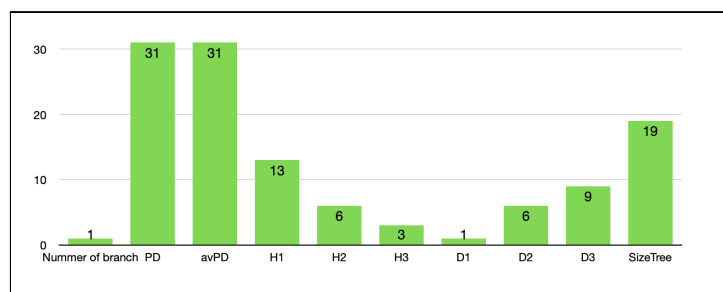


Figure 6: **Number of times the metrics were different.**  
Consider 40 pairs of BCR lineage trees with a score greater than zero.

### 3.2 Analysis of categorised BCR lineage trees

The second dataset that was analysed is composed of IgH repertoires of patients with chronic lymphocytic leukaemia. This dataset contains thirteen repertoires from different patients. For each of them, we reconstructed BCR lineage trees just for the most abundant clone. These trees were previously classified into four categories during a descriptive study at the biological haematology laboratory of the Pitié-Salpêtrière Hospital (Table 3).

Table 3: **Description of 4 categories of trees.**

<b>Category 1</b>	Almost all minority clonotypes are directly descended from the majority clonotype, a tree is shaped like a comb.
<b>Category 2</b>	A very early formation of a tree in the evolutionary history, which gives rise to the two main clonotypes, each responsible for a comb of descendants
<b>Category 3</b>	A secondary clonotype with significant numbers appears very early in evolutionary history and gives rise to the main clonotype.
<b>Category 4</b>	Trees too complex to belong to the previous categories.

We decided to examine whether our metrics could be used to determine the category of the tree. Firstly, we used the Principal Component Algorithm [11] (PCA) that takes as input all metrics values and generates a collection of vectors, where each vector  $v(i)$  tries to fit the data best while being orthogonal to vector  $v(i-1)$ . Each vector is called a principal component of the data. The first generated vector is the one that best fits with data, the second one is the second best, and so on. We calculated the first two principal components using all metrics values. Then, we generated a new table where we added the PCA result for each BCR lineage tree. Figure 7 represents that trees from Category 1 and from Category 2 seem to be located in specific regions. However, trees from Category 3 and Category 4 seem to be mixed. Then we calculate the explained variance ratio for our principal components (an array of the variance of the data explained by each of the principal components). It is known that if the sum of explained variance ratio for 2 principal components is around 0.8, these components explain over 80% of the variance in the original data and it is enough to classify data. For our first and second principal components (PC1 and PC2) the sum of explained variance ratio was 0.62 which is not enough.

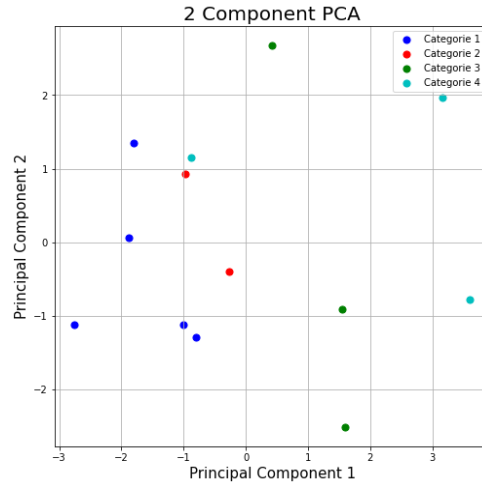


Figure 7: **Results of PCA for Categoricalised dataset.**

In order to try to recreate four qualitative categories, we decided to use the  $k$ -nearest neighbours [11] (KNN) algorithm that uses datasets to generate classes based on the number of asked classes and the similarity of the data points. This algorithm is able to classify all the points according to the similarity of previously defined classes. We set as input the data points coming from the output of PCA for each tree and using KNN algorithm, we generated four classes. We tried two different approaches to initialise our initial values for classes. In the first approach, we set all in zero, and in the second one we initialised using the average of PC1 and PC2 for each class. After initialising, we used the KNN algorithm with  $\alpha = 0.5$ . We stopped the algorithm when the difference between the centre in stage  $n$  and  $n-1$  was smaller than 0.001. Then, we classified the dataset using the new centres of each class. Centres initialised with zero were saved with name “KNN\_zero” while the centres initialised with the average were saved with the name “KNN\_moyenne”. The results of both of them are shown in Table 4.

We calculated sensitivity and specificity for results coming from KNN classification to compare them and determine if they are accurate enough to classify BCR lineage trees in our predefined categories from Table 4. In Table 5 we show the results for KNN, and in Table 6, the results for KNN\_moyenne. We observe that we got better results by initialising using average values for each class. Nevertheless, none of the combinations between PCA and KNN allow us to properly classify the clonal trees as we are getting very low sensitivity values. These results come from the fact that we are not able to differentiate classes after the PCA output presented in Figure 7.

Table 4: PCA and KNN results from Dataset1.

	PC1	PC2	Categorie	KNN_zero	KNN_moy
0	1.663964	1.345121	1	1.0	1.0
1	1.687469	0.047713	1	1.0	1.0
2	1.087055	-1.146615	1	1.0	1.0
3	1.013657	-1.320312	1	1.0	1.0
4	3.158271	-1.201718	1	1.0	1.0
5	-0.184434	-0.365643	2	4.0	2.0
6	0.526174	0.953448	2	3.0	2.0
7	-1.009050	2.730705	3	3.0	4.0
8	-1.845943	-2.473893	3	2.0	3.0
9	-2.193221	-0.839032	3	2.0	3.0
10	-1.764859	1.899486	4	3.0	4.0
11	0.779805	1.155897	4	3.0	2.0
12	-2.918889	-0.785157	4	2.0	3.0

Table 5: Sensitivity and Specificity for KNN\_zero using Dataset 1.

	True P	True N	False P	False N	Sensitivity	Specificity
1	5.0	8.0	0.0	0.0	1.000000	1.000000
2	0.0	8.0	3.0	2.0	0.000000	0.727273
3	1.0	7.0	3.0	2.0	0.333333	0.700000
4	0.0	9.0	1.0	3.0	0.000000	0.900000

Table 6: Sensitivity and Specificity for KNN\_moyenne using Dataset 1.

	True P	True N	False P	False N	Sensitivity	Specificity
1	5.0	8.0	0.0	0.0	1.000000	1.000000
2	2.0	10.0	1.0	0.0	1.000000	0.909091
3	2.0	9.0	1.0	1.0	0.666667	0.900000
4	1.0	9.0	1.0	2.0	0.333333	0.900000

### 3.3 Analysis of uncategorised BCR lineage trees

The datasets of IgH repertoires of patients with chronic lymphocytic leukaemia (LLC+MUT dataset) and IgH repertoires of patients with Waldenström macroglobulinemia (VHC dataset) were analysed during the final phase of the project. The VHC dataset contained 40 IgH repertoires, the LLC + MUT dataset - 184 repertoires. We calculated the metrics for the lineage tree of the most abundant clones in each repertoire.

The obtained metrics show that the BCR lineage trees are much more extensive and diverse than BCR lineage trees built for simulated data (Table 7). During processing data from these IgH repertoires we have found that some IgH lineage trees might have less than four IgH sequences including germline. Due to this we have changed our functions that calculated *Height* and *Depth* metrics, and they work even in cases where there are not enough sequences to find three most abundant IgH sequences.

Table 7: Mean values for all metrics for BCR lineage trees from LLC+MUT and VHC.

LLC + MUT dataset										
	Number of branch	PD	avPD	H1	H2	H3	D1	D2	D3	SizeTree
Mean ± sd	57 ± 28	1861 ± 980	33.3 ± 7.2	2.5 ± 1.2	1.3 ± 1.8	0.5 ± 1.1	2.5 ± 1.3	3.3 ± 1.6	3.8 ± 1.6	6.1 ± 1.8

VHC dataset										
	Number of branch	PD	avPD	H1	H2	H3	D1	D2	D3	SizeTree
Mean ± sd	51 ± 33	1676 ± 1178	34.0 ± 6.6	2.2 ± 1.1	0.5 ± 0.9	0.2 ± 0.9	2.2 ± 0.8	3.7 ± 1.3	3.8 ± 1.3	5.5 ± 1.6

During processing metrics data, we have noticed that for the VHC dataset, the second most abundant IgH sequence in 70% of the cases is a tree leaf (metric H2 is zero), while the third most abundant IgH sequence in 90% of the cases is a tree leaf (metric H3 is zero). Next, we decided that it might be important to determine the positions of the most abundant IgH sequence in the tree: in 25% of cases, it is one level below the root, in 25% of cases, it is placed two levels below the root, and in 50% of cases, this IgH sequence is positioned at three levels below the root with the average number of tree levels equal to six (Figure 8). We decided that the knowledge that the second and third most abundant IgH sequences do not have descendants and belong to the newest generation in the BCR lineage might be relevant for clinical purposes.



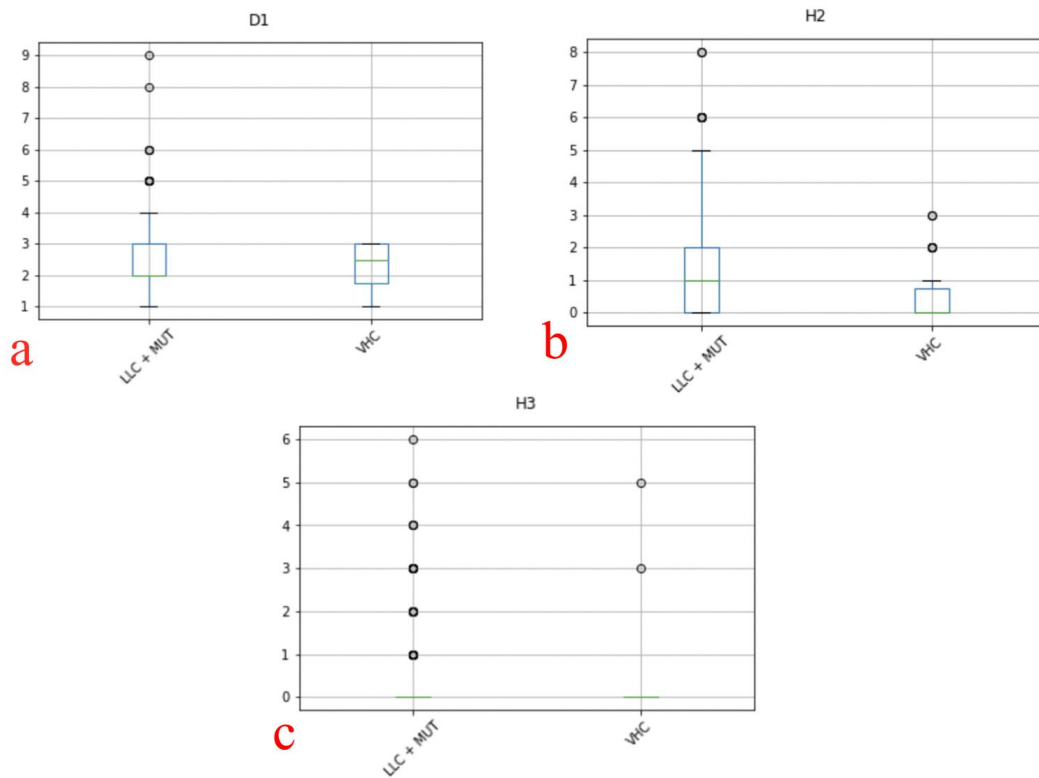


Figure 8 : **Positions of the three most abundant IgH sequences.**

- a** - D1 metric for the most abundant IgH sequences,
- b** - H2 metric for the second most abundant IgH sequences,
- c** - H3 metric for the third most abundant IgH sequences.

## 4 Discussion

In this project, we have presented several ways of applying phylogenetic metrics to characterise and describe BCR lineage trees, as well as an analysis of the results obtained using these metrics.

Firstly, we compared the performance of the ClonalTree and GCtree algorithms. We determine that for the same IgH clone these algorithms will reconstruct identical trees in approximately 60% of the cases. However, sometimes the ClonalTree algorithm will reconstruct BCR lineage trees with a larger number of levels in order to conserve even the smallest amount of evolutionary changes.

Then, we showed that by selecting different calculated metrics, we could generate and assign to the trees new scores. This proves to be a useful instrument to compare the trees as it is possible to establish quantitatively differences between two BCR lineage trees. In the long term, the comparison of the two trees may be applied to examine the mutations on a lineage coming from one same patient over a period of time and establish a relation between them and the statement of their illness.

Moreover, developed phylogenetic metrics might provide meaningful information about IgH lineage trees of patients with certain haematological disorders. We have determined that for IgH major clone of patients with Waldenstrom macroglobulinemia, the most abundant IgH cell is in general one or two levels below the germline, while the next two most abundant IgH cells will be represented as leaves of the BCR lineage tree and therefore they do not have descendants.

However, our set of parameters are not sufficient to recreate the qualitatively established categories defined previously during a descriptive study at the biological haematology laboratory of the Pitié-Salpêtrière Hospital. This showed that it is necessary to calculate more scores to better describe the properties of the BCR lineage trees.

In conclusion, our work on BCR lineage trees was the beginning of the application of phylogenetic metrics for studying lineage evolution in IgH sequences during their affinity maturation process and the clonal expansion. This study has the potential to explore the evolution of B-cells further.

## 5 Bibliography

- [1] Brigitte Gubler “Chapitre 10 : Les lymphocytes B : diversité et ontogenèse.” In Frédéric Batteux, Olivier Garraud, Yves Renaudineau, Laurent Vallat “Les Immunologie fondamentale et immunopathologie.” 2018, pp. 70-77.
- [2] Gur Yaari, Jason Vander Heiden, Mohamed Uduman, Daniel Gadala- Maria, Namita Gupta, Joel NH Stern, Kevin O’Connor, David Hafler, Uri Laserson, Francois Vigneault, et al. “Models of somatic hypermutation targeting and substitution based on synonymous mutations from high-throughput immunoglobulin sequencing data.” In: *Frontiers in immunology* 4 (2013), p. 358.
- [3] Uri Hershberg and Eline T Luning Prak. “The analysis of clonal expansions in normal and autoimmune B cell repertoires.” In: *Philosophical Transactions of the Royal Society B: Biological Sciences* 370.1676 (2015), p. 20140239.
- [4] Abdollahi, N. B cell receptor repertoire analysis in clinical context: new approaches for clonal grouping, intra-clonal diversity studies, and repertoire visualisation. *Immunology. Sorbonne Université*. 2021.
- [5] William S DeWitt, III, Luka Mesin, Gabriel D Victora, Vladimir N Minin, Frederick A Matsen, IV, Using Genotype Abundance to Improve Phylogenetic Inference, *Molecular Biology and Evolution*, Volume 35, Issue 5, May 2018, Pages 1253–1265, <https://doi.org/10.1093/molbev/msy020>
- [6] O. Grygorash, Y. Zhou and Z. Jorgensen, "Minimum Spanning Tree Based Clustering Algorithms," *2006 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'06)*, 2006, pp. 73-81, doi: 10.1109/ICTAI.2006.83.
- [7] R. C. Prim, "Shortest connection networks and some generalisations," in *The Bell System Technical Journal*, vol. 36, no. 6, pp. 1389-1401, Nov. 1957, doi: 10.1002/j.1538-7305.1957.tb01515.x.
- [8] Tucker, Caroline M. et al. “A Guide to Phylogenetic Metrics for Conservation, Community Ecology and Macroecology: A Guide to Phylogenetic Metrics for Ecology.” *Biological reviews of the Cambridge Philosophical Society* 92.2 (2017): 698–715. Web.
- [9] Faith, DANIEL P. "Phylogenetic diversity and conservation." *Conservation biology: Evolution in action* (2008): 99-115.
- [10] Duez M, Giraud M, Herbert R, Rocher T, Salson M, Thonier F (2016) Vidjil: A Web Platform for Analysis of High-Throughput Repertoire Sequencing. *PLoS ONE* 11(11): e0166126. Pmid:27835690
- [11] M. O. Arowolo, M. Adebisi, A. Adebisi and O. Okesola, "PCA Model For RNA-Seq Malaria Vector Data Classification Using KNN And Decision Tree Algorithm," *2020 International Conference in Mathematics, Computer Engineering and Computer Science (ICMCECS)*, 2020, pp. 1-8, doi: 10.1109/ICMCECS47690.2020.240881. (PCA, KNN)

## 6 Appendices

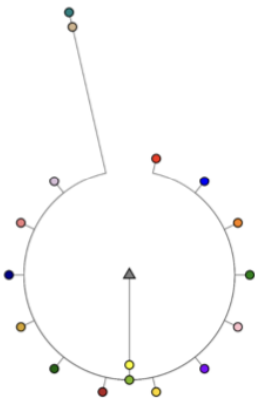
### 6.1 Categorised dataset

Table S1: Metrics for categorised dataset.

	dataset1_1	dataset2_1	dataset3_1	dataset4_1	dataset5_1	dataset6_1	dataset7_1	dataset8_1	dataset9_1	dataset10_1	dataset11_2	dataset12_1	dataset13_1
Categorie	1	1	1	1	1	2	2	3	3	3	4	4	4
Number of branches	16	17	17	16	17	18	17	22	19	17	28	18	22
(PD) - Sum of all branches length	39	31	53	63	46	38	33	34	74	49	170	47	139
(avPD) - Sum of all branch length divided by the number of clonotypes of the lineage	2.44	1.82	3.12	3.94	2.71	2.11	1.94	1.55	3.89	2.88	06.07	2.61	6.32
H1 - height of the most abundant	2	2	2	2	4	1	1	1	1	1	1	1	1
H2 - height of the second most abundant	1	1	1	1	1	2	1	0	5	4	1	1	3
H3 - height of the third most abundant	1	0	0	0	0	0	0	3	0	2	1	0	1
D1 - depth of the most abundant	1	2	3	3	2	3	2	3	3	5	3	2	5
D2 - depth of the second most abundant	2	3	4	4	5	2	1	1	2	2	2	2	2
D3 - depth of the third most abundant	2	3	4	4	3	4	3	3	6	4	3	2	4
SizeTree	4	5	6	6	7	5	4	7	8	7	6	4	7

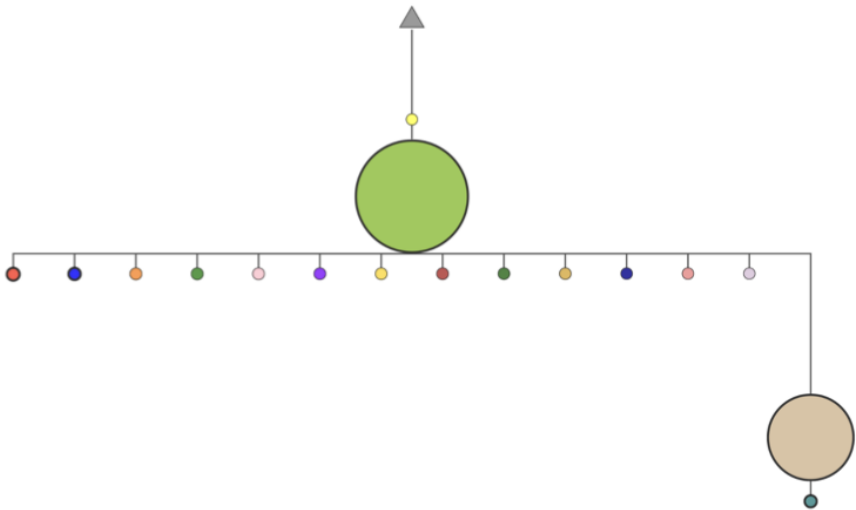
6.2 Visualisation of trees of different categories

	Catégorie	Number of branches	(PD) - Sum of all branches length	(avPD) - Sum of all branch length divided by the number of clonotypes of the lineage
dataset2_1_simplifie	1	17	31	1,82



dataset2\_1\_simplifie Circle tree

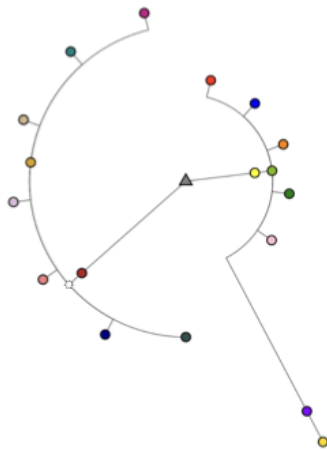
	H1	H2	H3	D1	D2	D3
dataset2_1_simplifie	2	1	0	2	3	3



dataset2\_1 simplifie Elbow tree

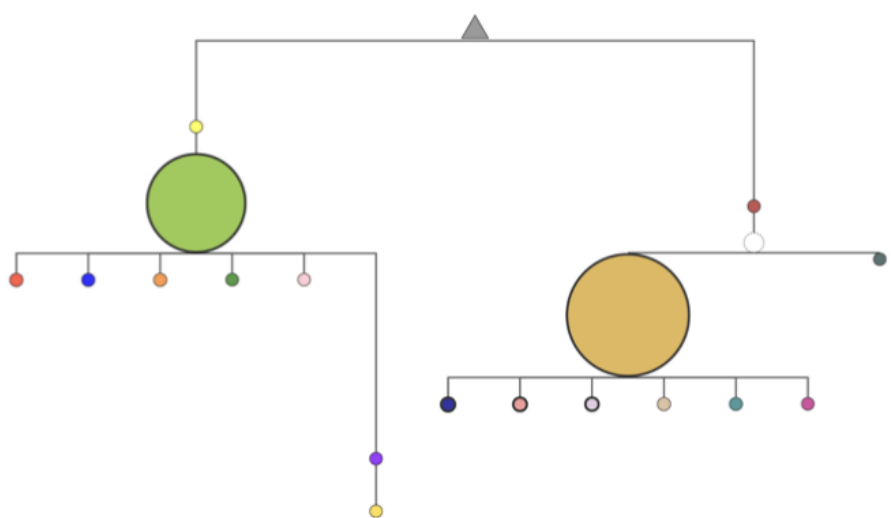
Figure S1 : **Visualisation of first category tree.** Almost all minority clonotypes are directly descended from the majority clonotype, a tree is shaped like a comb.

	Catégorie	Number of branches	(PD) - Sum of all branches length	(avPD) - Sum of all branch length divided by the number of clonotypes of the lineage
dataset6_1_simplifie	2	18	38	2,11



dataset6\_1\_simplifie Circle tree

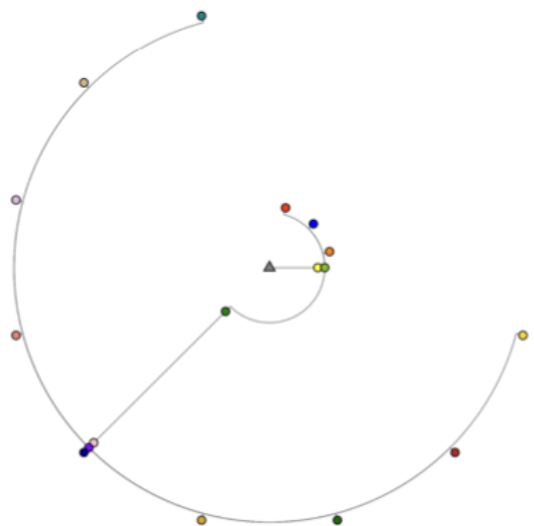
	H1	H2	H3	D1	D2	D3
dataset6_1_simplifie	1	2	0	3	2	4



dataset6\_1 simplifie Elbow tree

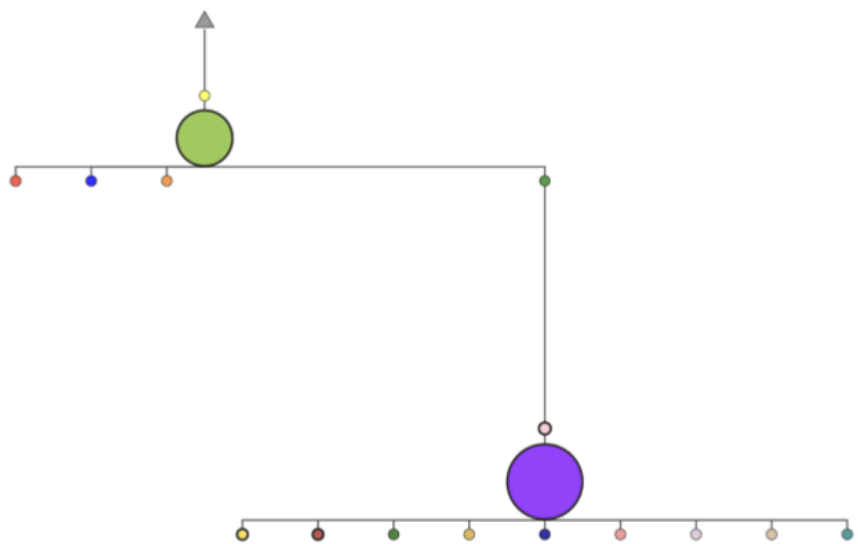
Figure S2 : **Visualisation of second category tree.** An early formation of a tree in evolutionary history, which gives rise to the two main clonotypes.

	Catégorie	Number of branches	(PD) - Sum of all branches length	(avPD) - Sum of all branch length divided by the number of clonotypes of the lineage
dataset10_1_simplifie	3	17	49	2,88



dataset10\_1\_simplifie Circle tree

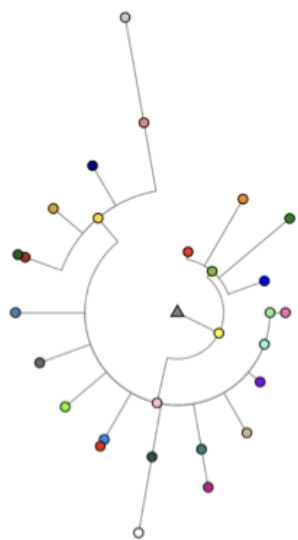
	H1	H2	H3	D1	D2	D3
dataset10_1_simplifie	1	4	2	5	2	4



dataset10\_1\_simplifie Elbow tree

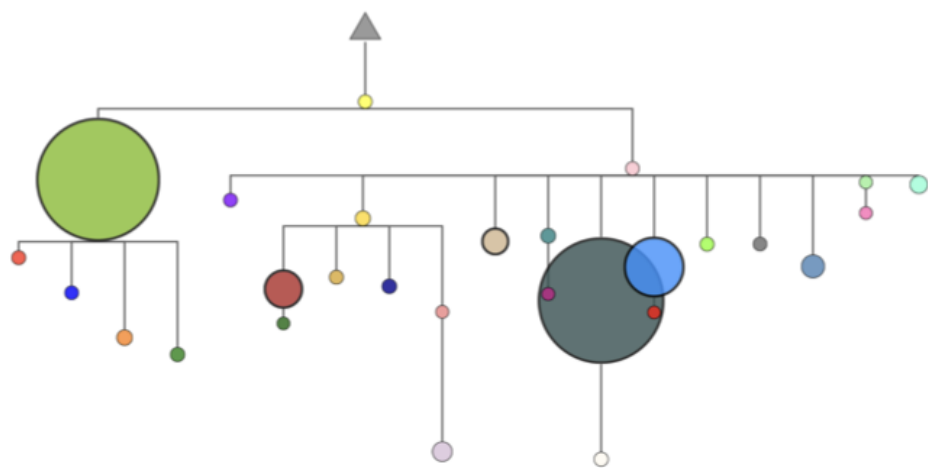
Figure S3 : **Visualisation of third category tree.** A secondary clonotype with significant numbers appears very early in evolutionary history and gives rise to the main clonotype.

	Catégorie	Number of branches	(PD) - Sum of all branches length	(avPD) - Sum of all branch length divided by the number of clonotypes of the lineage
dataset11_2_simplifie	4	28	170	6,07



dataset11\_2\_simplifie Circle tree

	H1	H2	H3	D1	D2	D3
dataset11_2_simplifie	1	1	1	3	2	2



dataset11\_2\_simplifie Elbow tree

Figure S4 : **Visualisation of fourth category tree.**  
Tree too complex to belong to the previous categories.



## 6.3 Code

### Metrics.py

```
import numpy as np
import random
import sys
from Bio import SeqIO
import re
from ete3 import Tree
import heapq, operator
import csv

#=====
# function to read Fasta file
# input - fasta file
# output - array of sequences names and array of sequences
#=====

def readFasta_for_metrics(fastaFile):
    labels = []
    arraySeqs = []
    with open (fastaFile, 'r') as file:
        for line in file:
            if (line[0] == '>'):
                name = line[1:].replace ('\n', '')
                name = re.sub('@[0-9]*', '', name)
                labels.append(name)
            else:
                arraySeqs.append(line.replace('\n', ''))
    return labels, arraySeqs

#=====
# function to calculate Hamming distance
# input - two sequences
# output - Hamming distance
#=====

def hamming_distance_for_metrics(seq1, seq2):
    return sum(c1 != c2 for c1, c2 in zip(seq1, seq2))

#=====
# function to calculate:
# The number of branches
# Sum of all branch lengths (PD)
# Sum of all branch lengths divided by the number of clonotypes of
the lineage (avPD)
# input1 - ete3 tree
```

```

# input2 - array of sequences names
# input3 - array of sequences
# output - number of branches, PD, avPD
#=====

def metricsPD(tree, labels, arraySeqs):
    num_of_branches = 0
    PD = 0
    avPD = 0.

    for parent in tree.traverse("preorder"):
        children = parent.children
        for child in children:
            if child.name != 'naive':
                if child.name == '':
                    remember_parent = parent
                else:
                    if parent.name == '':
                        parent_name = remember_parent.name
                    else:
                        parent_name = parent.name
                    child_name = child.name
                    index_p = labels.index(parent_name)
                    index_c = labels.index(child_name)
                    ham =
hamming_distance_for_metrics(arraySeqs[index_p],
arraySeqs[index_c])
                        num_of_branches +=1
                        PD += ham
                        #print (parent_name, child_name, ham)
        avPD = PD / num_of_branches
        avPD = round(avPD , 2)
    return num_of_branches, PD, avPD

#=====
# function to choose 3 most abundant sequences
# input - fasta file
# output - array of names for 3 most abundant sequences
#=====

def get_3_most_abundance (file) :
    abundance = {}
    #abundance["total"] = 0
    with open(file) as f:
        for line in f :
            if(line[0] == ">"):
                seq = line.split("\n")[0]

```

```

        seq_info = seq.split("@")
        seq_name = seq_info[0][1:]
        if(len(seq_info) == 2) :
            number_seq = int(seq_info[1])
            abundance[seq_name] = number_seq
            #abundance["total"] += number_seq
        else :
            if seq_name in abundance :
                abundance[seq_name] += 1
                #abundance["total"] += 1
            else :
                abundance[seq_name] = 1
                #abundance["total"] += 1

        abud3 = list(zip(*heapq.nlargest(3, abundance.items(),
key=operator.itemgetter(1))))[0]
        return abud3

#=====
# function that return:
# The path from nodeName to root
# input1 - nodeName : name of first node of path,
# input2 - ete3 tree
# output - list of nodes in the path
#=====

def pathToRoot(nodeName, tree):
    D = tree&nodeName
    # Get the path from B to the root
    node = D
    path = []
    while node.up:
        nn = node.name
        if node.name == '' or node.name.isdigit():
            nn = 'none'
        path.append(nn)
        node = node.up
    if path and path[len(path)-1] == 'none':
        path[len(path)-1] = 'naive'
    return path

#=====
# function that return:
# The path from nodeName to nodeFin
# input - nodeName : name of 1 node of path, nodeFin : name of
last node in path (won't print in path), tree
# output - list of nodes in the path
#=====

```

```

def pathToNode(nodeName, nodeFin, tree):
    D = tree[nodeName]
    node = D
    path = []
    while node.up and node.name != nodeFin:
        nn = node.name
        if node.name == '' or node.name.isdigit():
            nn = 'none'
        path.append(nn)
        node = node.up
    return path

#=====
# function that return:
# The height of a node : the number of edges on the longest path
# from the node to a leaf. A leaf node will have a height of 0.
# input - name of node that height we want to calculate, tree
# output1 - height
# output2 - flag = 1 if we had unobserved node on path
#=====

def height(nodeName, tree):
    leafs = []

    for node in tree.traverse("preorder"):
        if node.is_leaf() == True:
            leafs.append(node.name)

    path_depth = []
    if nodeName in leafs:
        h = 0
    else:
        h = -1
        for leaf in leafs:
            path = pathToNode(leaf, nodeName, tree)
            len_path = len(path)
            if len_path > h and ('naive' not in path):
                h = len_path
            path_depth = path # for print path and how we
received this height

    if 'none' in path_depth: #check for unobserved node
        flag = 1
    else:
        flag = 0

    return h, flag

```

```

#=====
# function that return:
# The depth of a node : the number of edges from the node to a
root.
# input - name of node that depth we want to calculate, tree
# output1 - depth
# output2 - flag = 1 if we had unobserved node on path
#=====

def depth(nodeName, tree):
    path_depth = pathToRoot(nodeName, tree)
    d = len(path_depth) - 1

    if 'none' in path_depth: #check for unobserved node
        flag = 1
    else:
        flag = 0

    return d, flag

def TreeSize(tree):

    leafs = []

    for node in tree.traverse("preorder"):
        if node.is_leaf() == True:
            leafs.append(node.name)

    h = -1
    for leaf in leafs:
        path = pathToRoot(leaf, tree)
        len_path = len(path)
        if len_path > h:
            h = len_path

    return h

#=====
# function that return:
# Array of depth and height
# input1 - fasta file
# input2 - tree
# outputs - height array, depth array, height flags, depth flags,
names of 3 most abundant sequences
#=====

def metricsDH(fasta, tree):

```

```

array3 = get_3_most_abundance(fasta)

d_list = []
d_list_flag = []
h_list = []
h_list_flag = []

for node in array3:

    d, flag_d = depth(node, tree)
    d_list.append(d)
    d_list_flag.append(flag_d)

    h, flag_h = height(node, tree)
    h_list.append(h)
    h_list_flag.append(flag_h)

return h_list, d_list, h_list_flag, d_list_flag, array3

#=====
# function that return:
# all metrics
# input1 - fasta file
# input2 - Newick file
# output - all metrics
#=====

def allMetrics(input_fasta, input_newick):
    labels1, arraySeqs1 = readFasta_for_metrics(input_fasta)
    with open (input_newick, 'r') as file:
        for line in file :
            t = Tree(line, format = 1)
            Num1, PD1, avPD1 = metricsPD(t, labels1, arraySeqs1)
            h1, d1, h1_flags, d1_flags, most_abund3 =
metricsDH(input_fasta, t)
            size = TreeSize(t)
            return Num1, PD1, avPD1, h1[0], h1[1], h1[2], d1[0], d1[1],
d1[2], size

#=====
# function that write all metrics in file
# input1 - fasta file
# input2 - tree
# input3 - output file
#=====

```

```

def all_metrics_write_in_file(input_fasta, tree, output_file):
    labels1, arraySeqs1 = readFasta_for_metrics(input_fasta)
    t = tree
    Num1, PD1, avPD1 = metricsPD(t, labels1, arraySeqs1)
    h1, d1, h1_flags, d1_flags, most_abund3 =
metricsDH(input_fasta, t)
    size = TreeSize(t)

    filename_txt = output_file.replace('.abRT.nk', '') +
'_Metrics.txt'
    f = open(filename_txt, 'w')
    f.writelines ('\n SizeTree = ' + str(size) + '\n')
    f.write('\n Number of branches : ' + str(Num1) + '\n')
    f.write('\n Sum of all branches length (PD) : ' + str(PD1) +
'\n')
    f.write('\n Sum of all branch length divided by the number of
clonotypes of the lineage (avPD) : ' + str(avPD1) + '\n')

    f.writelines
('\n#=====
=====#\n\n')
    f.writelines (' Height\n')

    if h1_flags[0] == 1:
        s_un = ' (Unobserved node!) \n'
    else:
        s_un = ' \n'

    f.writelines('\n The most abundant clonotype - ' +
most_abund3[0] + ' : height = ' + str(h1[0]) + s_un)

    if h1_flags[1] == 1:
        s_un = ' (Unobserved node!) \n'
    else:
        s_un = ' \n'

    f.writelines('\n The second most abundant clonotype - ' +
most_abund3[1] + ' : height = ' + str(h1[1]) + s_un)

    if h1_flags[2] == 1:
        s_un = ' (Unobserved node!) \n'
    else:
        s_un = ' \n'

    f.writelines('\n The third most abundant clonotype - ' +
most_abund3[2] + ' : height = ' + str(h1[2]) + s_un)

```

```

        f.writelines
('\\n#=====
=====#\\n\\n')
        f.writelines (' Depth\\n')

        if d1_flags[0] == 1:
            s_un = ' (Unobserved node!) \\n'
        else:
            s_un = ' \\n'

        f.writelines('\\n The most abundant clonotype - ' +
most_abund3[0] + ' : depth = ' + str(d1[0]) + s_un)

        if d1_flags[1] == 1:
            s_un = ' (Unobserved node!) \\n'
        else:
            s_un = ' \\n'

        f.writelines('\\n The second most abundant clonotype - ' +
most_abund3[1] + ' : depth = ' + str(d1[1]) + s_un)

        if d1_flags[2] == 1:
            s_un = ' (Unobserved node!) \\n'
        else:
            s_un = ' \\n'

        f.writelines('\\n The third most abundant clonotype - ' +
most_abund3[2] + ' : depth = ' + str(d1[2]) + s_un)

        f.close()

```

## ClonalTree\_GStree

```

import numpy as np
import random
import sys
import re
from ete3 import Tree
import csv
import pandas as pd
from scipy.spatial import distance
import matplotlib.pyplot as plt

pd.set_option('display.float_format', lambda x: '%.2f' % x)

```



```

# generate bash commands to launch clonalTree
def generate_command_for_bash(n, j):
    for i in range(1, n + 1):
        print ('python src/clonalTree.py -i Data/Simulations/' +
str(j) + '/' + str(j) + '_' + str(i) + '.fasta -o
Data/Simulations/' + str(j) + '/' + str(j) + '_' + str(i) +
'.clonalTree.nk -a 0 -r 0 -t 0')
nn = [50, 60, 70, 80, 90, 100, 200]
for number in nn:
    generate_command_for_bash(10, number)
print ('fin')

generate_command_for_bash(10, 150)

dict = {} # {name of sequence : [all metrics]}

#function for metrics of ClonalTree and GCTree sequences from 30_,
40_, 50_, 60_, 70_, 80_, 90_, 100_
def run2arbres(j):
    for i in range(1,11):
        fastaFile = 'Simulations/' + str(j) + '/' + str(j) + '_'
+ str(i) + '.fasta'
        nClonalTree = 'Simulations/' + str(j) + '/' + str(j) + '_'
+ str(i) + '.clonalTree.nk'
        nGT = 'Simulations/' + str(j) + '/' + str(j) + '_' +
str(i) + '.GT.naive.nk'
        if j < 100:
            dict[nClonalTree[15:-3]] = list(allMetrics(fastaFile,
nClonalTree))
            dict[nGT[15:-3]] = list(allMetrics(fastaFile, nGT))
        else:
            dict[nClonalTree[16:-3]] = list(allMetrics(fastaFile,
nClonalTree))
            dict[nGT[16:-3]] = list(allMetrics(fastaFile, nGT))

#function for metrics of ClonalTree and GCTree sequences from 200_
def run2arbres200(j):
    for i in range(1,11):
        if i != 2:
            fastaFile = 'Simulations/' + str(j) + '/' + str(j) +
 '_' + str(i) + '.fasta'
            nClonalTree = 'Simulations/' + str(j) + '/' + str(j) +
 '_' + str(i) + '.clonalTree.nk'
            nGT = 'Simulations/' + str(j) + '/' + str(j) + '_' +
str(i) + '.GT.naive.nk'
            dict[nClonalTree[16:-3]] = list(allMetrics(fastaFile,
nClonalTree))
            dict[nGT[16:-3]] = list(allMetrics(fastaFile, nGT))

```

```

#function for metrics of ClonalTree and GCTree sequences from 150_
def run2arbres150(j):
    for i in [1, 2, 10]:
        fastaFile = 'Simulations/' + str(j) + '/' + str(j) + '_' +
+ str(i) + '.fasta'
        nClonalTree = 'Simulations/' + str(j) + '/' + str(j) + '_' +
+ str(i) + '.clonalTree.nk'
        nGT = 'Simulations/' + str(j) + '/' + str(j) + '_' +
+ str(i) + '.GT.naive.nk'
        dict[nClonalTree[16:-3]] = list(allMetrics(fastaFile,
nClonalTree))
        dict[nGT[16:-3]] = list(allMetrics(fastaFile, nGT))

numbers = [30, 40, 50, 60, 70, 80, 90, 100]
for j in numbers:
    run2arbres(j)

run2arbres150 (150)
run2arbres200 (200)

# function for score (Euclidean distance)
def score(a, b):
    mean = (a + b) / 2
    sd = np.sum(np.sqrt(np.square(a - mean) + np.square(b -
mean)))
    sd = round (sd, 1)
    ed = np.sqrt(np.sum(np.square(a - b)))
    ed = round(ed, 3)

    met = []
    n = 0
    a_b = a - b
    metrics = ['Number of branches', 'PD', 'avPD', 'H1', 'H2',
'H3', 'D1', 'D2', 'D3', 'SizeTree']

    for i in range(len(metrics)):
        if a_b[i] != 0:
            met.append(metrics[i])

    return [sd, ed, met]

dict_euclidean = {} # {name of sequence : score}
dict_euclidean_m = {} # {name of sequence : array of absolute
value difference for all metrics}

```

```

dict_non_zero = {} # {name of sequence : score, [metrics with
difference]}

for key in dict.keys():
    if 'clonalTree' in key:
        a = np.asarray(dict[key])
        ab = key
        ab = re.sub('[a-z]*[A-Z][a-z]*', '', ab)
    if 'GT.naive' in key:
        b = np.asarray(dict[key])
        a_b = a - b
        if a_b.any() != 0:
            sc = score(a, b)
            dict_euclidean[ab] = sc
            #dict_euclidean[ab] = np.abs(a - b)
        else:
            dict_euclidean[ab] = [0, 0]
        if dict_euclidean[ab] != [0, 0]:
            dict_euclidean_m[ab] = np.absolute(a - b)
            dict_non_zero[ab] = dict_euclidean[ab]

dict_with_scores = {} # {name of sequence : score, groupe 30-100,
150, 200}
d_keys = []
d_score_sd = []
d_score_ed = []
d_groupe = []

for key in dict_euclidean.keys():
    if key[0:2] == '10':
        groupe = 100
    elif key[0:2] == '15':
        groupe = 150
    elif key[0:2] == '20':
        groupe = 200
    else :
        groupe = int (key[0:2])
    dict_with_scores[key] = [dict_euclidean[key][0],
dict_euclidean[key][1], groupe]
    d_keys.append(key)
    d_score_sd.append(dict_euclidean[key][0])
    d_score_ed.append (dict_euclidean[key][1])
    d_groupe.append(groupe)

```

```

# creating a Dataframe object from a list
df_scores = pd.DataFrame({'Name':d_keys , 'Score_sd':
d_score_sd,'Score_ed': d_score_ed, 'Groupe' : d_groupe})

# percentage of different scores
print ((df_scores['Score_ed'] > 6).sum(axis=0) /92 * 100)
print ((df_scores['Score_ed'] > 4).sum(axis=0) / 92 * 100)
print ((df_scores['Score_ed'] > 2).sum(axis=0) /92 * 100)
print ((df_scores['Score_ed'] > 0).sum(axis=0) / 92 * 100)
print ((df_scores['Score_ed'] == 0).sum(axis=0) / 92 * 100)

# box-plot for scores for different groups
boxplot = df_scores.boxplot(by = 'Groupe', column =['Score_ed'],
grid = True, rot=45, fontsize=15)
boxplot.get_figure().gca().set_title("")
boxplot.get_figure().suptitle('Scores for simulated lineages
trees')
boxplot.get_figure().gca().set_ylabel("Score")
boxplot.get_figure().gca().set_xlabel("Number of sequences in
repertoire")
boxplot.plot()

plt.show()

# write in csv difference between ClonalTree and GStree trees
for key in dict_euclidean_m.keys():
    s = [key]
    for elem in dict_euclidean_m[key]:
        s.append(elem)
    with open("2arbresdifferent.csv", "a") as p:
        pr = csv.writer(p, delimiter = ";", lineterminator = '\n')
        pr.writerow(s)

df2 = pd.read_csv('2arbresdifferent.csv', delimiter=';',
names=['Newick file', 'Number of branches', 'PD', 'avPD', 'H1',
'H2', 'H3', 'D1', 'D2', 'D3', 'SizeTree'])

# count how many times metrics were different
metrics = ['Number of branches', 'PD', 'avPD', 'H1', 'H2', 'H3',
'D1', 'D2', 'D3', 'SizeTree']
for metric in metrics:
    #print (df2[metric].value_counts())
    print ((df2[metric] != 0).sum(axis=0), '\n')

```

```

# statistics about metrics difference
df3 = df2
df3 = df3.replace(0, np.NaN)
df3.describe()

```

## **Metrics\_LL\_C\_MUT\_VHC.py**

```

import numpy as np
import random
import sys
from Bio import SeqIO
import re
from ete3 import Tree
import heapq, operator
import csv
import pandas as pd
from scipy.spatial import distance
import Metrics.py

# calculate metrics for VHC
dict_VHC = {}
for i in range (1, 10):
    file_fasta = 'VHC/VHC_dataset0' + str(i) +
'_1_200_sequences.aln.fa'
    file_newick = 'VHC/VHC_dataset0' +
str(i)+'_1_200_sequences.aln.fa.nk'
    dict_VHC[file_fasta[4:33]] = list(allMetrics(file_fasta,
file_newick))

for i in range (10, 41):
    file_fasta = 'VHC/VHC_dataset' + str(i) +
'_1_200_sequences.aln.fa'
    file_newick = 'VHC/VHC_dataset' +
str(i)+'_1_200_sequences.aln.fa.nk'
    dict_VHC[file_fasta[4:33]] = list(allMetrics(file_fasta,
file_newick))
    #print (i, allMetrics(file_fasta, file_newick))

# calculate metrics for MUT
MUT_fasta = []
with open ('MUTnames.txt', 'r') as file:
    for line in file:
        MUT_fasta.append(line.replace('\n', ''))

```

```

dict_MUT = {}
for seq in MUT_fasta:
    file_fasta = 'MUT/' + seq
    file_newick = 'MUT/' + seq + '.nk'
    dict_MUT[file_fasta[4:-7]] = list(allMetrics(file_fasta,
file_newick))

# calculate metrics for LLC
dict_LLC = {}
for i in range (1, 10):
    file_fasta = 'LLC/LLC_dataset0' + str(i) +
'_1_200_sequences.aln.fa'
    file_newick = 'LLC/LLC_dataset0' +
str(i)+'_1_200_sequences.aln.fa.nk'
    dict_LLC[file_fasta[4:33]] = list(allMetrics(file_fasta,
file_newick))

for i in range (10, 57):
    file_fasta = 'LLC/LLC_dataset' + str(i) +
'_1_200_sequences.aln.fa'
    file_newick = 'LLC/LLC_dataset' +
str(i)+'_1_200_sequences.aln.fa.nk'
    dict_LLC[file_fasta[4:33]] = list(allMetrics(file_fasta,
file_newick))

# create csv with metrics
def write_metrics_in_csv(our_dict, filename):
    for key in our_dict.keys():
        if filename == 'LLC' or filename == 'MUT':
            groupe = 'LLC + MUT'
        else:
            groupe = 'VHC'
        s = [key, groupe]
        for elem in our_dict[key]:
            s.append(elem)
        with open(groupe + '.csv', "a") as p:
            pr = csv.writer(p, delimiter = ";", lineterminator =
'\n')
            pr.writerow(s)

# create csv with metrics for PCA (categorie 1- LLC+MUT categorie
2 - VHC)
def write_metrics_in_csv_for_PCA(our_dict, filename):
    for key in our_dict.keys():
        if filename == 'LLC' or filename == 'MUT':
            name = 'LLC + MUT'

```

```

        groupe = 1
    else:
        name = 'VHC'
        groupe = 2
    s = [key, groupe]
    for elem in our_dict[key]:
        s.append(elem)
    with open(name + '.csv', "a") as p:
        pr = csv.writer(p, delimiter = ";", lineterminator =
'\n')
        pr.writerow(s)

write_metrics_in_csv(dict_LLC, 'LLC')
write_metrics_in_csv(dict_MUT, 'MUT')
write_metrics_in_csv(dict_VHC, 'VHC')

write_metrics_in_csv_for_PCA(dict_LLC, 'LLC')
write_metrics_in_csv_for_PCA(dict_MUT, 'MUT')
write_metrics_in_csv_for_PCA(dict_VHC, 'VHC')

# datasets LLC+MUT and VHC
df_LLC_MUT = pd.read_csv('LLC + MUT.csv', delimiter=';',
names=['Newick file', 'Dataset', 'Number of branches', 'PD',
'avPD', 'H1', 'H2', 'H3', 'D1', 'D2', 'D3', 'SizeTree'])
df_VHC = pd.read_csv('VHC.csv', delimiter=';', names=['Newick
file', 'Dataset', 'Number of branches', 'PD', 'avPD', 'H1', 'H2',
'H3', 'D1', 'D2', 'D3', 'SizeTree'])

# percentage of H1, H2, H3 == 0
def h_count_zero(metric):
    print ('LLC + MUT ', metric, ' : ', round ((df_LLC_MUT[metric]
== 0).sum(axis=0) / len(df_LLC_MUT), 1) * 100, '%\n')
    print ('VHC ', metric, ' : ', (df_VHC[metric] ==
0).sum(axis=0) / len(df_VHC) * 100, '%\n')

# percentage of D == i
def d_count(metric, i):
    print ('LLC + MUT ', metric, ' = ', i, ' : ', round
((df_LLC_MUT[metric] == i).sum(axis=0) / len(df_LLC_MUT), 1) *
100, '%\n')
    #print ('VHC ', metric, ' = ', i, ' : ', round
((df_VHC[metric] == i).sum(axis=0) / len(df_VHC), 1) * 100, '%\n')

metricsH = ['H1', 'H2', 'H3']

```

```

metricsD = [ 'D1', 'D2', 'D3']

for m in metricsH:
    h_count(m)

for m in metricsD:
    for i in range(1, 6):
        d_count(m, i)

pd.set_option('display.float_format', lambda x: '%.1f' % x)
df_VHC.describe()
df_LLC_MUT.describe()

df_all = df_LLC_MUT.append(df_VHC, ignore_index=True)

# boxplots for metrics
def made_boxplot(metric):
    boxplot = df_all.boxplot(by = 'Dataset', column =[metric],
grid = True, rot=45, fontsize=10)

    boxplot.get_figure().gca().set_title("")
    boxplot.get_figure().suptitle(metric)
    boxplot.plot()

    plt.show()

metrics = ['Number of branches', 'PD', 'avPD', 'H1', 'H2', 'H3',
'D1', 'D2', 'D3', 'SizeTree']
for metric in metrics:
    made_boxplot(metric)

```