University of Liège
School of Engineering and Computer Science

Machine learning for a fast sparse triangular solve

Master's thesis completed in order to obtain the degree of Master of Science in Software Engineering

DI RAIMO Gaël

Supervisor: Prof. Louveaux
Academic Year 2022-2023

Abstract

When solving simplex problems, multiple systems of linear equations are solved. This systems are decomposed in their *LU* forms, where L and U are sparse. This triangular systems can be solved with different algorithms that uses the sparse structure to save time. The context in which an algorithm is faster than another is not well known, 3 of these algorithms referred as *General algorithm*, *Two-Phase algorithm* and *One-Phase algorithm* can have very different computation times to solve the systems. One could think to use Machine Learning techniques to learn in which context an algorithm is faster than others. However, the time window between the two fastest algorithms can be narrow. Therefore, fast models should be used to predict the fastest algorithm. One parameter which is expected to have a lot of information is the number of non-zero elements in the solution, a model will be tested to see if knowing this value can lead to better average computation time. A simple model classifying these algorithms is created using only information on b, the right-hand side. This model will be used to compared its performances with new models which are created. The model that is proposed uses 4 regressions to predict the algorithm, one is used to predict the number of non-zero elements in the solution which is used as feature for the 3 other regressions. The other regressions predict the time needed by each algorithm and the classifier finally predict the one with the lowest predicted time. **RFAdaProxy (b)** is the classifier proposed. This model is able to solve the test set proposed in $34.78s$ whereas the simple model with just b information needs $44.85s$ and the fictional model which would always predict the best algorithm obtained a computation time of $19.95s$.

Acknowledgement

I would like to thank Professor Louveaux who guided me throughout this work and gave me helpful advice. I am also thankful to Professor Louveaux for the proofreading of some part of this work.

I would also like to thank Laurent Poirrier who gave me the dataset necessary for this master's thesis, without this data, this work would not be possible.

Contents

1 Introduction

When solving simplex[1] problems, a series of system of linear equations is solved. Each time, one changes from one vertex to another, a system of linear equations is solved. To find the solution, a LU decomposition that can be obtained using *Forrest Tomlin*[2] and *Suhl-Suhl* [3] is made. L and U being sparse triangular matrices, to solve these type of systems, some methods uses the advantage of sparse matrix format to find the solution as fast as possible as in *Direct methods for sparse linear systems*[4]. One algorithm taking advantage of the sparse matrix format is referred in this master's thesis as *General algorithm*, another one is called *Two-Phase algorithm* and the last one that will be used is the *One-Phase algorithm*. Each of these algorithms can be better than the others depending on the context. However, knowing in which context the algorithm is the fastest is not well understood. Indeed, the time taken by each algorithm depends on the non-zeros layout of L, U and b. An hypothesis is that the computation time of these algorithms depends on the number of non-zero elements in the solution. Nonetheless, knowing these number of non-zeros is impossible before solving the systems. An approximation of this number could be used to choose the faster algorithm, which in ideal cases lead to saving time. Machine learning algorithms can be used to obtain this approximation. Nonetheless, the time that can be saved while choosing the faster algorithm instead of another one can be very low. Therefore, fast machine learning techniques must be used in order to make the approximation to avoid adding an unnecessary computation time to the system solve. Thus, deep learning methods with multiple layers could lead to, actually, waste time. Consequently, These methods were not explored in this work.

This work starts by explaining the necessary concepts used in the creation of the classifier. The first concepts explained are the 3 algorithms implementation that are classified. Understanding these algorithms is necessary to know how features can be extracted in order to later better classify them. The data structures necessary to the solve of sparse triangular systems are also explained for the same reasons. Machine learning algorithm basic concepts will also be explained such as supervised learning, classification and regression. Some of the models used will also be explained. Furthermore, different metrics that can be used to evaluate the performances of regression and classification models are also described.

Some experiments will be done in order to support the claim that the number of non-zero elements in the solution is informative with respect to the computation times of the 3 algorithms. The relation between the number of non-zero of b will also be tested because this information can be accessed very fast. In that experiment random L matrices and b vectors are generated and the system $Lx = b$ is solved. Then, the computation time is plotted as a function of the number of non-zero elements of the solution, supporting the claim that this number is important for some algorithm. However, the size of b seem more important for the algorithm referred as *General algorithm.* The dataset used will be described and how less important data is removed to win some time on the computation of the features and on the solve of all the LU systems of the dataset.

A first simple model is created to be able to compared future models to the performances of this very simple one. This model is composed of 3 regression models, one per solve algorithm. Each regression tries to approximate the computation time of its assigned algorithm. These regressions are, actually, linear regressions that uses the information of b to approximate its algorithm computation time. The information used to predict the *General algorithm* computation time is the size of b. The two other algorithm times are predicted using the number of non-zero elements in b. The classification model first predicts the 3 algorithm computation times, then, it returns the algorithm which has the lowest predicted time.

Similarly to the first simple model, another model is created to show that with the number of non-zero elements in the solution another simple model can perform much better than the one based only on b. It also support the previous claim that the number of non-zero element of the solution is a key information to know which algorithm is the fastest.

Because the number of non-zero elements of the solution can be used to better classify the algorithm and since this value is not know before solving the system, regression models will be trained in order to predict this value. The available information being not enough, new features are engineered in order to predict the number of non-zero elements of x. Each of these feature have a computation time complexity in the order of the size of b because higher complexity could lead to a too much time overhead to predict the fastest algorithm. Then, these feature will be trained with different regression models. For the prediction of these non-zeros elements the best average $MAPE$ obtained one the test folds is around 0.5 which does not seem great. From these regression models, 4 are selected in order to be used as a proxy predictor in the next step of the algorithm classifier.

With these predicted values, regression models are trained to predict the time taken by each algorithm. Each regression model are trained with the 4 models predicting the proxy. As with the previous simple model, one regression is trained to predict the computation time of its assigned algorithm and the algorithm with the lowest time predicted is the final algorithm chosen for the solve. In that part, the regression of the time does not seem to be well predicted, however, the choice of algorithm with these regression model does not seem bad. The models that predict the least worst performances are selected and the average computation time on the test folds is computed. Then, the model with the lowest average time on the test folds is selected. Finally, this selected model is used on 20% of dataset that was not previously seen to assess the performance of the model. The final model chosen is referred as **RFAdaproxy (b)**, the first part is "RF" because the model uses *Random Forest* as algorithm time regressor. The second part "AdaProxy" is because the proxy is predicted using *AdaBoost*. The last part "(b)" is for the fact that the *General algorithm* time is actually predicted with a linear regression using only the size of b as feature. When tested on the test set, the model that would always predict the fastest algorithm obtained a computation time of $19.95s$, the one using linear regression and only b information obtained a computation time of $44.85s$ and the model **RFAdaproxy (b)** obtained a computation time of $34.78s$. This result is obtained when adding the feature and the prediction computation time. This results show that the model is able to better predict the fastest algorithm in average compared to the simple model.

2 Background

2.1 Solving systems of linear equations

To solve a system of linear equations which is represented by $Ax = b$, where A is a $n \times n$ matrix, b is a known vector of size $n \times 1$ and x is the solution which is a vector of size $n \times 1$, one well-known method is to compute the LU decomposition of the matrix A, where L is a lower triangular matrix and U is an upper triangular matrix. Therefore, the matrix equation becomes

$$LUx = b.$$

In order to find the unknown vector x with such decomposition, one must solve two triangular systems. The first step being to compute y using (1) to finally compute the solution x by injecting y in (2).

$$Ly = b \tag{1}$$

$$Ux = y \tag{2}$$

Solving (1) can be done by using *forward substitution.* This algorithm computes y_i in increasing order of i by using the element of the matrix $L = [l_{ij}]$ and by supposing that $l_{ii} \neq 0 \, \forall i \in$

$\{1, 2, ..., n\}$. This algorithm computes each y_i using

$$y_i = \frac{b_i - \sum_{k=1}^{i-1} l_{ik} y_k}{l_{ii}}. \tag{3}$$

After the computation of y_i, the solution x can be computed using, this time, the upper-triangular matrix $U = [u_{ij}]$ and supposing that $u_{ii} \neq 0 \; \forall i \in \{1, 2, ..., n\}$. The algorithm to solve linear system with upper-triangular is called *backward substitution* because the algorithm does not computes the solution x_i in increasing but in decreasing order of i, this means that the values of x are computed in the order $x_n, x_{n-1}, ..., x_1$, and each of these value is computed using

$$x_i = \frac{y_i - \sum_{k=i+1}^{n} u_{i,k} x_k}{u_{ii}}. \tag{4}$$

Another way tries to avoid computing this sum naively in order to solve the lower triangular system $Lx = b$. This method at each time a value of x_k is computed will compute one more term of the sum from (3) for each x_i s.t $i > k$. This is how Algorithm 1 finds x. Moreover, one can see in line 8 of the Algorithm 1 that it is not necessary to update x_i if the element l_{ij} is equal to 0 because it has no impact on the sum. Therefore, in cases where L matrices have many zero elements, some unnecessary computation can be avoided. One only needs to consider the non-zero elements and can use a data structure that saves only non-zero elements.

There exists multiple data structures to save matrices which have a high percentage of zero elements. These matrices are called sparse matrices. One simple way to avoid saving the zero elements of a matrix would be to use the *coordinate* format of the matrix. This representation uses 3 arrays, one that stores the row indices, another stores the column indices and the last one stores the values of each element. Let say that these array are called *row*, *col* and *val* respectively and that n_z is the number of non-zeros of a matrix L, then the *coordinate format* of the matrix L respects the equality

$$L[row[i], col[i]] = val[i] \; \forall i \in \{0, 1, ..., n_z - 1\}.$$

One can also see in the loop starting at line 7 of Algorithm 1, that the algorithm only accesses the elements of the column j of the matrix L that are non zero. However, accessing the element in a column of sparse matrices in *coordinate* format is not optimal. Indeed, to find the correct element l_{ij}, the algorithm will need to travel the *row* and *col* array both at the same time and find an index k such that if $row[k] = i$ and $col[k] = j$, therefore, $val[k] = l_{ij}$. The worst scenario would be that the value of the element accessed is the last element in the array *val*, thus, the complexity is $O(n_z)$, because n_z is the size of the the arrays *row* and *col*.

Algorithm 1 solve algorithm

```
1: function SOLVE(L,b)
2:     $x \leftarrow b$
3:
4:     for $j \in \{0, ..., n-1\}$ do
5:         $x_j \leftarrow \frac{x_j}{l_{jj}}$
6:
7:         for $i \in \{j+1, ..., n-1\}$ do
8:             $x_i \leftarrow x_i - l_{ij} x_j$
9:
10: return $x$
```

2.1.1 Compressed sparse structure

One way to have a faster access to a column j is to not use column indices in an array but to save pointers that indicate the start and the end of columns in the *row* and *val* array. This is how the *column sparse* format works. Therefore, 3 arrays are needed as in the coordinate format. One could call these arrays p, i and x. The array x stores the values of the elements, one column after the other and in increasing order. The array p stores the indices of the beginning of each columns in x. In some variation of the format, the array p has one more element than number of columns of the matrix. It is actually to store at the end of the array p the number of non-zeros elements of the matrix. The values stored in the second array i are the indices of the row of each element. The last array x stores the value of each element. For example, the lower triangular matrix shown in (5)

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 2 & 0 & 1 & 0 \\ 6 & 0 & 5 & 1 \end{pmatrix} \tag{5}$$

has a sparse representation with $L.nz = 8$, the number of non-zero element and these 3 arrays

$$L.p = [0, 4, 5, 7], \tag{6}$$
$$L.i = [0, 1, 2, 3, 1, 2, 3, 3] \text{ and} \tag{7}$$
$$L.x = [1, 2, 2, 6, 1, 1, 5, 1]. \tag{8}$$

The *compressed sparse column* format is better than *coordinate* format for accessing an entire column of a matrix. Indeed, for a matrix L, all elements of a column j can be accessed directly using the array with pointers $L.p$. One first finds the beginning index of the column in the array i and x by using $L.p[j]$ and finds the end by using $L.p[j+1]$, then, one can travel the indices and values of L with k, $\forall k \in \{L.p[j], L.p[j+1]-1\}$ by accessing $L.i[k]$ and $L.x[k]$ respectively. Accessing an entire column has a complexity that is equal to the number of non-zero elements of the column j, which is $O(L.p[j+1]-1-L.p[j])$. This complexity is much faster for a column access compared to *coordinate* format which is O(nz).

However, there are drawbacks when using *compressed sparse column* representation instead of using a 2 dimensional array. One of these drawbacks is the random access time of an element at a given row i and a column j. Indeed, to access an element A_{ij} of a matrix A represented with a sparse structure, first one must get the index of the start and end of the column j using $A.p$ which counts for two reads, then, it is unavoidable to go through a linear search in the part of the column j that is in the array $A.i$ in order to find the index where the value is stored in $A.x$. Therefore, random access in the worst case needs to travel the row index of the non-zero elements of an entire column, which leads to a complexity of $O(A.p[j+1]-1-A.p[j])$ for column j.

Another operation that is computationally expensive compared to the 2D array representation is to access elements of a same row, for example, an algorithm which would travel the columns of a matrix A of size $n \times n$ at a given row i and computes the sum

$$\sum_{j=1}^{n} A_{ij}, \tag{9}$$

this sum is computationally expensive because the only way to implement such an algorithm is to do a random access to each element A_{ij} which has complexity of $O(A.p[j+1]-1-A.p[j])$ resulting to a complexity of $O\big(n \times (A.p[j+1]-1-A.p[j])\big)$ for the sum, whereas the complexity for this algorithm when using a 2D array representation for A would simply be $O(n)$.

In some cases, it is interesting to have a fast access to row and not an access to column. Therefore, a data structure corresponding to *compressed sparse column* but for row access also exists. This structure is called *compressed sparse row* and the first array that can be called p will store pointers to the beginning and end of rows instead of the columns. The second array can be called j instead of i because it stores the indices of the columns and not the row indices. The last array x still stores the value of each element.

2.1.2 Sparse matrix solve algorithm

As explained in the previous section, Algorithm 1 can be solved faster if the second loop of the algorithm is only done over non-zero elements of the matrix L. Moreover, because this update actually needs to read columns of L, the *compressed sparse* structure for the matrix L is exactly what is needed. The next section is about the description of the general algorithm, which is similar to Algorithm 1 but with the use of a *column compressed sparse* structure for the matrix L.

2.1.3 General algorithm

The general algorithm to solve a triangular system takes as input the right-hand side b, an array of size n and a *compressed sparse column* triangular matrix L, which we suppose to be only lower triangular for the moment.

The first operation of the general algorithm is to initialise a vector x of size n with the same values as b and this vector x will at the end be the solution of the system. The structure used to represent x is an array of size n, therefore, at least n operations are needed to initialise to the value of x to the array b.

Once the vector x is initialised to b, the algorithm will loop over the row indices of x, with j which will take the value 0, 1, ..., $n-1$. In this loop, the general algorithm will compute $x[j]$ similarly to the line 5 of Algorithm 1 which is $x_j = \frac{x_j}{l_{jj}}$. However, because the matrix L is in *Compressed Sparse Column* format, the element l_{jj} is not accessed easily. Indeed, to find l_{jj} the algorithm must access the column j of the matrix L. To find the part of the column j in the array $L.i$ and $L.x$, one must can access $L.p[j]$ which indicate the starting index and $L.p[j+1]-1$ which is the last index. Therefore, to find the value l_{ij}, one must loop over $k \in \{L.p[j], ..., L.p[j+1]-1\}$ and find k such that $L.i[k] = j$ and if this statement is verified, $L.x[k] = l_{ij}$. The value l_{ij} being known, the algorithm can update $x[j]$ with $x[j] = \frac{x[j]}{L.x[k]}$.

The value x_j being updated, the general algorithm can make the operation corresponding to the second loop of Algorithm 1. This loop consist to update all the elements of x that depends on the previously computed x_j, these elements can be access as explained for the computation of the element x_j, i.e. with a loop over $k \in \{L.p[j], ..., L.p[j+1]-1\}$. The elements of x are updated with

$$x[L.i[k]] = x[L.i[k]] - x[j] \times L.x[k].$$

However, one must pay attention to not update twice the element x_j previously computed, thus, the update will be performed only if $L.i[k] \neq j$.

For clarity, the pseudo code of the algorithm is showed in Algorithm 2.

2.1.4 Two-phase algorithm

In some cases, the general algorithm performs unnecessary operations. Indeed, Algorithm 2 will initialise x to an array of size n to b, this initialisation have thus a complexity of $O(n)$ which could be very expensive in some cases. Moreover, the general algorithm travels each value of x, at each time, without considering if it is zero or not. These operations will not impact the results because if $x[j]$ equals zero then the line 6 of Algorithm 2 is the equivalent to store zeros in $x[j]$, therefore,

Algorithm 2 General algorithm

```
1:  function GeneralSolve(L,b)
2:      $x = b$
3:
4:      for $j \in \{0, ..., n\}$ do
5:
6:          for $k \in \{L.p[j], ..., L.p[j+1]-1\}$ do
7:
8:              if $L.i[k] = j$ then
9:                  $x[j] \leftarrow \frac{x[j]}{L.x[k]}$
10:
11:         for $k \in \{L.p[j], ..., L.p[j+1]-1\}$ do
12:
13:             if $L.i[k] \neq j$ then
14:                 $x[L.i[k]] \leftarrow x[L.i[k]] - x[j] \times L.x[k]$
15:
16: return $x$
```

these can be skipped. Moreover, the loop at line 8 will just set $x[k]$ to $x[k]$ because $x[j] = 0$, which is also not necessary. If the solution x is stored in a *compressed sparse column* format, the algorithm can avoid these unnecessary operations. This is what two-phase algorithm try to avoid doing and to do so the algorithm will have the same inputs as the general algorithm except that the format of the right-hand side b is in *compressed sparse column* format.

The solution chosen in the *Two-phase algorithm* to avoid the initialisation of x which cost $O(n)$ is to compute its number of non-zero elements before solving the system. This is performed by computing all the indices which could have a non-zero elements in the final solution of x. Determining the row indices of the non zero elements of the solution is the first phase of the two phases algorithm. Once the first phase is finished, the array i used in the *compressed sparse column* structure of x is known. Afterwards, in the second phase, the algorithm needs to travel these indices to compute the final solution, which is to compute the remaining unknown part of x, the array $x.x$.

In the first phase, one possibility to determine the indices of the non-zero elements of the solution is to first travel all indices of the elements of b with a value different than zero, this is easily done by accessing the array $b.i$. Each of these indices is added to a list l. In order to retrieve faster that value corresponding to that index, the element in the list can be a structure able to store two elements, the first one being an index and the second one a value corresponding to that index.

After adding all the elements of $b.i$ to the list, the list l is sorted in increasing order of the index. Then, the lowest index of the list l is read in order to find the row number of the non-zeros elements in the column j of L and add them to the list l. This actually corresponds to add all the elements of $L.i[k]\ \forall k \in \{L.p[j], ..., L.p[j+1]\}$, each index are added in the list with a value of 0 as second element. The next step, is to sort again the list l so that one can find the next index to explore. To find the next index j, the task performed is to find in l the lowest index which is greater than the previous element read from the list. These operations are performed until one cannot find any new index in the list greater than the previously read index.

In the second phase, the algorithm travels the sorted list l to compute the values of x. This corresponds to the same loop at line 4 of Algorithm 2 excepts that the algorithm loops over $x.i$ previously computed and that the structure of x is not a simple array anymore. An example of pseudo code for the two-phase algorithm is shown in Algorithm 1.

Algorithm 3 2 Phase algorithm: Phase 1

```
function 2PhaseSolve(L,b)
    $l \leftarrow list()$
    $sorted\_b \leftarrow SortByIndex(b)$

    for $j \in \{0, ..., sorted\_b.nz\}$ do
        $l.add(sorted\_b.i[j], sorted\_b.x[j])$

    $list\_index \leftarrow 0$
    $last\_explored\_row \leftarrow -1$

    while $length(l) > list\_index$ do

        if $last\_explored\_row = l[list\_index]$ then
            $list\_index \leftarrow list\_index + 1$
        else
            $len\_array \leftarrow L.p[l[list\_index] + 1] - L.p[l[list\_index]]$
            $new\_indices \leftarrow array(len\_array)$

            ▷ add all row indices at column $list\_index$ of L
            for $j \in \{L.p[l[list\_index]], ..., L.p[l[list\_index] + 1] - 1\}$ do
                $new\_indices[j - L.p[l[list\_index]]] = L.i[j]$

            $Sort(new\_indices)$

            ▷ Find where to insert in the list the new indices
            for $list\_index\_2 \in \{last\_explored\_row, ..., len(l) - 1\}$ do

                for $new\_index \in new\_indices$ do
                    if $new\_index = l[list\_index\_2].index$ then
                        break

                    if $new\_index < l[list\_index\_2].index$ then
                        ▷ Push the index $new\_index$ with a value of 0 at index $list\_index\_2$ of l
                        $l.push\_at(list\_index\_2, new\_index, 0)$

            $last\_explored\_row \leftarrow l[list\_index]$

    $x = CompressedSparseColumn(length(l))$
```

Algorithm 3 2 Phase algorithm: Phase 2

```
    $prev\_index \leftarrow -1$
    $x.nz \leftarrow 0$

    for $list\_index \in \{0, ..., length(l)\}$ do

        $new\_index \leftarrow l[list\_index].index$
        $x.i[x.nz] \leftarrow new\_index$

        ▷ Find diagonal value at $new\_index$ of $L$
        for $k \in \{L.p[new\_index], ..., L.p[new\_index + 1] - 1\}$ do
            if $L.i[k] = new\_index$ then
                $x.x[x.nz] \leftarrow \frac{l[list\_index].val}{L.x[k]}$

        for $k \in \{L.p[new\_index], ..., L.p[new\_index + 1] - 1\}$ do

            ▷ Travel the queue to update the corresponding element
            for $list\_index\_2 \in \{list\_index, ..., length(l)\}$ do
                if $L.i[k] = l[list\_index\_2].index$ then
                    $l[list\_index2].val \leftarrow l[list\_index\_2].val - x.x[x.nz] \times L.x[k]$

        $x.nz \leftarrow x.nz + 1$

    return $x$
```

2.1.5 One-phase algorithm

There is another way to avoid traveling all the zeros of the right-hand side b. The two-phase algorithm computes all the row indices of non-zero elements of the solution to initialise the exact size of x in *Compressed Sparse Column* format. Then, it can proceed to the actual computation of the solution. However, it is possible to do both at the same time. Such an algorithm, has only one phase.

The one-phase algorithm principle have some similarity with the two-phase algorithm. The two-phase algorithm has a sort that happens when row indices are added to a list. This sort is done such that in the second phase, the algorithm know the order of the computation of the element x_i. In order to know the next row index of x to compute, the one-phase algorithm will use a min priority queue adding the row indices as keys. Therefore, the one-phase algorithm just have to pop the element of the top of the queue to know the row index of the next x_i to compute.

An example of implementation of a one-phase algorithm could be to implement a min priority queue with a min heap. The first step of the algorithm is to initialise the min priority queue using b, with the row indices as key and to attach to that key the value at that corresponding row in b. Once the priority queue initialisation is finished, the algorithm can start the computation of x. The first non-zeros of x can be computed by taking the first element of the queue and its key and data. The key k which represent a row index is used to find the corresponding element in $L_{k,k}$. With the data d attached to the key k, the algorithm can compute x_k with

$$x_k = \frac{d}{L_{k,k}}. \tag{10}$$

Once the first value is computed, the algorithm travels the elements in the column k of L to add to the queue each row indices j of the non-zero elements of that column and by making sure to avoid adding the index k which is just computed. These row indices will be the key and the data attached to that key is computed with

$$data = -x_k \times L_{j,k}$$

The next non-zero elements of x are computed a bit differently. Now in the queue multiple elements may have the same key, this key may be added from the initialization of the queue or at a computation of a previous x_i. Therefore, when an element with key k is popped from the queue, the algorithm will see the top of the priority queue and pop again if the top key is still equal to k and compute d with

$$d = \sum_{\forall data \text{ s.t } key=k} data.$$

Then, the algorithm can compute x_k with (10). Afterwards, the algorithm can as previously explained add all row indices of the non-zero elements of the column k of L and compute the next x values as described in previous paragraph, the algorithm will keep computing a new value of x until the priority is empty.

The corresponding pseudo code for that example implementation is shown in Algorithm 4.

Algorithm 4 1-Phase algorithm

```
function 1PhaseSolve(L,b)
    $minQueue \leftarrow MinPriorityQueue()$

    for $j \in \{0, ..., b.nz - 1\}$ do
        $minQueue.add(b.i[j], b.x[j])$
    $x = CompressedSparseColumn(0)$
    $node \leftarrow minQueue.pop()$

    while not $minQueue.isEmpty()$ do

        $prev\_index = node.index$
        $x.i[x.nz] \leftarrow node.index$
        $x.x[x.nz] \leftarrow 0$

        while $prev\_index = node.index$ do
            $x.x[x.nz] \leftarrow x.x[x.nz] + node.val$
            $node \leftarrow minQueue.pop()$

        for $k \in \{L.p[prev\_index], ..., L.p[prev\_index + 1] - 1\}$ do
            if $L.i[k] = prev\_index$ then
                $x.x[x.nz] \leftarrow \frac{x.x[x.nz]}{L.x[k]}$
                break

        for $k \in \{L.p[prev\_index], ..., L.p[prev\_index + 1] - 1\}$ do
            if $L.i[k] \neq prev\_index$ then
                $minQueue.add(L.i[k], -L.x[k] \times x.x[x.nz])$
        $x.nz \leftarrow x.nz + 1$
    return $x$
```

2.2 Machine learning

2.2.1 Supervised Learning

As explain in **Introduction of Machine Learning** given at the **University of Liège** [5] the problem of Supervised Learning is to find a function that given data is able to better fit an output.

The formal definition of Supervised Learning given in the **Introduction of Machine Learning** given at the **University of Liège** [5] is :

"From a learning sample $\{(x_i, y_i)|i = 1, ..., N\}$ with $x_i \in \mathcal{X}$ and $y_i \in \mathcal{Y}$, find a function $f : \mathcal{X} \to \mathcal{Y}$ that minimize the expectation of some loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ over the joint distribution of input/output pairs: $E_{\mathcal{X},\mathcal{Y}}\{\ell(f(x, y))\}$"

The different Machine Learning model will have their own way to find this function given the data. There exist multiple lost function, one must find the loss function which is better suited for the encountered problems.

In these problems, there is a distinction when the predicted output y is discrete with a finite number of possibilities. In that case, it is a classification problem. One can distinguish the binary classification and multiclass classifications. Indeed, some metrics in binary classification does not work in the multiclass cases. However, when the output y can take continuous value, it is a regression problem and these problems have also different metrics to evaluate models.

2.3 Metrics for classification

There are multiple metrics to assess the performance of a classifier and they are suited to different type of problems. Metrics used in this work for classification are explained in the next pages. The different metric definitions are taken from **Scikit-Learn** user guide [6]. In the following definitions, n is the number of samples in the dataset.

The **accuracy** is one of the simplest metric, this metric just computes the number of well predicted samples over the total number of samples. Binary classification is when there is only two ouputs, positive or negative, for these classifications the **accuracy** corresponds to compute

$$\textbf{accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

where TP is the number of samples that are predicted positive and their real values are also positive. FP is the number of samples that are predicted positive but their true values are negative. TN is the number of samples which are well predicted negative, FN is the number of samples which are wrongly predicted negative.

However, this metric is not well suited for classification with an unbalanced class distribution. Indeed, if there is a binary classification problem which has a minority of the samples, for example 1%, then, a model that predicts the majority class all the time will have a 99% **accuracy** which seems great but it does not make a great job at predicting the low majority class which may be the initial goal of the classifier.

For classification with more than two classes the **accuracy** is computed differently, if y are the true value of the class and $\hat{y}$ are the predictions, then, the **accuracy** can be computed with

$$\textbf{accuracy}(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n} 1(\hat{y}_i = y_i).$$

The **balanced accuracy** is similar to the accuracy except that this metric takes into consideration the proportions of the classes in the dataset. The **balanced accuracy** can be computed with (11) for binary classification.

$$\textbf{balanced-accuracy} = \frac{1}{2}\Big(\frac{TP}{TP + FN} + \frac{TN}{TN + FP}\Big) \tag{11}$$

For classifier with more than 2 classes, the balanced accuracy is computed by first adjusting the weights of the samples with

$$\hat{w}_i = \frac{w_i}{\sum_{j=1}^{n} 1(y_j = y_i) w_j},$$

where y_i is the true class of sample i.

By default, the weight given to samples is 1. In that case, the weight $\hat{w}_i$ of the sample i is equal to the inverse of the number of times its class is represented in the dataset. If the prediction of the sample i is noted $\hat{y}_i$ then the computation of the **balanced accuracy** is

$$\textbf{balanced-accuracy}(y, \hat{y}, w) = \frac{1}{\sum_{i=1}^{n} \hat{w}_i} \sum_i 1(y_i = \hat{y}_i) \hat{w}_i \tag{12}$$

For instance, the **balanced accuracy** of a dataset of 100 samples with 3 classes and with a proportion for the majority class of 80% and 10% for the two other classes. If a classifier gives only the first class as prediction, then

$$\begin{aligned}\textbf{balanced accuracy} &= \frac{1}{80 \times \frac{1}{80} + 10 \times \frac{1}{10} + 10 \times \frac{1}{10}} \times \left(80 \times \frac{1}{80} + 0 + 0\right) \\ &= \frac{1}{3}.\end{aligned}$$

Whereas, the **accuracy** of this example is equal to 0.8. Actually, in a case where a classifier predicts only one class, the **balanced accuracy** is $\frac{1}{n\_class}$ where *n_class* is the number of classes.

When the dataset is balanced, each class is represented with an equal proportion, then, the **balanced accuracy** is equal to the **accuracy**.

2.4 Metrics for regression

Regression problems uses other metrics to evaluate a regression model. The one that are used in this work are the following metrics.

- One metric used for regressions is called **coefficient of determination**, which is also known as R^2. This score is computed with

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{n} (y_i - \bar{y}_i)^2} \tag{13}$$

  where n is the number of samples in the dataset and $\bar{y}_i = \sum_{i=1}^{n} \frac{y_i}{n}$. The R^2 score cannot be higher than 1. If $R^2 = 1$, this means that the predictions are equal to the true values. It is also possible to have a negative R^2. Indeed, if the predictions have an higher squared error than a model predicting the mean $\bar{y}$, $\sum_{i=1}^{n} (y_i - \hat{y}_i)^2 > \sum_{i=1}^{n} (y_i - \bar{y}_i)^2$ and, thus, the ratio in (13) is greater than 1 and R^2 is negative.

- Another metric that can be used is the **Mean Absolute Percentage Error**, which is also called **MAPE**. This value is computed with

$$MAPE(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} \frac{|y_i - \hat{y}_i|}{max(\epsilon, |y_i|)}. \tag{14}$$

  The value ϵ is an arbitrarily small value chosen to avoid dividing by 0 if $y = 0$.

2.5 Models

2.5.1 Decision Tree

The following formulas and examples in order to explain decision trees are taken from the course of **Introduction of Machine Learning** given at the **University of Liège** [5].
A decision tree for classification is a tree where each node within the tree represents a decision made on a feature. Each branch of the tree is actually a value of a feature and the last nodes of the tree called leaves are the final predicted classes. For example, one could have a tree to classify the decision to play tennis or not with Figure 1. In that example, the first choice is made on the feature *Outlook* which is the top node, *Outlook* can take 3 values *Sunny*, *Overcast* and *Rain* that are represented by the branches and if *Outlook* has the value *Overcast* then it leads to a leaf where the decision is to play tennis.

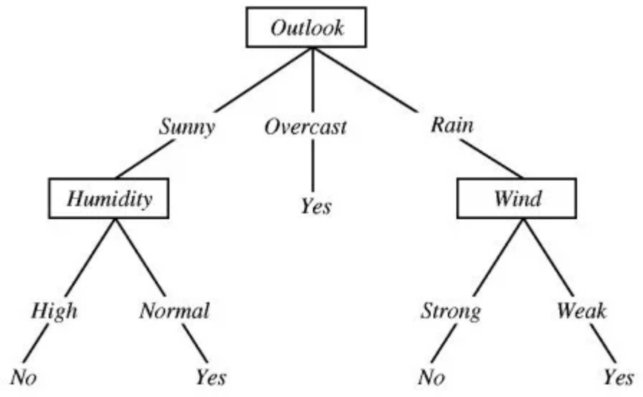


Figure 1: Decision tree to make the choice to play tennis or not. [5]

One way to construct such a tree is to use what is called the top-down approach. The decision tree is created by taking the learning samples and choosing an attribute according to a strategy to split the training set in smaller sets, one set for each possible value of the chosen attribute, this process actually creates new branches of the tree. Then, these smaller sets are afterward also divided using the same process, creating deeper branches of the tree. This division and creation of new branches continues until the final set only contains a class. This last set is thus a called leaf and contains samples with a unique class which will be the predicted class of an element which features corresponds to the path in the tree.

Actually, to choose the feature to divide a set, one can compute what is called an **impurity measure** that will evaluate how well the sample set is divided. Therefore, to choose which feature will be used in one node, one can computes for each features the expected impurity reduction and select the feature that maximize this expected **impurity measure**.

An **impurity measure** of a samples of objects LS, with the values p_j that represents the proportions of the objects with the class j in LS where $j \in \{0, 1, ..., J\}$. The **impurity measure** $I(LS)$ follows three properties:

- $I(LS)$ is minimum if the sample is alone, i.e. $\exists i \in LS$ s.t. $\forall j \in \{LS|j \neq i\}$ then $p_i = 1$ and $p_j = 0$;
- $I(LS)$ is maximum when the LS is divided in sets with equal number of samples. $\forall j, p_j = \frac{1}{J}$;
- $I(LS)$ is symmetric with the arguments p_j.

A function that respects the properties of **impurity measure** is the **Shannon Entropy** which is defines when using the previous notation as:

$$H(LS) = -\sum_{j=1}^{J} p_j log_2(p_j).$$

In the course, it is also explained how to finds the best feature to use in order to split the set into multiples sets. It is done choosing the feature that maximize the following **impurity measure**:

$$\Delta I(LS, A) = I(LS) - \sum_{a \in A(LS)} \frac{|LS_a|}{|LS|} I(LS_a), \tag{15}$$

where $A(LS)$ is the set of possible values taken by the feature A and LS_a is the subsets of LS where the features A have the value a.

Therefore, using these principles one tree can be built in order to make a classification model. However, this construction method does not explain how to build a tree for regression which is tailored to the prediction of non-zeros elements of the solution of a LU system.

To build a tree for regression, first, the leaf of the tree does not represent a class but the predicted value that is computed by averaging the prediction of the samples in that leaf. Moreover, the impurity measure is different, one that could be used is the variance of the output of the samples, which is

$$I(LS) = var_{y|LS}\{y\} = E_{y|LS}\Big\{\big(y - E_{y|LS}\{y\}\big)^2\Big\},$$

where $E_{y|LS}\{y\}$ is the average of the output y in the set LS.

The best split for regression are computed similarly to the split for classification three. They are chosen using (15) but with the impurity measure is replaced by an impurity measure for regression.

These process to create tree only cover features that are discrete and not continuous. In the case of continuous features, the features which are continuous are either pre-discretised before growing the tree or the feature can be discretised while the tree is being constructed. For example, one could discretize each continuous feature C before choosing the feature making the highest reduction of the impurity measure. In order to discretize C, one can create two sets of samples, one LS_{c^-} which is the set of all samples which have $C(o) \leq c$ and LS_{c^+} which is the set of all samples which have $C(o) > c$. Therefore, to discretize the set of the continuous feature C in two discrete variable, the algorithm can tests all the possibles splits for each values c of the feature C and pick the split that have the most impurity reduction. Thus, (16) can be computed for each feature $c \in C$ in order to find the value c where $\Delta I(LS, c)$ is maximum. Then, this discretized feature can be used in the splitting criteria of the actual tree.

$$\Delta I(LS, c) = I(LS) - \frac{|LS_{c^+}|}{|LS|} I(LS_{c^+}) - \frac{|LS_{c^-}|}{|LS|} I(LS), \tag{16}$$

An application of decision trees is *feature selection.* One can obtain a score representing how much a feature is important in the prediction of the decision tree. This score is obtained for a feature A by summing the impurity difference each times a particular feature A is used to make a split, which corresponds to compute

$$I(A) = \sum_{Node\ where\ A\ is\ tested} |LS_{node}| \Delta I(LS, A). \tag{17}$$

By computing the feature importance of each feature, one can also evaluate how much each feature gives information compared to other features and select for example the k best feature of the data.

3 Experiment for potential parameters

The goal of this section is not to give a proof of the influence that different parameters have on the time taken by an algorithm. The goal is rather to have an idea on how and which parameters can influence the computation time of the different algorithms.

In section 2.1, the *Two-Phase algorithm* 1 will initialize in the first phase a list with the non-zero elements of b and, then, it travels this list in order to find the indices of the non-zero elements of the solution. The *One-Phase algorithm* 4 initializes a priority queue with the non-zeros elements of b. Therefore, the two algorithm have a loop over the non-zero elements of the right-hand side, thus, this number can be a parameter for the computation time taken by methods.

To illustrate the influence of the number of non-zero elements in b on the computation time, the three algorithms were used to solve systems with the same matrix L. This matrix L is generated with a percentage of number of non-zero elements of 0.1% and with a size of $10^4 \times 10^4$, which is common in the dataset used. The number of non-zero elements given to a column of L is generated with a *triangular probability distribution* in order to have a uniform distribution in the matrix. This means that each column has a probability to have a number of non-zero elements inversely proportional to its column number. For this matrix L, multiple b are generated with different percentages of non-zero elements from 0.01% to 5%. A *triangular probability distribution* is also used to generate b so that there is more chance to have a non-zero element in the highest row number than in the first ones. This probability distribution is used in order to have a more uniform distribution for the number of non-zero elements in x which is another parameter which will be tested in this section. Indeed, if this distribution is not used, there is a high chance that the final solution will be dense.

One can see in Figure 2 that the time taken by the *Two-Phase algorithm* increases more than the two other algorithms when the number of non-zero elements in b increases. Figure 3 shows the same results but without the *Two-Phase algorithm* in order to better see the difference of performance between the *General* and *One-Phase algorithm.* In Figure 3, one can see that for b with a low number of non-zero elements, the *One-Phase algorithm* is better than the *General algorithm.* However, the time taken for *One-Phase algorithm* to solve systems increases with the number of non-zero elements in the right-hand side, whereas, the *General algorithm* seem to remain constant. Moreover, the standard deviation of the *One-Phase* algorithm seem to have a high variance which could lead to confusion when deciding which algorithm is better for a given b.

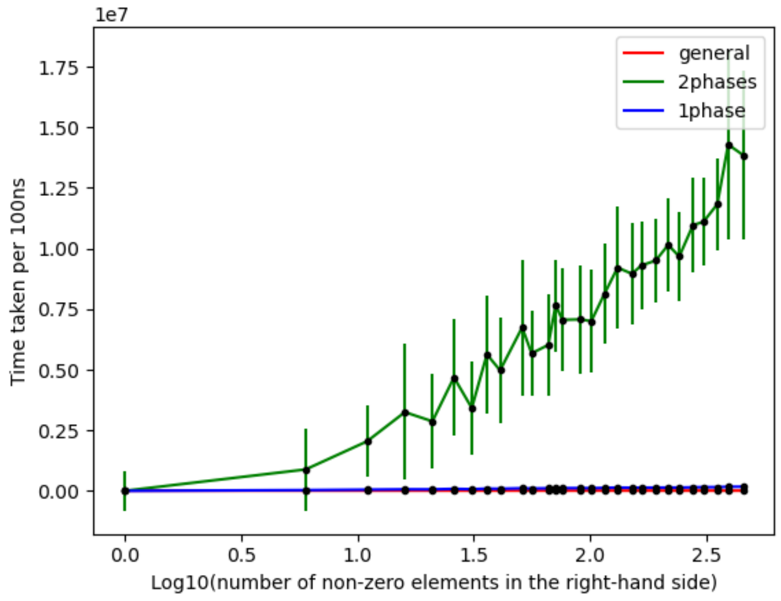


Figure 2: Times taken per methods with the logarithm of base 10 of the number of non-zero elements in the right-hand side with the bars indicating the standard deviation of the time.

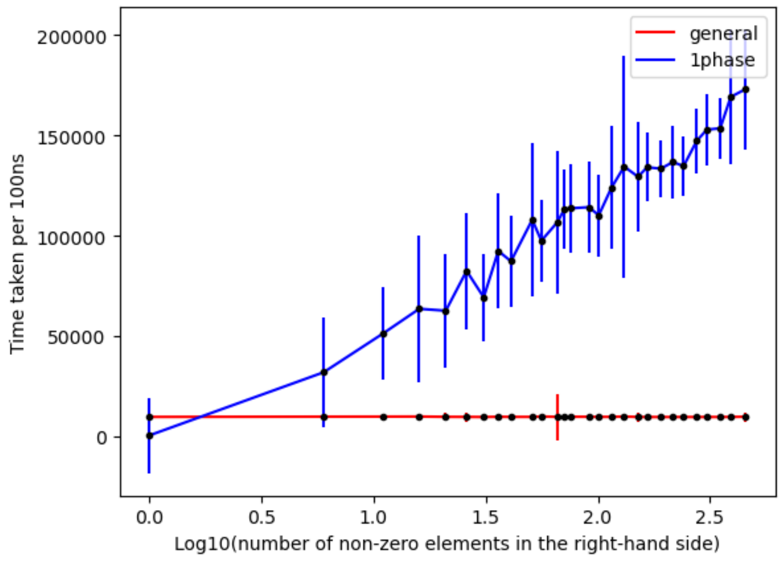


Figure 3: Times taken in 100 ns per methods with the logarithm of base 10 of the number of non-zero elements in the right-hand side without the *Two-Phase algorithm*.

An other parameter that can be tested to have information on the time taken per method is the number of non-zero elements in solution. Indeed, the *Two-Phase algorithm* in the first phase will compute this number of non-zero elements in x. Then, in the second phase, it will iterate over that list in order to compute each non-zero elements of x. Moreover, the *One-Phase algorithm* have a priority queue in order to travel only non-zero elements of x. Therefore, the number of non-zeros elements in the solution intuitively plays a role on the number of operations. However, the *General algorithm* will not try to keep track of the non-zero elements of x, it travels every elements of x which is initialized to b, thus, the *General algorithm* may less depend on the number of non-zero elements of x.

The Figure 4 shows the time taken per algorithm as a function of the logarithm of base 10 of non-zero elements of the solution. One can see that the *Two-Phase algorithm* and the *One-Phase algorithm* are better for low number of non-zero elements in the solution. Nevertheless, for solutions with more non-zero elements the *General algorithm* becomes faster than the other algorithm. This Figure also shows a lower standard deviation compared to the Figure 2 and 3 despite showing 3 times the standard deviation in this plot.

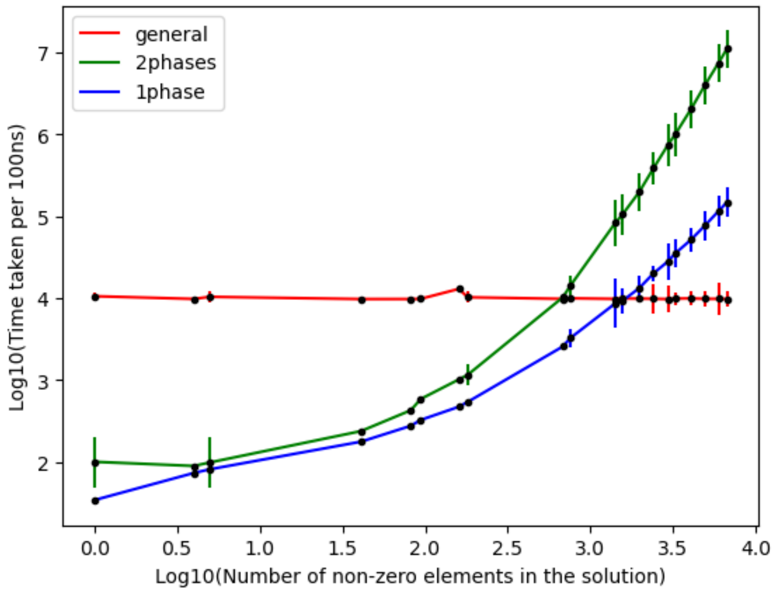


Figure 4: Logarithm of base 10 of the time taken per methods in 100 ns with the logarithm of base 10 of the number of non-zero elements in the solution. The bars indicates 3 times the standard deviation of the y-axis.

When testing on only one matrix L, the number of non-zero elements in the right-hand side seems to be less interesting as a feature than the number of non-zero elements in the solution. Moreover, the standard deviation of the time taken per algorithm is much higher when using the number of non-zeros in b than the one of x, this would lead to a lower accuracy to predict the best algorithm. However, the number of non-zero elements of b is much more easier to obtain than the one of x because the solution is unknown when the choice of the algorithm has to be made. Whereas, b if stored in a format as *compressed sparse column*, its number of non-zero elements can be obtained in only one read.

In order to observe the influence of L, multiple matrix L where generated as for the first experience but with difference percentage of non-zero elements and for each L generated multiple b with different percentage of non-zero elements where generated. The results of this experience shown in Figure 5 is similar to Figure 4 but with a greater standard deviation.

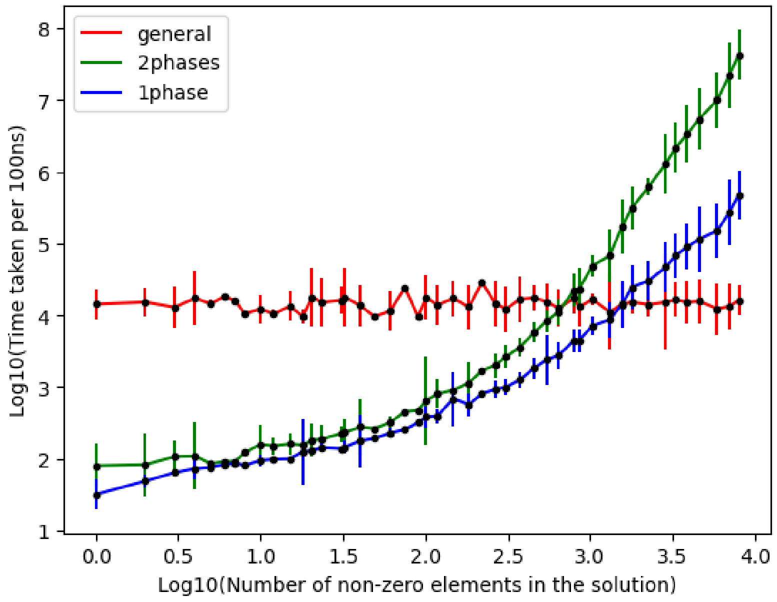


Figure 5: Logarithm of base 10 of time taken per methods in 100 ns with the logarithm of base 10 of the number of non-zero elements in the solution. The bars indicates 3 times the standard deviation of the y-axis. The L matrices generated were with a non-zeros percentage from 0.1% to 0.2375% included with a step of 0.0125%. Therefore, 12 L matrices where generated and for each matrices 100 b where generated, with 10 different percentages from 0.01% to 4.51% included with a step of 0.5%. For each b sparsity 10 samples where generated.

In order to understand what in L may increase the variance in Figure 5. An hypothesis is that sum of the number of non-zero elements of the column i of L where x_i is non-zero influences the time of the computation. If L is in *compressed sparse column format*, this value can be computed with

$$\alpha = \sum_{\forall i \ s.t. \ x_i \neq 0} L.p[i+1] - L.p[i]. \tag{18}$$

Intuitively, this value can influence the time taken by an algorithm because each algorithm will travel the corresponding column i for each non-zero elements of x_i in order to update the value of x. To verify the influence of this parameter on the computation time, multiple L matrices where generated for the same solution x Figure 6 shows the influence of the parameter α on the computation time for the same solution x. In that Figure, one can see that it is possible that for some case, the number of non-zero elements in the solution is not enough to know which algorithm is the fastest, in these cases, the parameter α may be needed to make the best prediction, this parameters alpha seem to influence the computation time of the 3 algorithms linearly.

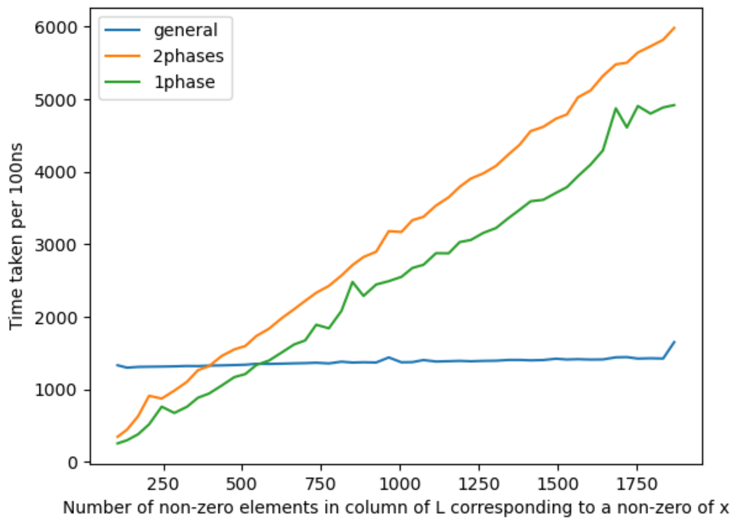
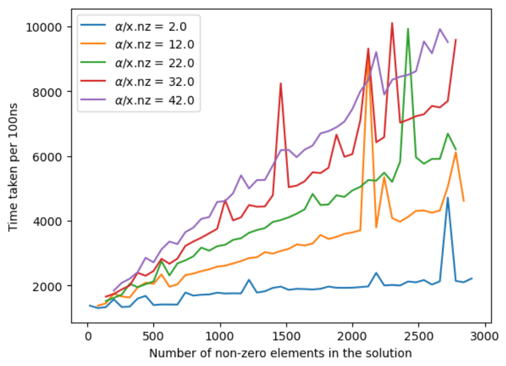
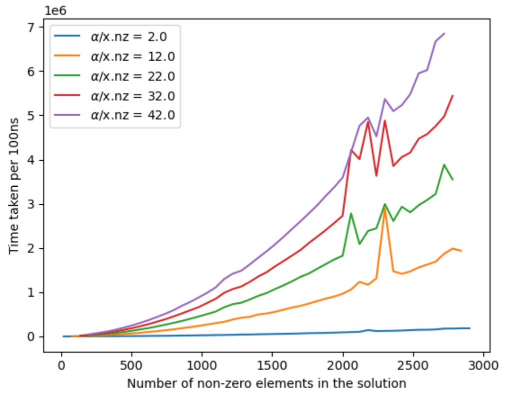


Figure 6: Time taken per methods in 100 ns with the parameter α. The solution x being with a non-zeros percentage of 1%, the L matrices having a parameter α from 100 to 1990 included with a step of 10.

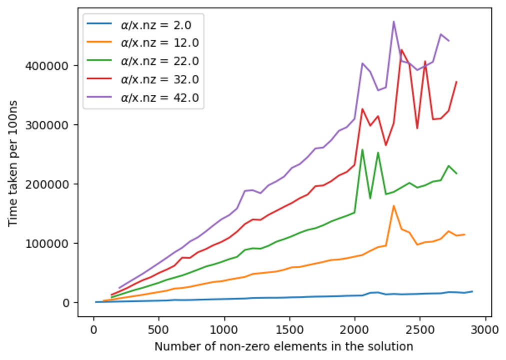
In the Figure 7, multiple L were generated with multiple x in order to show the influence on the computation time that the number of non-zero elements in x and α have at the same time. For each x multiple L were generated with a parameter *alpha* proportional to the number of non-zero elements in x. One can also see that the *Two-Phase algorithm* seem to have a non-linear behavior.



(a) *General algorithm*



(b) *Two-Phase algorithm*



(c) *One-Phase algorithm*

Figure 7: Computation time in 100ns of the three different algorithms with different solution x and with different parameter α.

4 Dataset

The dataset contains around 2.563×10^6 samples which seem to be more than enough, therefore, in order to save computation time and because the number of samples per datasets is unbalanced. Indeed, each dataset have a constant size of b and some datasets have more than 7×10^4 samples whereas some have less than 10^3 samples, as one can see in Figure 9. To balance the entire dataset, each subdataset has been sampled with the following strategy. If the dataset has less than 4×10^3 samples, the dataset is entirely drawn and if it has more than 4×10^3 samples, 4×10^3 samples are uniformly drawn from the dataset, the remaining samples after the under-sampling is 7×10^5 samples.

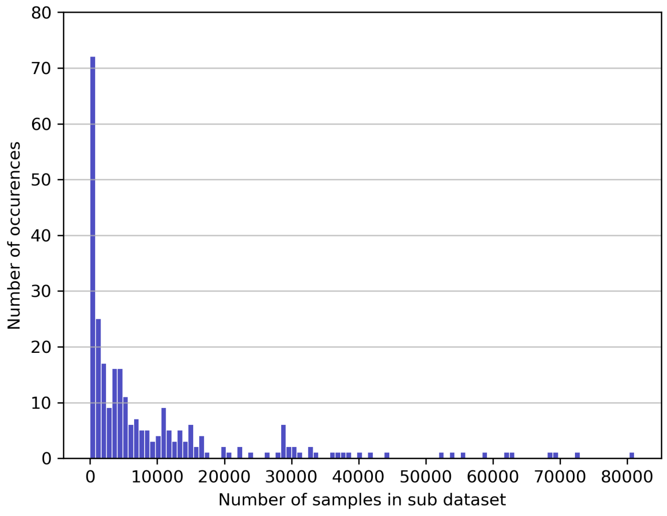


Figure 8: Histogram of the occurrences of sub datasets with their number of samples.

One aspect that one must pay attention to is the dataset balance for the predicted class. The classes for the problem that is tackled is, actually, the number of the algorithm, 0 is for the *General algorithm*, 1 is for *Two-Phase algorithm* and 2 is for the *One-Phase algorithm*. The classification problem that this work is trying to solve is to predict the class number of the fastest algorithm. One can see in Figure 9 the proportion in the dataset each algorithm is the fastest. As one can see in that figure, the distribution of the classes is quite unbalanced, thus, this must be considered while choosing the metrics to evaluate the classifier. Indeed, choosing the **accuracy** can lead to a wrong evaluation of models. For example, one classifier could predict every time that the *One-Phase algorithm* is the fastest algorithm. In that case, the accuracy is misleading. Indeed, this simple classifier could achieve an accuracy of 70% which could be considered great but is not giving any useful information.

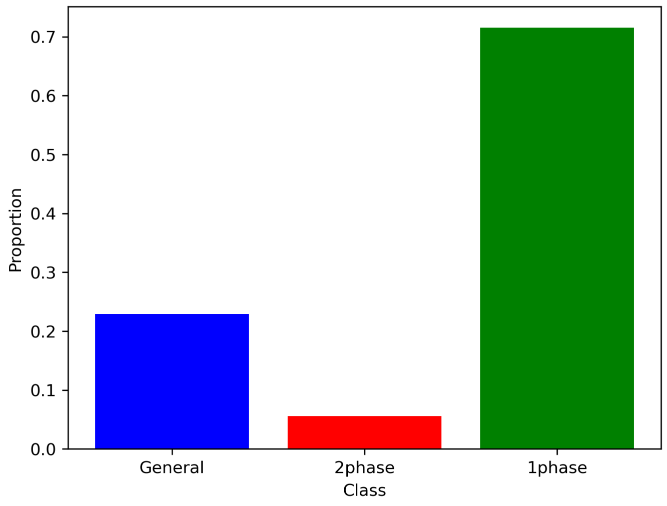


Figure 9: Bar chart of the proportion of times each algorithm is the fastest in the undersampled dataset.

Because each instance is solved two times, one time it is $Ly = b$ that is solved and then it is $Ux = y$ that is solved, each being a sample of linear triangular solve, the final dataset contains, thus, when merging these two linear system forms, a total number of sample equal to 1.4×10^6. The dataset is, actually, made of 253 subdataset each subdataset containing systems of linear equations that are depended. The dataset will be divided into 2 parts, one part will be containing 20% of the sub-datasets and the remaining 80% will be used in a cross validation with 10 folds. When dividing the dataset, the division will be between the subdataset, each subdataset will virtually count as a sample. This is to prevent to have samples from the same dataset in both training and testing sets. This is done because the data in a sub-dataset may have similarities. Indeed, in a subdataset, the simplex method is used to solve linear problems and leading to multiple solves of the system $LUx = b$. These solves may share some information. In the simplex method, when changing vertex, the matrix A changes by only a column. This matrix is later transform into its LU form and then solved, this LU will most likely have similarity. Therefore, having samples from the same subdataset in both training and testing may lead to too optimistic performance because of that similarity. The similarity can be used by machine learning methods. It can for example use the knowledge on which simplex solve or which subdataset the sample is from to to better fit the data. However, this context is not an information that could be obtained at run time, thus, it could lead to wrong performance assessment. This is why when testing performance, the test set should be made of subdatasets which are not known by the model.

5 First model

In Section 3, one could notice that the number of non-zeros in the right hand side plays a role in the time that the algorithm will take to solve the system. Moreover, the number of non-zero elements in the right-hand side can be easily obtained, thus, one naive way to create a model is to use only this information. In order to predict which algorithm is the fastest, one regression model will be trained per method to predict the method computation time. Then, the algorithm that will be predicted the fastest will be the one with the lowest predicted time. Two models will be tested, *LRNNZB (a)* and *(b)*. The name *LRNNZB* stands for Linear Regression using the Number of Non-Zero elements in b. *LRNNZB (a)* uses a linear regression only based one the non-zeros elements in b. Whereas, *LRNNZB (b)* uses another feature for the regression of the *General algorithm* computation time, the size of the right-hand side.

The linear regressions does not predict well the computation time for the 3 algorithms when using the number of non-zeros elements in b. Indeed, the mean R^2 score on the test folds is -0.401 for the *General algorithm* regression, -7.019 for the *Two-Phase algorithm* regression and -4.690 for the *One-Phase algorithm* regression. The Figure 12 shows the time predicted for every algorithm, one can see that the *Two-Phase algorithm* will never be predicted as the fastest and the one-phase algorithm will be predicted the fastest for problems with a number of non-zero elements in b lower than $10^{1.3} \approx 20$ and for every other cases the *General algorithm* will be predicted as the fastest. Even if the regressions are not fitting well the computation times, the classification of the algorithms seems to have significant results. Indeed, model choosing naively the *One-Phase algorithm* takes $127.99s$ when averaging the times taken by models predicted on the test folds and for a model choosing at random an algorithm, its choices leads to a $2456.25s$ computation time. The average computation time per fold when the algorithm is chosen by the model *LRNNZB (a)* is $31.508s$ whereas if the model was always choosing the best algorim, the computation time would be $7.242s$. One of the metric that evaluate the performance of the classification model is the *balanced-accuracy* which is used instead of the *accuracy* because the dataset for the classification is imbalanced as shown in Section 4. The *balanced-accuracy* of *LRNNZB (a)* is 0.506.

The total computation time does not show all the information. Indeed, the classifier could do well for some problems that are highly represented in the dataset and predict the wrong algorithm for some other part of the dataset. Indeed, if some instances are not well represented in the dataset, the impact on the total computation time is less important. The Figure 10 shows the average computation times taken by the predicted algorithm with different range of the right-hand side and also the time taken by always choosing the best algorithm. One can see in that Figure that the time taken by the predicted algorithm is greater for systems with a right-hand side size greater than 26×10^4. One can see on Figure 11 that the error in the high range of right-hand side size seems to be due to the proportion of time the *General algorithm* is predicted instead of the *One-Phase algorithm*.

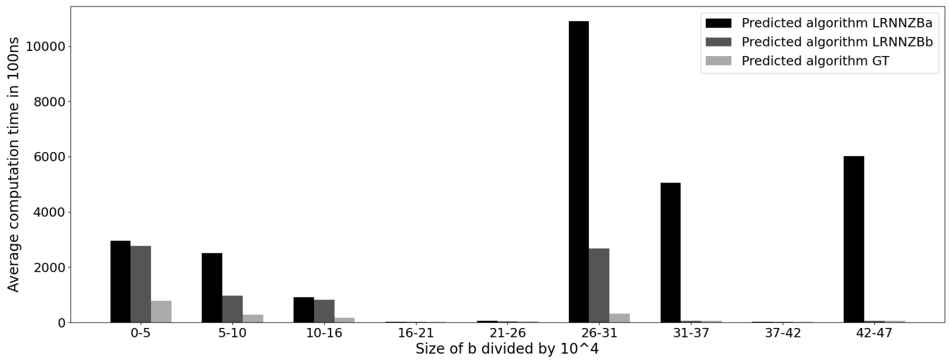


Figure 10: Bar plot of the average computation time by the predicted algorithm by *LRNNZB (a)*, *(b)* and the model that would lead to the lowest computation times for different ranges of the right-hand side.

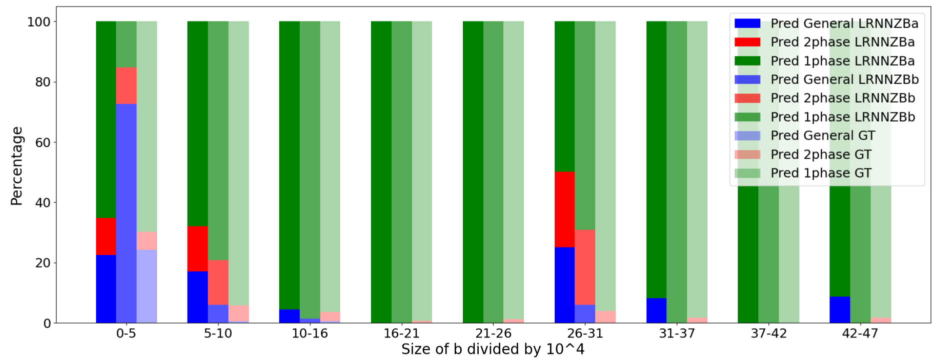


Figure 11: Bar plot showing the proportion of time each algorithm are predicted best with the size of the right-hand side. "Pred" standing for predictions and "GT" for ground truth.

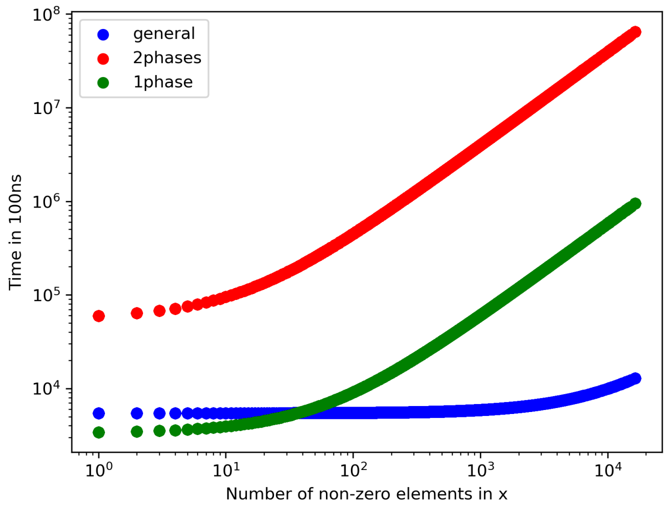


Figure 12: Scatter plot of the predictions of the times taken per algorithm of *LRNNZB(a)* with the number of non zeros elements of b.

In Section 3, the *General algorithm* seems to be less dependent on the number of non-zero elements and because it travels every elements in the right-hand side, the time to solve should intuitively be more dependant on the size of the right-hand side. Therefore, another model was used for the regression of the *General algorithm* computation time using as feature the size of the right-hand side instead of its number of non-zero elements.

The R^2 score for the regression of the *General algorithm* increased, it is equal to 0.652 whereas previous model had a $R^2 = -0.401$. However, the classification of the algorithm performed by this model, which will be referred as *LRNNZB (b)*, seem to be less accurate than the previous model with a *balanced accuracy* of 0.459 instead of 0.519 for *LRNNZB (a)*. Even if the balanced accuracy is lower, the total computation time is better with $27.338s$ for *LRNNZB (b)* instead of $31.508s$ taken by the algorithm chosen by *LRNNZB (a)*. One can also see in Figure 10 that *LRNNZB (b)* choices have a better average computation time than the one of *LRNNZB (a)* for problems with a right-hand side size in the interval $[26 \times 10^4; 47 \times 10^4]$. The Figure 11 shows that for the right-hand side size range in $[26 \times 10^4; 47 \times 10^4]$, the proportion of time the *General algorithm* is chosen is lower for *LRNNZB (b)* compared to *LRNNZB (a)*, which is closer to the ground truth. However, for range with a right-hand side in $[0; 5 \times 10^4]$ the *General algorithm* is predicted best more than 70% of the time but it should be around 20%. Even if the *General algorithm* is to often chosen the average computation time is similar to *LRNNZB (a)* for that range of right-hand side size.

In further section, the goal is to create a model that is able to better choose the fastest algorithm than the simple model *LRNNZB (b)* so that the average computation time on the test folds is lower.

6 Proxy

6.1 Model based on the true number of non-zero elements in the solution

As shown in Section 3 for simple cases, the number of non-zero elements in solution can give some information about the computation time for the *One-phase algorithm* and *Two phase algorithm*. However, this was not tested for a large number of samples . Therefore, to show that the number of non-zero elements can help to better classify the different algorithms, 3 regression models will be trained similarly to Section 5. Nonetheless, this time the feature used is the number of non-zero elements in the solution. The goal being to prove that using the number of non-zero elements of the solution can be used as proxy to predict the time of each algorithm.

Three classification models where tested. These will be referred as *LRNNZX* which stands for Linear Regression using the Number of Non-zeros of X. The first one, *LRNNZX (a)*, have the 3 linear regressions trained only using the number of non-zero elements of the solution as feature. The second one, *LRNNZX (b)*, has a regression model using the size of the right-hand side as feature to predict the *General algorithm* computation time. Similarly to *LRNNZX (a)*, *LRNNZX (b)* uses the number of non-zero elements in the solution to predict the computation time of the two remaining algorithms. The third one, *LRNNZX (c)*, uses the same regression model as *LRNNZX (b)* for the regression of the *General algorithm* time. However, the regression of the *Two-phase algorithm* uses the number of non-zero elements in the solution squared and the regression of the *One-phase algorithm* time is made using the number of non-zero elements in the solution times its logarithm of base 10.

The model *LRNNZX (a)* has a lower cross validation R^2 score than the model *LRNNZB(b)* which is only based on b information. Indeed, the average R^2 score on the test folds is -0.397 for the *General algorithm* computation time whereas the regression of the *General algorithm* of the model *LRNNZB(b)* obtained 0.652. These results suggest that using the size of b to predict the *General algorithm* computation time is better than using the number of non-zero elements in the solution. As the *General algorithm* time regression, the average R^2 score on the test folds is lower for the regression of the *two-phase algorithm* and *one-phase algorithm* have a lower cross validation R^2 score with a -11.442 and -5.967 respectively whereas *LRNNZB(b)* obtained a R^2 of -7.019 and -4.690 respectively. The conclusion that should be made is that the number of non-zero elements in the solution is not usefull to predict which algorithm is the fastest. However, even if these results are not great, the time taken by the algorithm chosen by the model *LRNNZX (a)* on the test folds is lower than the one of *LRNNZB (b)* with $13.383s$ instead of $27.337s$. The balanced accuracy also is higher which also reinforce the claim that *LRNNZX (a)* is a better classifier than *LRNNZB (b)*. These results are shown in Table 1.

Model Name	Algorithm	R^2 train	R^2 test	Average computation time (s)	Balanced accuracy
LRNNZB(b)	General	0.739	0.652	27.337	0.459
	Two-Phase	0.088	-7.019		
	One-Phase	0.081	-4.690		
LRNNZX(a)	General	0.001	-0.397	13.383	0.509
	Two-Phase	0.18	-11.442		
	One-Phase	0.11	-5.967		
LRNNZX(b)	General	0.739	0.652	10.993	0.532
	Two-Phase	0.18	-11.442		
	One-Phase	0.11	-5.967		
LRNNZX(c)	General	0.739	0.652	9.472	0.611
	Two-Phase	0.093	-1.195		
	One-Phase	0.107	-5.300		

Table 1: Results of the LRNNZX models compared to the model LRNNZB(b). The average computation time is the average of the computation time computed on the 10 test folds. The Balanced accuracy is computed by averaging the balanced accuracy obtained on the test folds when training the model on the other folds.

The model *LRNNZX (a)* seem to have the same problem as *LRNNZB (a)* concerning the computation time of instances with a size of b in $[26 \times 10^4, 31 \times 10^4]$, as one can see in Figure 13.

The model *LRNNZX (b)* has performance similar to *LRNNZB (b)* for the regression of the *general algorithm* time with a R^2 score of 0.652, which is coherent because it uses exactly the same feature as *LRNNZB (b)*. Moreover, the R^2 for the two other algorithms are equal to the one of model *LRNNZX (a)* for similar reasons. However, by using the size of b to predict the *General algorithm* computation time, the average computation on the test fold improved with a time of $10.993s$ which is the fastest so far whereas the model *LRNNZX (a)* took $13.383s$. The $balanced - accuracy$ is also the best one so far with 0.532.

The model *LRNNZX (c)* using functions of the number of non-zero elements in the solution for the regression of the *Two-phase algorithm* time and the *one-phase algorithm* time does not performed well but it has a better R^2 score on the test folds in average. Indeed, the R^2 for the *two-phase algorithm* and the one of the *one-phase algorithm* is higher than the previous model with -1.196 for the regression of the *two-phase algorithm* computation time and -5.3 for the one of the *one-phase algorithm*. Even if the score is higher, the regression does not seem to have significant result. However, the classification part had the best results with the choice of algorithm leading to a total computation time of $9.472s$, with $balanced - accuracy$ of 0.611.

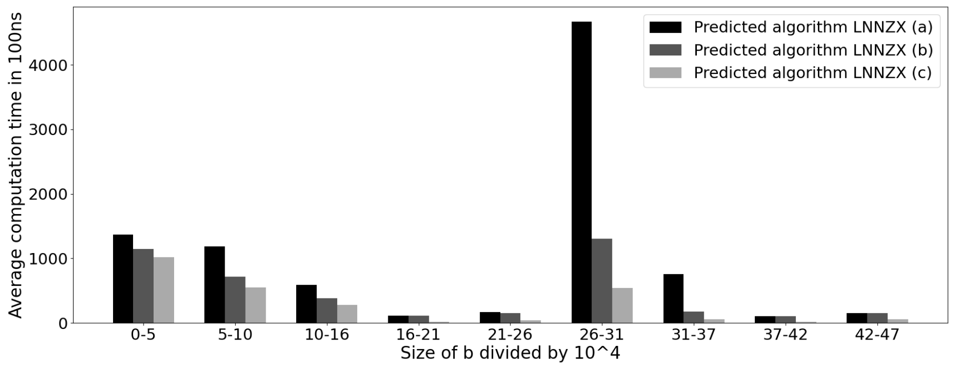


Figure 13: Bar plot of the average computation times of the algorithm predicted with the size of the right-hand side for *LRNNZX (a), (b)* and *(c)* .

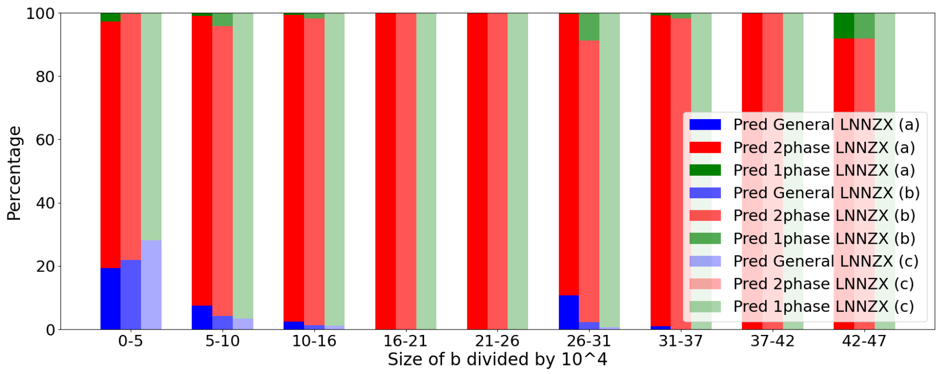
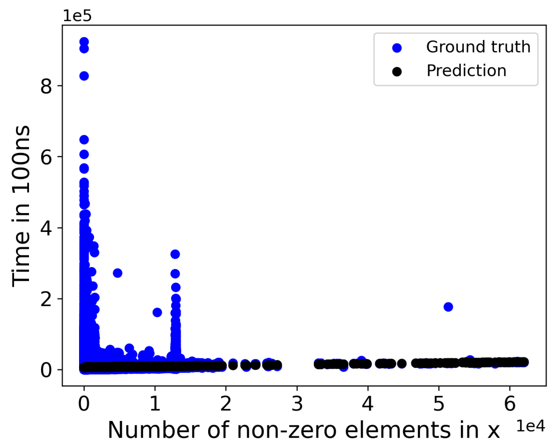
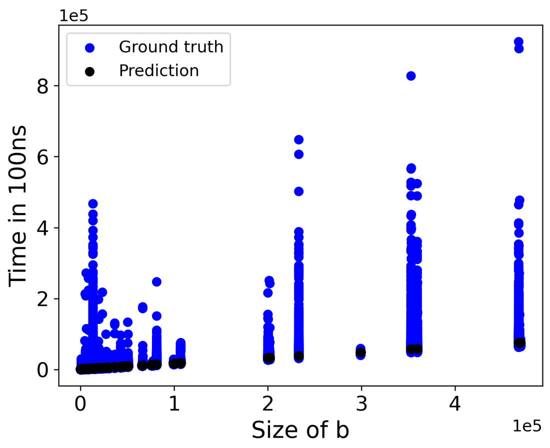


Figure 14: Bar plot showing the proportion of time each algorithm are predicted best for every model *LRNNZX* with different range of size for the right-hand side.

In Figure 15, the regression of the *General algorithm* time seems to be better when using the number of non-zero elements of x than when using the size of b. However, the R^2 score of the first regression is much lower than the one of the second regression. This can due to the fact that the low number of non-zero elements in the solution which corresponds to a spike in Figure 15a is the most important part of the dataset. Actually, 81.364% of the samples of the dataset dedicated to the cross validation have a number of non-zero elements in the solution lower than 200, therefore, having a bad regression for that part of the dataset should lead to a low $R^2 score$. Having a very high variance for this bandwidth of number of non-zero elements in x is plausible because the *General algorithm* will initialize x to b, regardless of the solution. Therefore, having a solution with the same number of non-zero elements but one with a b with a much lower size than another can lead to completely different computation time depending on the difference in b sizes. In the later case, the *General algorithm* will spend most of its computation time initializing the vector x. However, even if the R^2 score is better when using the size of b as feature the regression is far from perfect as one can see in Figure 15b.



(a) The regression model used by LRNNZX (a).



(b) The regression model used by LRNNZB (b) and LRNNZX (b), (c).

Figure 15: Scatter plot of linear regressions used to predict the *general algorithm* computation time of the validation set.

The regressions of the *two-phase algorithm* and the *one-phase algorithm* computation time seem to be better when using the number of non-zero elements in x compared to the number of non-zero elements in b. Similarly to the regression of the *General algorithm* computation time, most of the dataset samples have a low number of non-zero elements in b and x, therefore, having a better regression in that range should lead to better results. In Figures 18a and 19a, one can see that there is a peak in the beginning of the x-axis which is less dense in the the Figures 18b and 19b, which may increase the R^2 when using the number of non-zero elements in x instead of the one of b. One can also see that the slope of the regression seem too high leading to greater residual the higher the number of non-zero elements in x is. This may be due to the high peak and that the models tries to make a trade off between the error made on the peak and the one on the high range of the x-axis.

One can see the regressions of the *Two-phase algorithm* and the *One-phase algorithm* computation time of the model *LRNNZX (c)* in Figures 18c and 19c. These regressions seem to be worse than the one shown in Figures 18a and 19a which are used in model *LRNNZX (a)* and *(b)*. However, these regressions leads to a better computation time when averaging the computation time on the test folds. The Figure 14 shows the proportion of time each algorithm is predicted the fastest by the *LRNNZX* models over different sizes of problems. The models *LRNNZX (a)* and *(b)* seem to predict the *Two-phase algorithm* too often knowing that this class was, actually, the one which was the less often the fastest, as seen in the class distribution of the dataset. One can see the intersections of the two algorithm regressions in Figure 20, in this figure one could think that the *one-phase algorithm* will be predicted best most of time, however, the part which is the most important one is the part where the *two-phase algorithm* is predicted the fastest. Indeed, the intersection occurs in $x = 10^{2.1}$ which corresponds to a number of non-zero elements in the solution equal to 154 and 79.4% of the samples dedicated to the cross validation have a number of non-zero elements lower than 154. Therefore, the *Two-phase algorithm* will be predicted best most of the time which is what can be seen in Figure 14. In the Figure 20c, it is shown that by using the number of non-zero elements squared to predict the time taken by the *Two-phase algorithm*, the model *LRNNZX (c)* will actually predict the *one-phase algorithm* every time instead of the *two-phase algorithm*. This strategy seems to be a better choice in average for the different problems size as shown in Figure 13. The Figure 16 and Figure 17 shows the comparison between the model *LRNNZX (c)* and the model *LRNNZB (b)* which was the one with the lowest total computation time using only information on b. In that Figure, GT, which is the model that would always predict the fastest algorithm, is also shown. In these Figures, one can see that the model *LRNNZX (c)* is much closer to the ground truth algorithm prediction and computation time than the model *LRNNZB (b)*. Furthermore, the error of the model *LRNNZB (b)* where the *general algorithm* seemed to be predicted the fastest to frequently for the size of b in between $[0, 5 \times 10^4]$ is not present in the model *LRNNZX (c)*.

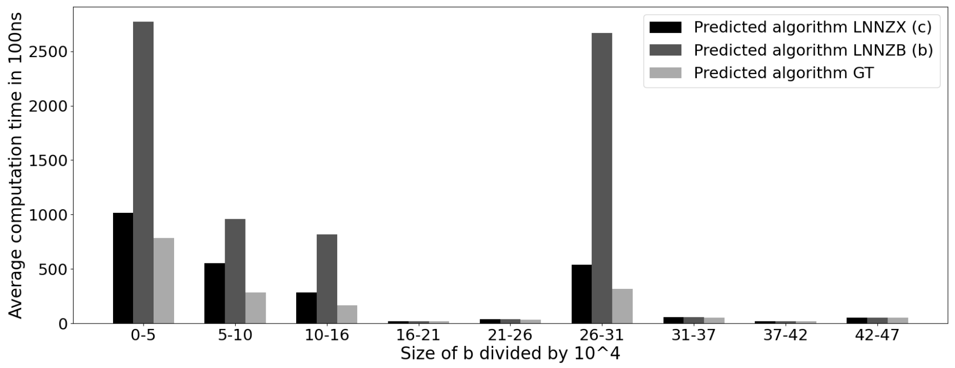


Figure 16: Bar plot of the average computation times of the algorithm predicted with the size of the right-hand side for *LRNNZX (c)*, *LRNNZB (b)* and a model that would predict the best algorithm every time.

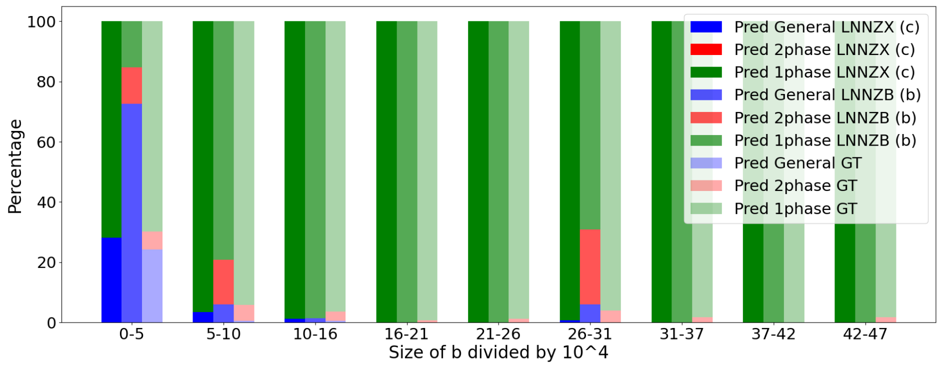
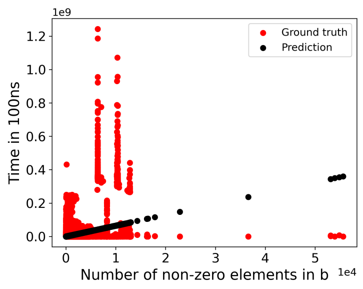
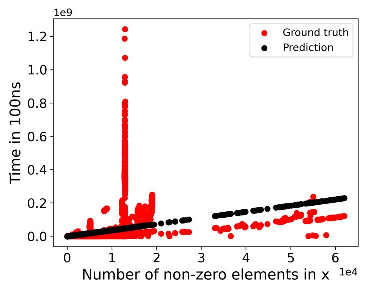


Figure 17: Bar plot showing the proportion of time each algorithm are predicted best by the model *LRNNZX (c)*, *LRNNZB (b)* and a model that would predict the fastest algorithm every time. The proportions are shown with different ranges of b sizes.

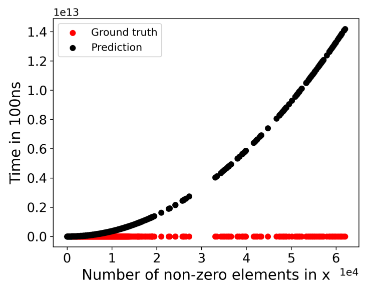
The R^2 score of the regression models seem to show that knowing the number of non-zero elements in the right-hand side is not enough to precisely predict the time taken by the 3 algorithms, another feature might be able to improve the regressions. Even if there is an unknown feature capable to improve the predictions of the computation time, these models shows that knowing the number of non-zero elements in the solution helps to improve the performance of the total computation time compared to *LRNNZB (b)*. However, knowing the number of non-zero elements in the solution is not possible before solving the systems. Therefore, the goal of the next chapter will be to find features and create a model that is able to predict the number of non-zero elements in the solution without, actually, solving the system and maybe to find the feature that combined with the number of non-zeros elements in the solution would improve the predicted computation time. These models should be fast enough to be able to win time over the best model using only information on b which is *LNNZB (b)*.



(a) The regression model used by LRNNZB(a) and (b).

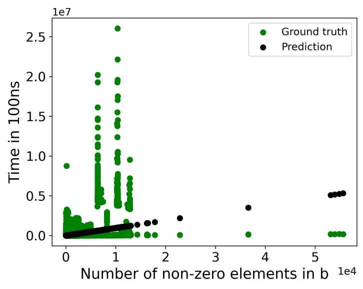


(b) The regression model used by LRNNZX (a) and (b).

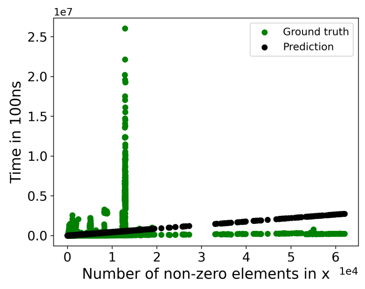


(c) The regression model used by LRNNZX (c).

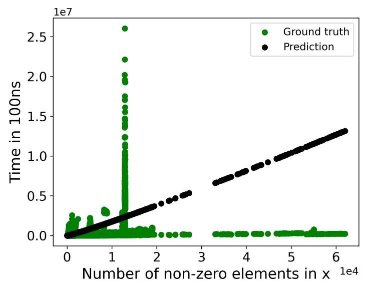
Figure 18: Scatter plot of linear regression used to predict the *two-phase algorithm* computation time of the validation set.



(a) The regression model used by LRNNZB (a) and (b).

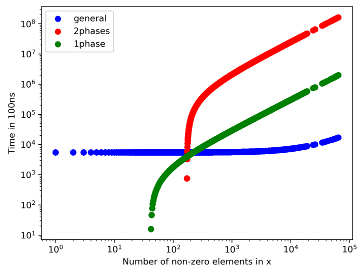


(b) The regression model used by LRNNZX (a) and (b).

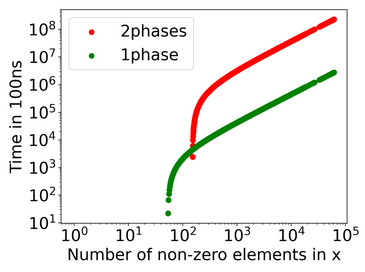


(c) The regression model used by LRNNZX (c).

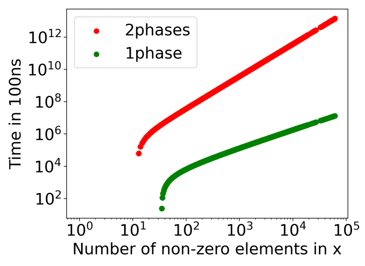
Figure 19: Scatter plot of linear regressions used to predict the *one-phase algorithm* computation time of the validation set.



(a) Prediction of the validation set of *LRNNZX (a)*.



(b) Prediction of the validation set of *LRNNZX (b)*.



(c) Prediction of the validation set of *LRNNZX (x)*.

Figure 20: Scatter plot of linear regressions used to predict the computation time of the validation set by the model *LRNNZX (a)*,*(b)* and *(c)*. The general algorithm time regression is not represented for the model *LRNNZX (b)*,*(c)* because it uses the size of b as feature and not its number of non-zeros.

6.2 Feature engineering

At first, features are proposed regardless of their computation times in order to see what values are relevant to predict the number of non-zeros of the solution.

To solve a system that is in a LU form 3 algorithms are proposed and to make the decision of which solving algorithm to choose. Some data are available with just a read if the matrices are stored in a *compressed sparse column* format. Such as, the number of non-zeros elements of the matrices L or U and its size, the number of non-zeros in the right-hand side. Another available information is the solve-type which take the value 1 if the system is in the form $LUx = b$ and 0 if the system is in the form $x^T LU = b^T$.

The other features that are tested are less easily computed. The following features will need an order of complexity proportional to the number of non-zero elements present in the right-hand side to be computed.

- The first feature that will be called f_0 is to compute the number of non-zeros of each column j which has a non-zero element at the corresponding index j of the right-hand side, the set of these column indices will be called J. The feature f_0 is motivated by the fact that if a matrix L has no row index duplicates for all the elements of the columns j and if in the solution some element computation does not give 0, then it represent a minimal bound of the number of non-zero elements in the solution. The computation of the feature is shown in (19). In this equation, the matrix A represent either L or U depending on the system and b represent the right-hand side. The computation have a complexity of the order of the number of non-zeros elements of the right-hand side thanks to the condense sparse column or row format.

$$f_0 = \sum_{\forall j \in J} A.p[j+1] - A.p[j] \tag{19}$$

  where $J = \{j \mid b_j = 0\}$.

  This feature f_0 is computed similarly in the case that the matrix is upper triangular. Moreover, if the solve type is in the form of $x^T L = b^T$, then, computation is exactly the same except that in that case the matrix is loaded in the *compressed sparse row* format instead of the *compressed sparse column* format.

  The problem of this feature is that it is completely blind to the impact that a non-zero element could create at computation since one non-zero will lead to more and more non-zeros depending on the matrix L or U, and to the fact that multiple columns may have the same row, thus, this feature may count more than one time a non zero element.

- Another feature that will be used is a score that tries to capture the possibility to create new non-zeros. This feature also uses the index of the right-hand side non-zero elements and find the corresponding column in the triangular matrix involved in the system. The score is computed using (20). The idea behind this feature is to highlight the fact that for system $Lx = b$ and system of the type $x^T U = b^T$. Intuitively, the lower is the index in the right-hand side, the higher is the probability to create new non-zeros elements.

$$f_1 = \sum_{\forall j \in J} A_{nb\_row} - j \tag{20}$$

where A_{nb_row} is the number of row of A.

In the case that the system is in the form of $x^T L = b^T$ or the system is $Ux = b$, then, the importance changes. The higher is the row (resp. column) number, the higher is the probability to create non-zeros. In these cases, this feature is computed with (21)

$$f_1 = \sum_{\forall j \in J} j. \tag{21}$$

- The third feature is similar to the second except that each element of the sum will be weighted by the number of non-zero elements in the column or row of the matrix depending on the type of solve. For the system of the type $Lx = b$ and $Ux = b$, it is the number of non-zeros element in the column that is taken. If the system has the type $x^T L = b^T$ or $x^T U = b^T$, then, it is the number of non zeros element in the row that is taken. Similarly to the first feature, the computation is actually the same for system with x^T, it is just the format used to load the matrices that changes. The corresponding equation for the systems $Lx = b$ and $x^T U = b$ is shown in (22) and the one for $x^T L = b^T$ or $Ux = b$ corresponds to (23).

$$f_3 = \sum_{\forall j \in J} (A_{nb\_row} - j) \times (A.p[j+1] - A.p[j]) \tag{22}$$

$$f_3 = \sum_{\forall j \in J} j \times (A.p[j+1] - A.p[j]) \tag{23}$$

- The last feature that is computed is similar to the third feature but it is not weighted but powered. This may better fit the number of non-zeros since it may better fit the snowball effects of one non-zero element creating other non-zero elements, this corresponds to compute (24) for $Lx = b$ and $x^T U = b^T$ and (25) for the system $x^T L = b^T$ and $Ux = b$.

$$f_3 = \sum_{\forall j \in J} (A_{nb\_row} - j)^{A.p[j+1] - A.p[j]} \tag{24}$$

$$f_3 = \sum_{\forall j \in J} j^{A.p[j+1] - A.p[j]} \tag{25}$$

6.2.1 Feature importance and computation time

Some features have been proposed but their importances have not been tested. Furthermore, using irrelevant features in a model can result to more time to make a prediction. It is even more important in a context where the time window to make a prediction is very low. That is why feature selection can be a valuable option in this work.

The feature importance is computed using decision tree as explained in sectionDecision Tree. The importance of the features is shown in Figure 21. The feature computed such as the size of the system and the feature f_0 seem to have played a role in the creation of the decision tree. This figure is only to show that some feature that have been proposed are not completely useless, the score of the feature will not be used for feature selection. The features that seem the more important are the size of the matrix L or U depending on the system, the feature f_0 and the number of non-zero elements in the triangular matrix.

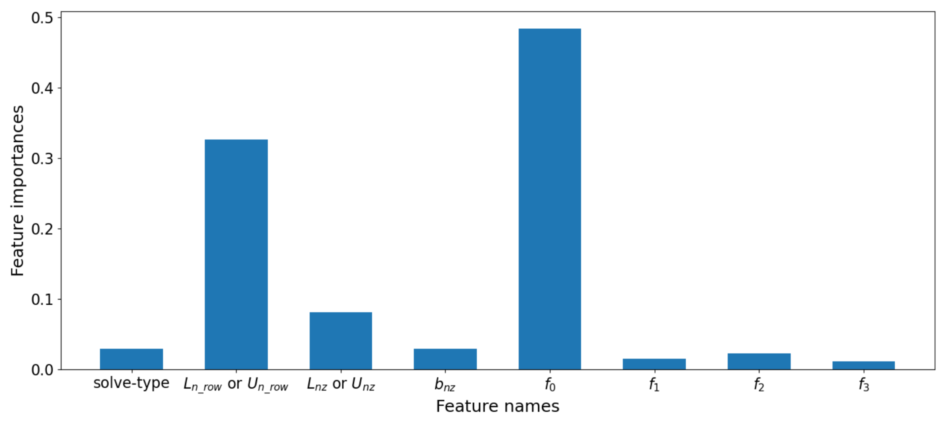


Figure 21: Feature importance for the prediction of the number of non-zero elements in the solution NNZ standing for Number of Non-Zero elements.

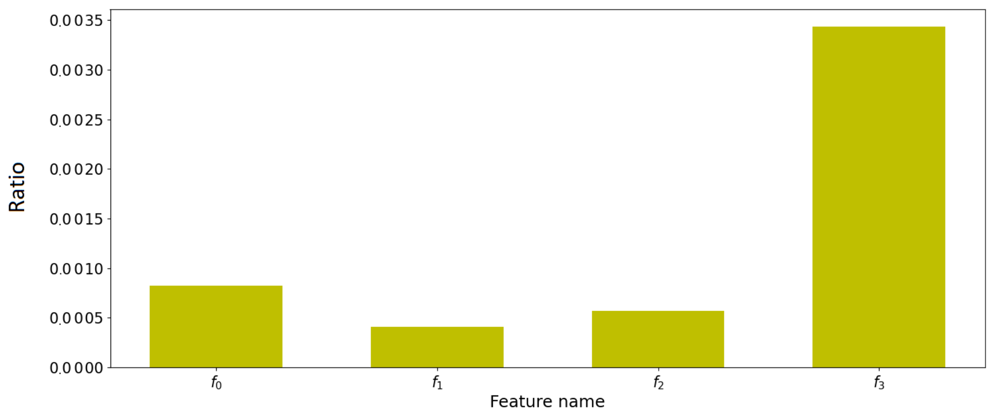


Figure 22: Ratio between the feature computation time and the time difference between the best and second best algorithm.

In order to see if there is still time to win when computing the features when choosing the best model over the second model, one can see the Figure 22 that shows the average of the ratio between the feature computation time and the time difference between the fastest algorithm and the second fastest algorithm. This computation is done over the training set to avoid making a selection of the feature based on the test set. Indeed, doing so would lead to a fit of the test set and maybe a bias on the performance analysis of the model. One can see that the computation times of the features are lower than 1% compared to the time difference between the two fastest algorithms. On average, computing all the features will lead to a small overhead that leaves more than 99.48% of the time window between the first and second choice algorithm.

Previous results were showing the average proportion that the feature computation times takes over the possible time to win if the best algorithm were chosen instead of the second choice. Therefore, it is still possible that computing these features leads to an overhead greater than the possible time to win in some cases. The percentage of these cases was computed over the training set and represent, actually, 1% of this set. For those systems, the average computation time is very low 1.30ns for the fastest algorithm and 1.39ns for the second fastest. This leaves around

$0.09ns$ to compute the features and in those cases the computation of f_0 is $0.2ns$. Those cases are not considered for now but it can be great to know more about those cases to maybe use a simpler method to predict the fastest algorithm.

6.3 Proxy model selection

The model predicting the number of non-zero elements in the solution will be trained with the same folds used to train the models **LRNNZB** and **LRNNZX**. However, some of the plots that will be shown in this section needs a training and test set, it is not possible to make an average of the measure on the test folds. Therefore, these plots are made using the test set methods by taking 7 folds as learning set LS and the 3 other folds as second TS. The 30% which was put aside before the model must not be used, these sample must be kept and used only for the model assessment.

In order to train the model predicting the time in next section, the model will also be trained on the 10 folds as previous model. In that case, for the fold i which is used to evaluate how the model perform on new data, the model predicting the proxy should not be trained on that folds. Indeed, this will lead to an overestimation of the performance of the model predicting the time. To avoid this situation, the model trained to predict the proxy will be trained on the same folds as the model predicting the time.

In order to measure if an algorithm is fast enough, a metric will be used to measure the efficiency of the algorithm. This metric is the ratio between the mean time of prediction over the difference between the time taken by the fastest algorithm and the one of the second fastest algorithm. The metric is defined as

$$\beta_{1,2}(t\_pred, t) = \frac{\sum_{i=1}^{n} t\_pred_i}{\sum_{i=1}^{n} \min\limits_{\forall j \setminus \{\text{argmin}_j(t_{ij})\}}(t_{ij}) - \min\limits_{\forall j}(t_{ij})}.$$

The time t_pred_i is the time taken to make the prediction of the sample i including the feature computation time and the value t_{ij} is the time taken for the solver j to solve the sample i.

This metric being an average, there is still a possibility that using a model to predict the number of non-zeros adds an overhead that is greater than the time difference between the two fastest algorithm. Therefore, a second metric is used to measure the ratio of such cases on a dataset. This value uses the same input as $\beta_{1,2}$ and it also uses n, which is the size of the dataset. This metric is defined as

$$\gamma_{1,2}(t\_pred, t) = \frac{\sum_{i=1}^{n} 1\Big(t\_pred_i > \big(\min\limits_{\forall j \setminus \{\text{argmin}_j(t_{ij})\}}(t_{ij}) - \min\limits_{\forall j}(t_{ij})\big)\Big)}{n}.$$

6.3.1 Linear Regression

One of the model that is tested to predict the number of non-zero elements in the solution is the ordinary least square regression. If the data follows a linear model, thus, the residuals follows a normal distribution. This means that if the linear regression fits best the model, the residuals of the fitted models only becomes noise which is present in the dataset. However, when training a linear model with the 7 first folds and using the 3 other folds as test set , the residual does not seem to be following a normal distribution as shown in Figure 24.

Nonetheless, for the number of non-zero elements between 10000 and 25000, the residual seem to increase linearly with the number of non-zero elements which can be due to a too low slope of the model for that range of output. Therefore, the error increases with the output value.

One can also see in that figure that most of the error seems to be located in the low part of the output, which is for a predicted value lower than 10000. Furthermore, the R^2 score does not seem bad considering the simplicity of the model. It is equal to 0.395 when using the training set and 0.372 when using the test set. However, when using cross-validation, the average R^2 score on the test folds is -1.644×10^{13} and the average R^2 score on the training folds is 0.389. the reason why the R^2 score on the test folds is so low in average is actually because there is a folds in particular that the linear regression does not predict well when not trained on. Indeed, without the fold having an R^2 on test of -1.1644×10^{14}, the average R^2 score becomes 0.342.

The time added to predict the number of non-zeros elements in x is very low compared to the time needed to compute the features. Indeed, the prediction complexity of the linear regression is very low. To make a prediction, there is actually a multiplication with the slope which is obtained by training for every feature and the bias b. Therefore, the complexity is just $O(n_{features})$ which is very low. In that case, the average computation time has a ratio of 8×10^{-5} over the difference between the fastest and the second fastest algorithm. Most of the overhead added to make the prediction is, actually, due to the feature computation. The parameter $\beta_{1,2}$ for the Linear regression is 0.0053 which is very close to the time needed to compute the feature. The prediction time is actually negligible compared to the feature computation time.

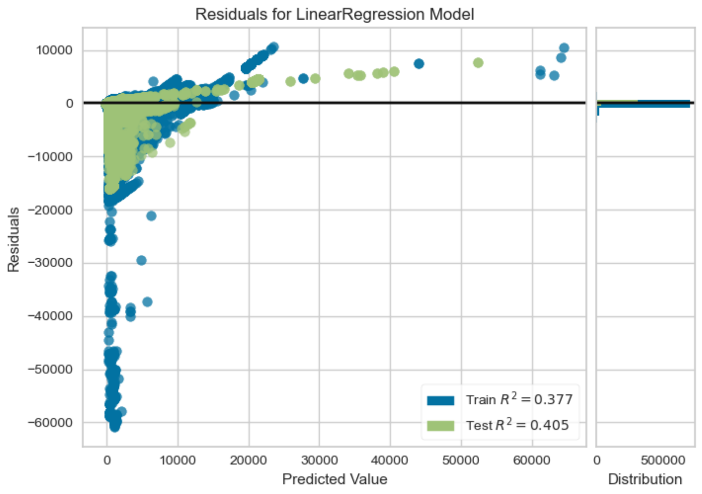


Figure 23: Residual plot when using Ordinary Least Squares regression. The training set used to make the figure is the 7 first folds and the test set is the remaining 3 folds.

6.3.2 KNN

The KNN regressor that was first tested was the default one of **Scikit-Learn**[7] which uses a number of neighbors equal to 5. This KNN regressor obtained an average R^2 score on the test folds equal to 0.0790 and the one obtained on the training set equal to 0.8037, this may be due to an overfitting of the training set by the model. Therefore, multiple value of nearest neighbors were tested in order to find the model which has the best bias and variance trade-off. The variance for the KNN is increasing when the number of nearest neighbors decrease and the bias is decreasing when the number of nearest neighbors increase, which means that the lower the number of neighbors the more complex the model is.

As one can see, Table 2 the average R^2 score on the test folds increases with the number of neighbors which were tested. An higher number of neighbor could actually lead to a better R^2 on test folds. However, one can also see the increase of the parameter $\beta_{1,2}$ with the parameter k which is coherent because the time complexity of KNN prediction when using a tree is $O(k*log(n))$, with n the number of samples in the dataset, which increases with the number of neighbors. Furthermore, the parameter $\gamma_{1,2}$ is actually already at 0.8068 with $k = 50$, this means that 80% on average on the test folds will to win time if the best algorithm is chosen over the second best algorithm. Therefore, no higher k were tested. Moreover, the time complexity also depends on the number of samples in the training set, therefore, if the model is trained with more samples for example when adding the test set to the training, the computation time will increase, thus, $\beta_{1,2}$ will also increase.

One can also see in Table 2 that despite being the model with the lowest R^2, the model KNN with $k = 50$ is not the model with the lowest $MAPE$ on the test set. The KNN regressor with the lowest $MAPE$ is the one with $k = 5$, the $MAPE$ on average on the test fold is 14.0453 for KNN with $k = 5$. Therefore, KNN algorithm does not seem to be well predicting the number of non-zero elements in the solution. Indeed, the KNN with the lowest $MAPE$ makes in average an error which is 14.0453 times its value.

n_neighbors	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
5	0.0790	0.8037	14.0453	4.6097	0.0629	0.7515
8	0.1259	0.7581	14.2764	5.7806	0.0727	0.7653
10	0.1398	0.7369	14.3015	6.2688	0.0724	0.7623
50	0.2835	0.5979	15.1331	9.6880	0.1476	0.8068

Table 2: Results of the KNN regressor using 4 values of nearest neighbors, which are $[1, 2, 5, 10]$.

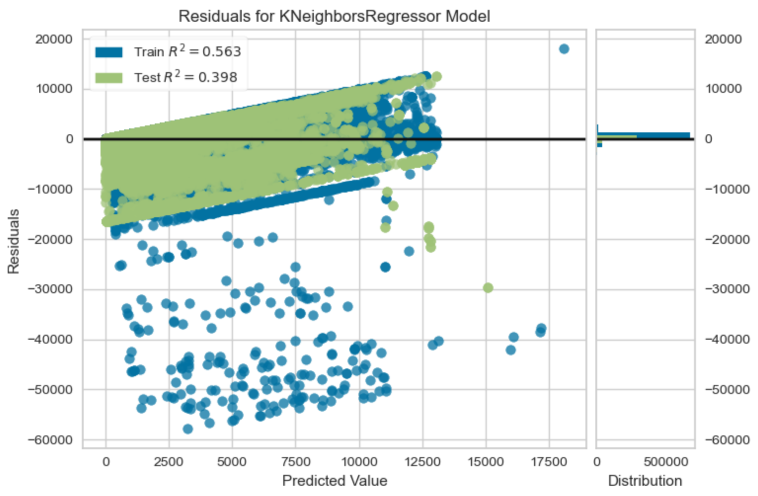


Figure 24: Residual plot when using KNN with $N = 50$. The training set used to make the figure is the 7 first folds that are used in cross validation and the test set the remaining 3 folds.

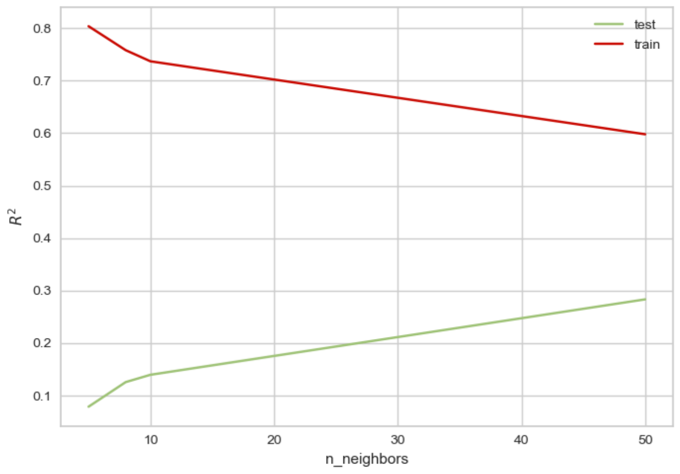


Figure 25: Average R^2 score of the test and training folds with the number of neighbors parameter of the KNN regressor model.

6.3.3 Multilayer Perceptron

The Multi-layer Perceptron regressor was trained using default parameters of *Scikit-learn* but with different hidden layer size. Noted that, the MLPs actually needs to have a scalar in order to put the feature in a range between 0 and 1 which is not the case for the other regressors. This will add 2 operations per features before making the prediction of the MLP. Indeed, Scaler computes the mean and standard deviation for each feature and store it. Then, when making a prediction for each feature, the value that will be given to the MLP is computed with

$$z = (x - \mu)\sigma$$

where z is the scaled feature, x is the feature not scaled, μ is the mean and σ is the standard deviation.

Different hidden layer size for the MLP were tested and are shown in Table 3 to find the best complexity of the model. Even if the lowest average R^2 score on the training folds is equal to 0.74, the highest average R^2 on test folds of the MLP regressor is obtained with an hidden layer size of 10 with a value of -1.3756×10^{14} which is lower than the one obtained with linear regression which have a R^2 on test set equal to -1.644×10^{13}. When computing the R^2 score obtained on each fold when they are used as test fold, as the linear regression method, the $R2$ score obtained on the fold 3 is much lower than the other with a value equal to -1.3756×10^{15} whereas the average $R2$ score when removing the score on the folds 3 is 0.604 which is higher than this score obtained by linear regression. The reason why the average R^2 is lower when using MLP than when using the linear regression is because MLP fits less well the fold 3 is unknown. The Figure 26 shows the residual of the MLP with an hidden layer size equal to 10 and one can see that the spike error of the MLP for systems that should have a very low number of non-zero elements the model predict actually a very high value.

hidden_layer_sizes	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
10	-1.3756×10^{14}	0.7413	1.0373×10^6	16.9972	0.0005	0.0223
50	-2.2330×10^{14}	0.7931	1.3212×10^6	15.3506	0.0015	0.1050
100	-2.0554×10^{14}	0.8141	1.2678×10^6	17.0552	0.0036	0.2094
200	-5.1828×10^{14}	0.8302	2.0133×10^6	17.8585	0.0050	0.2298
500	-3.43257×10^{14}	0.8433	1.6384×10^6	19.0117	0.0134	0.3528

Table 3: Results of the MLP regressor different hidden layer sizes.

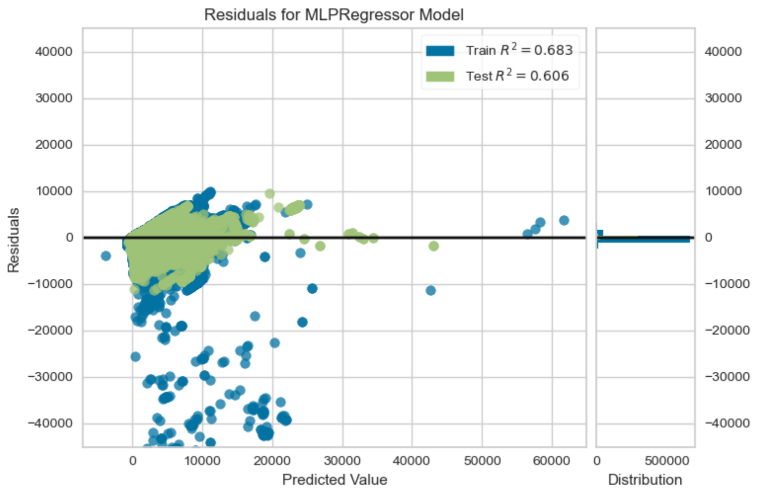


Figure 26: Residual plot when using MLP with an hidden layer size equal to 10. The training set used to make the figure is composed of the 7 first folds and the test set is the remaining 3 folds.

6.3.4 Decision Tree

One can see in Table 4 the results of Decision Tree trained with different maximal depths. The decision tree achieved better R^2 score than previous models with a maximal average R^2 score on the test folds equal to 0.37 which is obtained with a depth of 10. However, the lowest $MAPE$ is obtained with a depth of 20 with a value equal to 0.6339. For higher depths, the $MAPE$ increases which can be due to overfitting. Therefore, the Decision Tree that is supposed best for the regression of the number of non-zero elements in the solution is the one with a depth of 20. The residual plot of this model is shown in Figure 27. That plot shows that even if the average $MAPE$ in test folds is equal to 0.6339, there is still cases in the model that can lead to high error. In the Table 4, it is shown that parameters $\beta_{1,2}$ and $\gamma_{1,2}$ increase with the depth of the decision tree with a maximal value of 0.0007 for $\beta_{1,2}$ and 0.0335 for $\gamma_{1,2}$. These value have a great ratio performance per computation time, the KNN regressor with $N = 50$ obtained an average R^2 on the test folds equal to 0.2835 which is lower than the one of the Decision Tree and this KNN have parameter $\gamma_{1,2} = 0.8068$ which is much greater than the one of the DT. The reason why the parameters $\beta_{1,2}$ and $\gamma_{1,2}$ increases with the depth of the tree i the because the time complexity of the prediction of a decision tree is proportional to the depth of the tree, the complexity is $O(D)$ with D being the depth.

max_depth	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
10	0.3700	0.9505	0.6865	0.5420	0.0003	0.0131
20	0.3410	0.9986	0.6339	0.1710	0.0006	0.0263
30	0.3097	1.0000	0.6849	0.0219	0.0006	0.0281
40	0.3199	1.0000	0.6950	0.0006	0.0007	0.0328
50	0.3034	1.0000	0.6740	0.0000	0.0007	0.0328
60	0.3218	1.0000	0.6776	0.0000	0.0007	0.0317
70	0.3218	1.0000	0.6777	0.0000	0.0007	0.0328
80	0.3218	1.0000	0.6777	0.0000	0.0007	0.0335

Table 4: Results of Decision tree regressors using 8 values of maximum depth. In this table the test means that the metrics is taken by averaging the value obtained on the test folds.

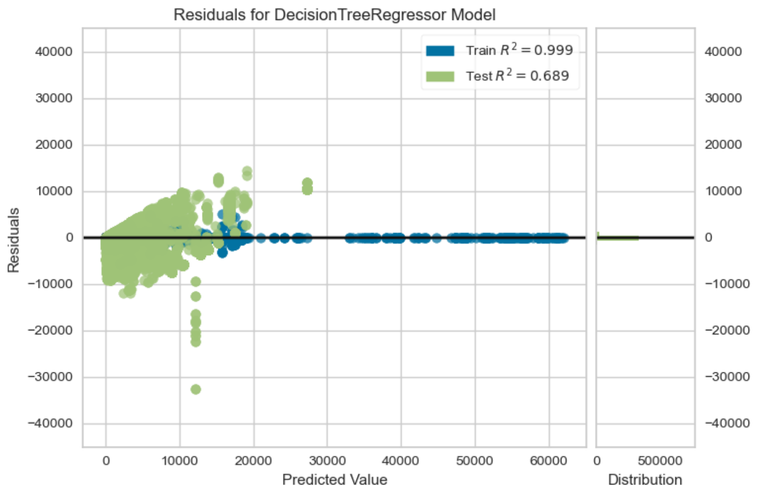


Figure 27: Residual plot when using the supposed best Decision Tree, which is the one with maximal depth of 20. The training set used to make the figure is the combination of 7 folds and the test set is 3 remaining ones.

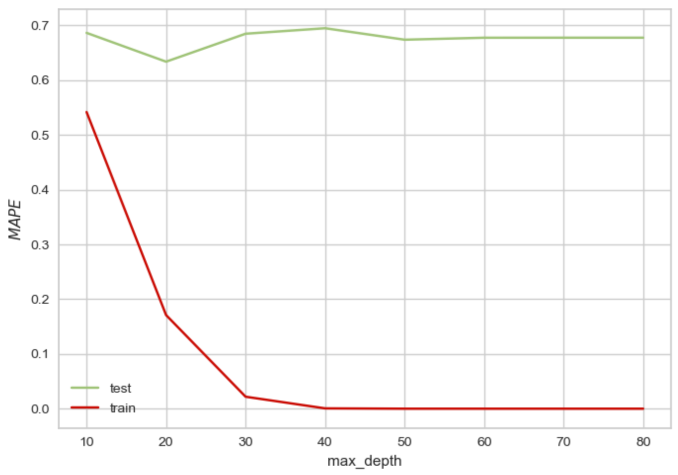


Figure 28: $MAPE$ of the test being 3 folds and training the 7 remaining folds with maximum depth of the Decision Tree regressor model.

6.3.5 Random Forest

The Table 5 shows the results of the Random Forest regressor with different number of features used to make the split. This table shows that one of the Random Forest regressor that obtained the lowest average $MAPE$ on the test folds is the one using all available features to make the split, which is actually the parameter suggested in *Sklearn*. Therefore, the Random Forest with this parameter is the same as doing Bagging with decision trees. Indeed, the difference between Random Forest and Bagging with trees is that the random forest introduces randomness by using a number of feature to select. In that case all the feature are used, so there is no randomness added. Even if the average $MAPE$ on test folds is one of the lowest, the average R^2 seem to be the highest when using 2 features.

The Random Forest using all the features to make the split was used to test another parameter, which is the number of decision trees used to make the prediction. The result that the metric $\beta_{1,2}$ increases with the number of tree used is what is expected because the complexity of the prediction made by a Random Forest is $O(N \times D)$, with N the number of trees and D the maximum depth of the trees. As one can see in Table 6, the higher the number of trees the lower the $MAPE$ and the higher the R^2. However, the $MAPE$ and R^2 reach a plateau very fast, the performance increase but this cost more and more computation time.

k	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
1	0.7278	0.9946	3.0284	0.2017	0.0940	0.7794
2	0.7452	0.9958	1.2111	0.1054	0.0740	0.7637
3	0.7118	0.9964	0.8662	0.0804	0.0552	0.7389
4	0.6966	0.9967	0.6804	0.0734	0.0549	0.7379
5	0.6952	0.9968	0.6453	0.0708	0.0552	0.7396
6	0.6265	0.9969	0.6413	0.0696	0.0494	0.7253
7	0.5872	0.9969	0.6431	0.0687	0.0493	0.7241
8	0.4742	0.9968	0.6415	0.0683	0.0465	0.7150

Table 5: Results of the Random Forest regressor different number of feature used to make the split. These value are called k in the table.

n_estimators	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
10	0.4992	0.9955	0.6625	0.0724	0.0053	0.2342
20	0.4887	0.9960	0.6464	0.0703	0.0123	0.3401
50	0.4802	0.9966	0.6455	0.0688	0.0240	0.4824
100	0.4742	0.9968	0.6415	0.0683	0.0474	0.7181
200	0.4664	0.9969	0.6386	0.0679	0.1107	0.7884

Table 6: Results of the Random Forest regressor with different number of estimators.

In the same table, the parameters $\gamma_{1,2}$ is very high the lowest one being the Random Forest with a number of estimators equal to 10 that already have a value around 0.23. This means that around 23% of the samples will not win time when choosing the best algorithm over the second one. Moreover, three regressions will also come after this one to predict the computation time of each algorithm, thus, Random Forest with lower number of estimators and with a limited depth of tree where tested to find one with similar performance with a lower prediction time. One can see in Table 7 that Random Forest with 2 estimators and with a depth of 20 can compete with Decision Tree with a depth of 20 that had an average $MAPE$ on the test folds of 0.6339 whereas Random Forest have the lowest $MAPE$ obtained so for with a value equal to 0.6190 but with a gamma which is approximately two times the one of the Decision Tree.

n_estimators	max_depth	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
2	10	0.5203	0.9504	0.6995	0.5362	0.0005	0.0226
	20	0.5041	0.9877	0.6190	0.1895	0.0010	0.0502
	30	0.5123	0.9882	0.6247	0.0878	0.0013	0.0767
	50	0.4844	0.9883	0.6532	0.0778	0.0012	0.0803

Table 7: Results of the Random Forest regressor with 2 estimators and different maximal depth of trees.

The Figure 29 shows the residual of the Random Forest with the lowest $MAPE$ obtained which is the one with the parameter $k = 8$, $n\_estimators = 2$ and $max\_depth = 20$. The Random forest makes more error on the training set than the Decision Tree, the Decision Tree residuals points were very close to the $x - axis$. This can be due to the fact that the Decision Tree with a depth of 20 may be overfitting the training set compared to the Random Forest model.

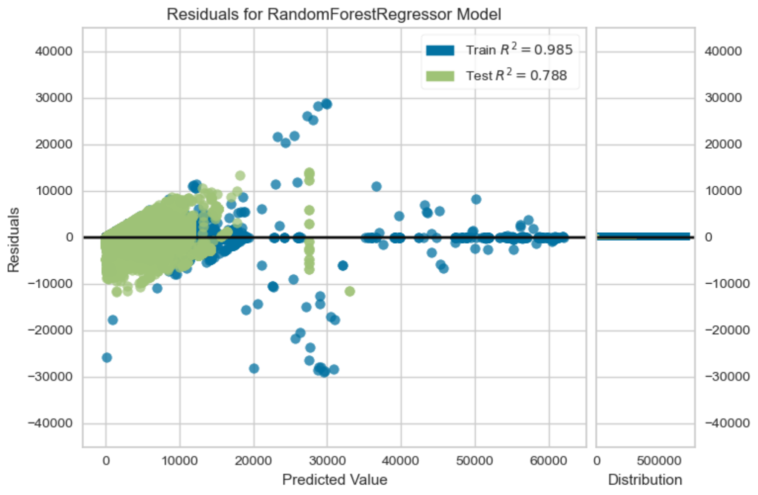


Figure 29: Residual plot when using Random Forest with $n\_estimators = 200$ and $k = 8$. The training set used to make the figure is the combination of the 7 first folds and the test set is the remaining 3 folds.

6.3.6 Extremely Randomized Trees

Extremely Randomized Trees have similar parameters with the Random Forest such as the number of features used to make the split. The Table 8 shows the metrics obtained with different value of this parameter. One can see that similarly to the Random Forest, the Extremely Randomized Trees that has the lowest average $MAPE$ score on the test folds is the one with the number of feature used in the split equal to the number of feature in the dataset. However, the R^2 test is not the maximum with $k = 8$. The maximal R^2 test is obtained with $k = 4$.

A table with the result of Extremely Randomized Trees with $k = 8$ and using different number of estimators was made. As shown in the Table 9, similarly to Random Forest, the higher the number of estimators the higher the R^2 score and the lower the $MAPE$ test but at the cost of computation time, that is why higher number of estimators where not tested. Indeed, the Extremely Randomized Trees with a number of estimators equal to 100 already have a $\gamma_{1,2}$ equal to 0.7662 and because Extremely randomized trees prediction has a complexity equal to $O(N \times D)$. Therefore, higher number of estimators would necessarily lead to higher $\gamma_{1,2}$.

With the parameters tested, the Extremely Randomized Trees that obtained the lowest average $MAPE$ on the test folds with the default parameter of *Scikit-learn*, this means with $n\_estimators = 100$ and $k = 8$. The $MAPE$ test of this regressor was 0.6241 which is slightly higher than the one of the Random Forest with the lowest $MAPE$ which had a value of 0.6190. However, the Extremely Randomized Trees obtained a much higher average R^2 on the test folds with a value of 0.7328 whereas the Random Forest obtained 0.5041.

k	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
1	0.7220	1.0000	3.5532	0.0000	0.0989	0.7839
2	0.7454	1.0000	1.8964	0.0000	0.0780	0.7688
3	0.7512	1.0000	1.1991	0.0000	0.0969	0.7819
4	0.7661	1.0000	0.7949	0.0000	0.0764	0.7663
5	0.7658	1.0000	0.7713	0.0000	0.0812	0.7696
6	0.7495	1.0000	0.6739	0.0000	0.0683	0.7575
7	0.7461	1.0000	0.6645	0.0000	0.1194	0.7764
8	0.7328	1.0000	0.6241	0.0000	0.0866	0.7703

Table 8: Results of the Extremely Randomized Trees trained using different value of k, which is the maximum number of feature used in a split.

n_estimators	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
5	0.6342	1.0000	0.6716	0.0000	0.0035	0.2095
10	0.6514	1.0000	0.6534	0.0000	0.0072	0.2593
20	0.7076	1.0000	0.6296	0.0000	0.0142	0.3619
30	0.7138	1.0000	0.6278	0.0000	0.0229	0.4571
50	0.7229	1.0000	0.6232	0.0000	0.0377	0.6659
100	0.7328	1.0000	0.6241	0.0000	0.0714	0.7662

Table 9: Results of the Extremely Randomized Trees using different number of trees generated.

Similarly to Random Forest, without pruning the trees or reducing the number of estimators, the value $\gamma_{1,2}$ of the Extremely Randomized Trees with the lowest average $MAPE$ on test folds is quiet high. Therefore, lowest value of the parameter *n_estimators* and *max_depth* where tested. The Table 10 shows with 2 to 3 number of estimators and a limited depth to 50, the Extremely Randomized Trees seem to need an higher value for the depth to achieve similar average $MAPE$ on test folds than Random Forest. Indeed, the Extremely Randomized Trees with a number of estimators of 2 with a maximal depth of 50 obtained an average $MAPE$ on test folds equal to 0.6445 whereas Random Forest with the same number of estimators had 0.6190 but with a maximal depth of 20. Increasing, the number of estimators to 3 did increases the R^2 score but it did not decrease the $MAPE$. Furthermore, the computation time is higher, thus, it increases $\gamma_{1,2}$.

n_estimators	max_depth	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
2	10	0.5336	0.7790	12.6068	12.1897	0.0004	0.0182
	20	0.5200	0.9631	1.2097	1.0531	0.0007	0.0315
	30	0.4806	0.9971	0.7319	0.2322	0.0012	0.0885
	50	0.5486	1.0000	0.6445	0.0053	0.0014	0.1055
3	10	0.6328	0.8034	10.6884	10.3683	0.0005	0.0226
	20	0.6329	0.9698	1.0952	0.9507	0.0010	0.0659
	30	0.4507	0.9981	0.6947	0.2393	0.0018	0.1436
	50	0.6242	1.0000	0.6562	0.0060	0.0023	0.1718

Table 10: Results of the Extremely Randomized Trees using different number of trees generated.

One can see in Figure 30 that the residual of the test set does not seem to be different than the one of the Random Forest. However, the residual on the training set is close to zero. Therefore, there is not much error on the training set compared to the one on the test set as the

R^2 score on the two sets suggests. This may be due to an overfit of the Extremely Randomized Trees compared to Random Forest that leads to an higher error on the test set.

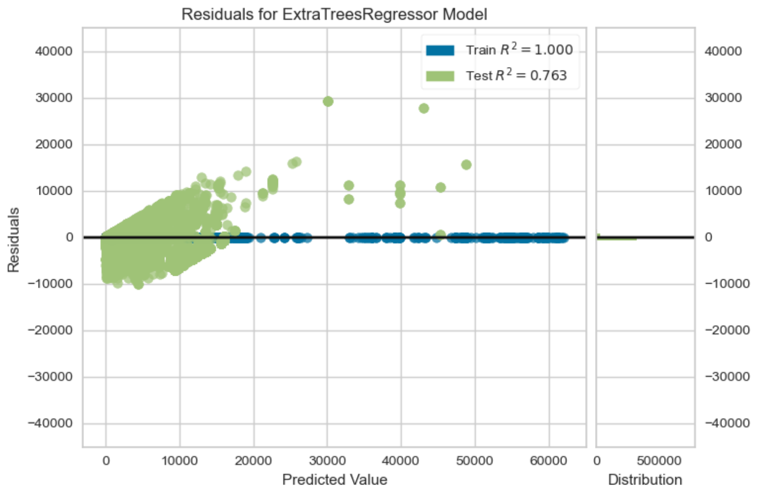


Figure 30: Residual plot when using Extremely Randomized Trees with $k = 8$, the number of estimators equal to 2 and a maximal depth of 50.

6.3.7 Adaboost

The Table 11 shows the results of the regressor Adaboost when finetuning the parameter *max_depth* of the decision tree that is used as based estimator, every other parameters are kept with the default values of *Scikit-learn*. The Adaboost regressor which obtained the lowest $MAPE$ is the one with a *max_depth* equal to 50 with a value of 0.4265. This $MAPE$ is actually the lowest obtained so far.

The Table 12 shows Adaboost performance with the parameter *max_depth* equal to 50 but with different number of estimators. The parameter that have the lowest $MAPE$ on the test set is the one with the number of estimator equal to 50, which is, actually, the default value. However, the R^2 score on the test may still increase when the number of estimators increases. The highest average R^2 on test folds obtained so far is also with a number of estimators equal to 50, it has value of 0.7056 whereas the Extremely Random Trees with $k = 4$ and with all other parameters kept with default obtained 0.7661.

max_depth	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
10	0.5414	0.9842	0.9305	0.7470	0.0181	0.3852
20	0.6804	0.9997	0.5170	0.1627	0.0282	0.5711
50	0.7056	1.0000	0.4265	0.0036	0.0345	0.6473
70	0.5821	1.0000	0.4429	0.0038	0.0317	0.6241
80	0.5821	1.0000	0.4429	0.0038	0.0321	0.6264

Table 11: Results of the ADABoost regressor with different value of the *max_depth* parameter of the decision tree used as a base estimator.

n_estimators	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
5	0.5913	0.9997	0.5088	0.0177	0.0047	0.2255
10	0.6504	0.9999	0.4868	0.0100	0.0075	0.2700
20	0.6668	1.0000	0.4627	0.0050	0.0135	0.3546
50	0.7056	1.0000	0.4265	0.0036	0.0323	0.6289

Table 12: Results of the ADABoost regressor using a number of estimator equal to 5, 10, 20 and 50.

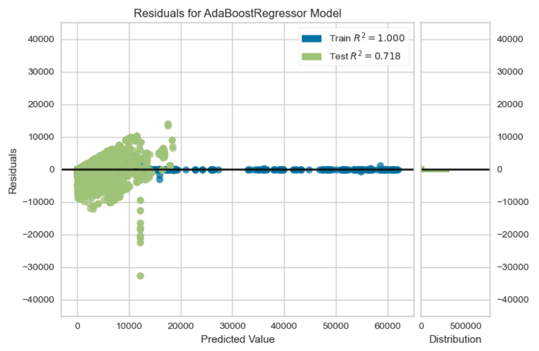
The parameter $\gamma_{1,2}$ for Adaboost with *max_depth* equal to 50 and a number of estimator equal to 50 is quite high with a value of 0.6289, therefore, if this estimator is used an average 62.89% of the samples will not have time benefit when choosing the best algorithm instead of the second. In order to have Adaboost with lower $\gamma_{1,2}$, lower number of estimators and maximal depths were tested. One can see in Table 12 that Adaboost does not perform better than random forest and Extremely Randomized Trees if one does not accept a $\gamma_{1,2}$ greater than 0.11. Extremely Randomized Trees optained an average $MAPE$ on test folds equal to 0.6445 for a $\gamma_{1,2} = 0.1055$ and Random Forest obtained a $MAPE$ of 0.6190 for a $\gamma_{1,2} = 0.502$. However, if more time can be accepted, Adaboost manage to obtain a $MAPE$ equal to 0.5088 with $\gamma_{1,2} = 0.2255$.

n_estimators	max_depth	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
2	10	0.2716	0.9415	0.6966	0.5497	0.0009	0.0476
	20	0.1651	0.9800	0.6759	0.1925	0.0013	0.0804
	30	0.3993	0.9819	0.6831	0.0928	0.0014	0.1076
	50	0.3820	0.9812	0.6937	0.0800	0.0014	0.1035
3	10	0.6227	0.9654	0.7025	0.5624	0.0013	0.0932
	20	0.4788	0.9980	0.5351	0.1680	0.0020	0.1524
	30	0.5744	0.9989	0.5406	0.0438	0.0021	0.1659
	50	0.4738	0.9989	0.5422	0.0297	0.0025	0.1723
5	10	0.6831	0.9714	0.7283	0.5839	0.0023	0.1575
	20	0.6317	0.9988	0.5142	0.1642	0.0032	0.2011
	30	0.6378	0.9997	0.5139	0.0334	0.0037	0.2079
	50	0.5913	0.9997	0.5088	0.0177	0.0047	0.2255

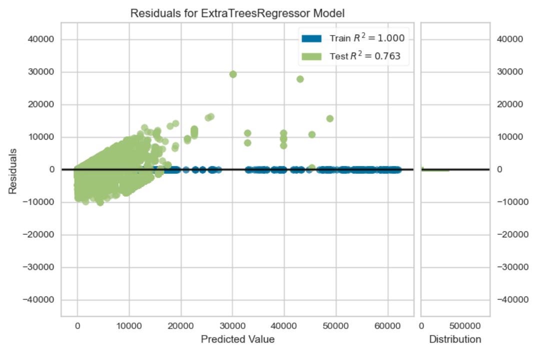
Table 13: Results of the ADABoost regressor with different value of the *max_depth* parameter of the decision tree used as a base estimator.

Another advantage that Adaboost does not have compared to Extremely Random Trees and Random Forest is that it cannot be parallelized but the two other can. Indeed, the estimators of Adaboost needs the results of the previous estimator to make its prediction whereas for Extremely Random Trees and Random Forest the estimators can run in parallel and, then, the results of the estimators can be averaged.

Adaboost is known to be better on difficult cases. Indeed, it will train a tree one after the other and putting more weights on samples which are less well predicted on previous tree. One can see in the Figure 31 that the residual for some prediction seem to be reduced compared to the one of the Extremely Randomized Tree.



(a) Residual plot of Adaboost regressor with the parameter *n_estimator* equal to 5 and the parameter *max_depth* equal to 50.



(b) Residual plot of Extremely Randomized Trees with k equal to 8, the number of estimators equal to 2 and a max depth of 50.

Figure 31: Residual plot of the supposed best Extremely Randomized Tree model and the supposed best and fast Adaboost model with the test set being the 3 last folds and training set being the remaining 7 folds.

7 Model selection

In section Proxy model selection, some regression models were trained to predict the number of non-zeros elements in the solution. In order to avoid testing every model that were trained, the one with the best performance in cross validation will be used. The first model that will be tested is Linear Regression even if its prediction of the proxy were not performing well compared to other models.This model is kept because it is very fast. The models which predict well the number of non-zero elements in the solution are Decision Tree with a depth of 50, Random Forest with a number of estimator of 2 using all the features and with a maximal depth a 20 and the last proxy model that is used is AdaBoost with 5 estimators and with a maximal depth of 20. The output of these models will be used as a feature for three other models, these 3 models will predict the time of taken by its assigned algorithm. This section is to try to find the best combination of proxy model and models predicting the solve computation time, and to finally try which combination predicts best the fastest algorithm.

However, some regression models used in previous section will not be used in this section to predict the computation time. Indeed, when KNN was tested in previous section, it had a parameter $\gamma_{1,2} = 0.7515$, adding 3 more KNN to predict the time of the 3 algorithms will necessarily lead to higher $\gamma_{1,2}$, which means that even less than 30% of the systems will save time if the right choice of the algorithm is made instead of the second best choice. The MLP was also not predicting well the number of non-zero elements in the solution, therefore, it was not further tested as proxy. The Extremely Randomized Trees obtained reasonable performance compared to other models but it needed more time to have similar performance as Random Forest, thus, this proxy predictor is not used in this section.

7.1 Linear Regression

The Table 14 shows the score of the Linear Regression when predicting the 3 algorithm solve times using a Linear regression to predict the proxy. In this table, one can see that the average R^2 score of the test folds is negative and very low indicating that this model is very bad at predicting the computation time.

Algorithm	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
General	-5.1404$\times10^7$	0.8690	1.4129$\times10^3$	0.2074	0.0003	0.0114
Two-Phase	-3.3304$\times10^{12}$	0.7809	3.0865$\times10^7$	2.7769$\times10^3$	0.0002	0.0115
One-Phase	-3.1918$\times10^{12}$	0.8178	2.1119$\times10^5$	48.5596	0.0002	0.0104

Table 14: Results of the Linear Regression to predict the time taken by each algorithm using also a **Linear regression** to predict the number of non-zero elements in the solution.

The Table 15 shows the same model but with another model to predict the proxy. In this table, the proxy is predicted with a Decision Tree with a maximal depth of 50. Changing the model predicting the proxy to the Decision Tree one does not seem to help the prediction of the algorithm solve times. The Table 16 and 17 shows the score of the models when using the 2 other proxy regressors. In these tables, one can see that the R^2 score does not seem to increase significantly. The highest average R^2 score of the test folds of the *Two-Phase algorithm* and *One-Phase algorithm* time regression is obtained when using the Random forest regressor to predict the proxy.

Algorithm	R^2 test	R^2train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
General	-4.2378$\times10^8$	0.8690	4.0564$\times10^3$	0.2044	0.0007	0.0262
Two-Phase	-9.1182$\times10^{11}$	0.8027	1.6154$\times10^7$	5.8134$\times10^3$	0.0007	0.0262
One-Phase	-5.7534$\times10^{12}$	0.8189	2.8354$\times10^5$	63.1362	0.0006	0.0262

Table 15: Results of the Linear Regression to predict the time taken by each algorithm using a **Decision Tree** with a maximal depth of 50 to predict the number of non-zero elements in the solution.

Algorithm	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
General	-4.2528$\times10^8$	0.8690	4.0635$\times10^3$	0.2046	0.0013	0.0878
Two-Phase	-8.5940$\times10^{11}$	0.8017	1.5683 $\times10^7$	5.7631 $\times10^3$	0.0012	0.0812
One-Phase	-5.7170$\times10^{12}$	0.8188	2.8264 $\times10^5$	62.7912	0.0013	0.0874

Table 16: Results of the Linear Regression to predict the time taken by each algorithm using **Random Forest** with two decision tree with a maximal depth of 20 to predict the number of non-zero elements in the solution.

Algorithm	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
General	-4.2329$\times10^8$	0.8690	4.0540$\times10^3$	0.2044	0.0042	0.2200
Two-Phase	-9.2075$\times10^{11}$	0.8027	1.6233$\times10^7$	5.7966$\times10^3$	0.0042	0.2194
One-Phase	-5.7467$\times10^{12}$	0.8189	2.8338$\times10^5$	62.9924	0.0042	0.2199

Table 17: Results of the Linear Regression to predict the time taken by each algorithm using **AdaBoost** using 3 Decision trees with a maximal depth of 20 to predict the number of non-zero elements in the solution.

However, when predicting the time of the algorithm in Section First model with the number on non-zero elements in the solution, the R^2 score were not significant either but that model was still predicting well which algorithm was the fastest. Therefore, one model using Linear regression were used to try predicting the fastest algorithm and this even if their regressions were bad. The Table shows that the predicted algorithm does not have better results than the model **LRNNZB**

(b) which obtained an average computation time of the test folds of $27.338s$ and a **balanced-accuracy** of 0.459. The new model $LRRFProxy$ uses only the size of b as feature to predict the *General algorithm* solve time, whereas, the two other algorithm solve time are predicted using the same features used for the proxy and it also uses the proxy itself as feature. This model does not have lowest average time than **LRNNZB (b)**, it took $36.4851s$ for **LRRFProxy** and $27.3383s$ for **LRNNZB (b)**. This time includes the solve time, the time to compute the feature, the proxy prediction and the 3 algorithm solve time regressions.

Model	Average time (s)	Balanced accuracy	$\beta_{1,2}$	$\gamma_{1,2}$	$\gamma_{1,3}$
$LRRFProxy$	36.4851	0.4668	0.0010	0.0568	0.0000
$LRNNZB(b)$	**27.3383**	0.4589	0.0001	0.0053	0.0000

Table 18: Classifier comparison between $LRNNZB(b)$ and $LRRFProxy$. The average time is average of the total time needed to solve the test fold. The metrics average time, $\beta_{1,2}$, $\gamma_{1,2}$ and $\gamma_{1,3}$ includes all the time necessary to predict the algorithm.

7.2 Decision Tree

Similarly to other Linear Regression, Decision Tree models were tested with different models that predicts the number of non-zero element in the solution. As in the proxy model selection when using Decision Tree, different maximal depth of the Decision Tree were tested in order to find the best variance/bias trade-off.

The Table 19 shows the scores of the regression of the algorithm solve time when using a Linear Regression to predict the number of non-zero elements in the solution. The R^2 test score are not as bad as when using Linear Regression to predict the time, even if they are still negative. One can also see that the average $MAPE$ test for the prediction of the General Algorithm solve time is not bad. However, a simple Linear Regression with only the size of b achieve a better R^2 score as show in the Section First model. The best $MAPE$ test for the regression of the *Two-Phase* algorithm is obtained with a depth of 30, with a R^2 test score of -2.1930 and a $MAPE$ test of 9.1725 whereas the depth with the lowest average $MAPE$ over the test folds for the regression of the *One-Phase* algorithm is obtained with a maximal depth equal to 20 with a R^2 test score of -1.6358 and $MAPE$ test of 2.1068.

Algorithm	max_depth	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
General	10	-0.3626	0.9060	**0.1632**	0.0912	0.0003	0.0131
	20	-0.5545	0.9795	0.2104	0.0463	0.0006	0.0270
	30	-0.6511	0.9964	0.2428	0.0103	0.0009	0.0461
	40	-0.5640	0.9980	0.2325	0.0028	0.0007	0.0315
	50	-0.5737	0.9981	0.2331	0.0024	0.0008	0.0407
	80	-0.6553	0.9981	0.2391	0.0023	0.0008	0.0334
Two-Phase	10	-1.3976	0.9614	14.5672	11.2169	0.0003	0.0120
	20	-2.3100	0.9986	13.5959	1.6076	0.0005	0.0225
	30	-2.1930	1.0000	**9.1725**	0.4070	0.0009	0.0492
	40	-2.2787	1.0000	12.9155	0.1103	0.0009	0.0484
	50	-2.2446	1.0000	10.2121	0.0268	0.0010	0.0641
	80	-2.2192	1.0000	10.1793	0.0076	0.0010	0.0537
One-Phase	10	-1.3791	0.9537	7.2406	5.5790	0.0003	0.0120
	20	-1.6358	0.9945	**2.1068**	0.6573	0.0005	0.0214
	30	-1.7296	1.0000	2.2554	0.1472	0.0010	0.0570
	40	-1.6968	1.0000	2.2750	0.0236	0.0009	0.0435
	50	-1.7088	1.0000	2.3909	0.0063	0.0010	0.0549
	80	-1.6538	1.0000	2.2095	0.0043	0.0010	0.0534

Table 19: Results of the Decision Tree to predict the time taken by each algorithm and using a **Linear Regression** to predict the proxy.

One can see on Table 20 that using Decision Tree instead of a Linear Regression to predict the number of non-zero elements in the solution increases the average R^2 score of the test folds. Indeed, for a maximal depth of 10 the *General algorithm* time regression has a positive R^2 test score with a value of 0.3208, the *Two-Phase algorithm* time regression with a depth of 50 has a similar $MAPE$ to the one with a depth of 20 and because the R^2 is higher for a depth of 20, the selected depth is 20. This Decision Tree does not have a positive R^2 test score but it has the highest one seen so far with a value of -0.6010. The lowest $MAPE$ of the *One-Phase algorithm* time regression obtained in the table is with a Decision Tree with a maximal depth equal to 20. However, the R^2 test score of the Decision Tree with a maximal depth of 10 was higher for the *One-Phase algorithm.*

The Table 21 shows the score of the Decision Tree when using the supposed best Random Forest to predict the number of non-zero elements in the solution. In that table, one can see that the average $MAPE$ of the test folds is slightly lower than the model that uses a Decision Tree to predict the proxy. The best $MAPE$ for the regression are obtained in the same maximal depth of the Decision Tree. Even if the $MAPE$ are lower for the 3 algorithm solve time regressions, the R^2 test score decreases.

Algorithm	max_depth	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
General	10	0.3208	0.9050	**0.1309**	0.0911	0.0009	0.0527
	20	0.2873	0.9820	0.1770	0.0463	0.0011	0.0746
	30	-0.0539	0.9977	0.2031	0.0100	0.0015	0.1114
	40	-0.0301	0.9981	0.1991	0.0027	0.0015	0.1027
	50	0.2041	0.9981	0.1940	0.0023	0.0014	0.1029
	80	0.0200	0.9981	0.1949	0.0023	0.0013	0.0935
Two-Phase	10	-0.8207	0.9762	38.9513	34.0940	0.0008	0.0516
	20	-0.6010	0.9998	**9.0966**	0.7849	0.0010	0.0600
	30	-0.7755	1.0000	9.2831	0.4561	0.0014	0.1064
	40	-0.6633	1.0000	9.2489	0.1847	0.0016	0.1266
	50	-0.6558	1.0000	9.0493	0.0878	0.0019	0.1513
	80	-0.8338	1.0000	9.3362	0.0132	0.0017	0.1352
One-Phase	10	-0.6373	0.9573	2.3987	0.6951	0.0009	0.0621
	20	-0.9388	0.9983	**2.0195**	0.2185	0.0011	0.0712
	30	-0.9499	1.0000	2.2590	0.0833	0.0015	0.1074
	40	-1.2115	1.0000	2.2959	0.0203	0.0018	0.1385
	50	-0.9566	1.0000	2.3071	0.0062	0.0019	0.1482
	80	-1.1673	1.0000	2.2699	0.0043	0.0016	0.1268

Table 20: Results of the Decision Tree to predict the time taken by each algorithm and using a **Decision Tree** to predict the proxy.

Algorithm	max_depth	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
General	10	0.3340	0.9052	0.1301	0.0911	0.0010	0.0602
	20	-0.0045	0.9802	0.1719	0.0464	**0.0013**	0.0764
	30	0.2719	0.9970	0.1899	0.0100	0.0014	0.1124
	40	0.0072	0.9981	0.1991	0.0027	0.0015	0.1197
	50	-0.0529	0.9981	0.2117	0.0023	0.0015	0.1159
	80	-0.0840	0.9981	0.2048	0.0023	0.0015	0.1159
Two-Phase	10	-4.7540	0.9755	34.5469	29.9838	0.0010	0.0502
	20	-11.7417	0.9998	**7.8288**	0.9978	0.0012	0.0764
	30	-11.7208	1.0000	8.2587	0.5074	0.0014	0.1041
	40	-11.7824	1.0000	8.7648	0.1708	0.0016	0.1422
	50	-11.7368	1.0000	8.9521	0.0625	0.0018	0.1479
	80	-11.4859	1.0000	9.0313	0.0089	0.0018	0.1499
One-Phase	10	-1.3475	0.9573	1.8059	0.8704	0.0010	0.0568
	20	-2.0446	0.9982	**1.5749**	0.3620	0.0012	0.0764
	30	-1.4165	1.0000	1.7398	0.1220	0.0015	0.1159
	40	-1.3428	1.0000	1.7716	0.0252	0.0017	0.1422
	50	-1.5752	1.0000	1.6450	0.0070	0.0018	0.1479
	80	-1.3865	1.0000	1.6767	0.0043	0.0018	0.1479

Table 21: Results of the Decision Tree to predict the time taken by each algorithm and using **Random Forest** with a 2 estimator and a maximal depth of 20 to predict the proxy.

The Table 22 shows the results of the Decision Trees predicting the algorithm solve times when using AdaBoost to predict the number of non-zero elements in the solution. When it concerns the prediction of the General algorithm times, the average R^2 score and $MAPE$ of the test folds are very similar to the one using the Random Forest proxy regressor. However, the minimal mean $MAPE$ over the test folds of the *Two-Phase algorithm* is lower with a value of 5.7097, this value is obtained with a Decision Tree with a maximal depth of 50. The best $MAPE$ that was obtained previously was 7.8288. Even if the regression of the *Two-Phase algorithm* seems better when using AdaBoost to predict the proxy rather than using Random Forest, the regression of the *One-Phase algorithm* seem worse. Indeed, the minimal average $MAPE$ over the test set was 1.5749 when using Random forest to predict the proxy and the one when using AdaBoost obtained a value of 2.0576.

Algorithm	max_depth	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
General	10	0.3349	0.9050	**0.1300**	0.0912	0.0028	0.1940
	20	-0.0028	0.9794	0.1754	0.0463	0.0030	0.2005
	30	-0.0447	0.9962	0.2087	0.0100	0.0032	0.2034
	40	-0.0865	0.9981	0.1951	0.0027	0.0032	0.2036
	50	-0.0997	0.9981	0.2091	0.0023	0.0033	0.2053
	80	0.1961	0.9981	0.2014	0.0023	0.0037	0.2115
Two-Phase	10	-0.1561	0.9766	37.5835	35.8641	0.0028	0.1935
	20	-0.1467	0.9998	5.9658	0.9050	0.0029	0.1978
	30	-0.2395	1.0000	5.7619	0.5246	0.0032	0.2021
	40	-0.1324	1.0000	5.8583	0.1813	0.0034	0.2069
	50	-0.2638	1.0000	**5.7097**	0.0657	0.0036	0.2096
	80	-0.1230	1.0000	6.2848	0.0090	0.0040	0.2168
One-Phase	10	-0.9918	0.9554	2.1454	0.8186	0.0028	0.1940
	20	-1.1063	0.9968	**2.0576**	0.3410	0.0030	0.1983
	30	-1.9912	1.0000	2.1544	0.1108	0.0033	0.2043
	40	-1.0740	1.0000	2.1992	0.0221	0.0035	0.2074
	50	-1.1160	1.0000	2.2053	0.0064	0.0035	0.2093
	80	-1.1450	1.0000	2.2037	0.0043	0.0040	0.2156

Table 22: Results of the Decision Tree to predict the time taken by each algorithm and using **AdaBoost** with a 5 estimator and a maximal depth of 20 to predict the proxy.

Four classifier models using a Decision Tree to predict the time taken by the algorithms were tested and compared against **LRNNZB (b)**. Each of these models uses as **LRNNZB(b)** a Linear Regression which only uses the size of b as feature to predict the *General algorithm* solve time. One of these classifier will be called **DTRFproxy** using a Random Forest to predict the proxy, the maximum depth of tree to predict the *Two-Phase algorithm* and *One-Phase algorithm* is 20. The second model tested that will be called **DTLRProxy** is using the Linear Regression to predict the proxy because the Linear Regression is fast and the model based on this proxy prediction had a $MAPE$ close to the one which was using a Random Forest to predict the proxy. The maximal depth of the Decision Tree for the *Two-Phase algorithm* is 30 and the one for the *One-Phase algorithm* is set to 20. Another classifier is the one that uses Decision Tree for the regression of the proxy and algorithm time regression. This classifier uses a maximal depth of 20 for both the regression of the *Two-Phase algorithm* time and the one of the *One-Phase algorithm* time. The last classifier model uses AdaBoost to predict the proxy. This model is tested because it had the best $MAPE$ when predicting the *Two-Phase algorithm* time. That model maximal depth for the Decision Tree predicting *Two-Phase algorithm* time is set to 30 and the one of the *One-Phase algorithm* is set to 20. The last model will be referred to as **DTAdaProxy**.

The results of the classifier are shown in Table 23, one can see that the classifier **DTDTProxy** and **DTAdaProxy** obtained a much lower average computation time of the test folds than the model **LRNNZB (b)** which is only based on the size and number of non-zero elements of b.

However, one can see that the metric $\gamma_{1,2}$ is high for the 3 classifiers. Indeed, in the model using a Linear regression to predict the proxy obtained a $\gamma_{1,2}$ 0.0764 and the one using AdaBoost to predict the proxy have one of 0.2140. Interestingly, the use of a Decision Tree to predict the proxy obtained a great average time whereas the one using a Random Forest, which predicted slightly better the time taken by the algorithms, obtained an higher average computation time of the test folds. As in Section First model, the **balanced-accuracy** does not capture the weight of choices that would lead to higher computation time. Therefore, some classification model have a better **balanced-accuracy** but a worse average computation time of the test folds. Namely, **DRLRProxy** which has a **balanced-accuracy** of 0.6314 and an average computation time of $24.3714s$ whereas **DTDTProxy** has a **balanced-accuracy** of 0.5557 and an average computation time of $14.5731s$.

Model	Average time (s)	Balanced accuracy	$\beta_{1,2}$	$\gamma_{1,2}$	$\gamma_{1,3}$
DTLRProxy	24.3714	0.6314	0.0012	0.0764	0.0000
DTDTProxy	14.5731	0.5557	0.0014	**0.1041**	0.0000
DTRFProxy	18.3274	0.5643	0.0016	0.1293	0.0000
DTAdaProxy	**12.4303**	0.6060	0.0037	0.2129	0.0000
LRNNZB (b)	27.3382	0.4589	0.0001	0.0062	0.0000

Table 23: Classifier comparison between $LRNNZB(b)$, $DTLRProxy$, $DTRFProxy$ and $DTAdaProxy$. The average time is the average of the total time needed to solve the test folds. The metrics average time, $\beta_{1,2}$, $\gamma_{1,2}$ includes all the time necessary to predict the fastest algorithm.

7.3 Random Forest

Similarly to previous regressors, Random Forest was tested with different maximal depth and with the 4 proxy models selected. The Table 24 shows the result of Random forest when using Linear Regression to predict the proxy. This model does not seem to be better to fit the *General algorithm* solve time than the Linear Regression using the size of b. The lowest $MAPE$ test of the *Two-Phase algorithm* time regression is not lower than the *Decision Tree* found to predict the computation time, this model have a lowest $MAPE$ test of 9.0966 and the one using a Random forest to predict the computation time has a value of 11.6213. Similarly, the *One-Phase* time regression has an highest $MAPE$ test when using Random Forest.

Algorithm	max_depth	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
General	10	0.0341	0.9038	**0.1502**	0.0891	0.0006	0.0245
	20	-0.0763	0.9445	0.1924	0.0549	0.0011	0.0652
	30	0.0733	0.9503	0.2038	0.0372	0.0015	0.1156
	40	0.0121	0.9501	0.2052	0.0345	0.0015	0.1191
	50	0.0653	0.9505	0.2082	0.0344	0.0014	0.1009
	80	-0.0209	0.9505	0.2032	0.0344	0.0014	0.1009
Two-Phase	10	-2.0911	0.9603	15.5057	11.5227	0.0005	0.0212
	20	-2.1783	0.9779	12.6080	1.9995	0.0009	0.0561
	30	-2.2118	0.9783	13.7250	0.9359	0.0015	0.1112
	40	-2.7146	0.9785	**11.6213**	0.6546	0.0019	0.1542
	50	-2.2062	0.9781	14.8658	0.5858	0.0019	0.1565
	80	-2.2490	0.9789	14.1782	0.5856	0.0019	0.1545
One-Phase	10	0.0643	0.9279	7.6334	5.2639	0.0005	0.0212
	20	-0.0135	0.9471	3.8842	0.6861	0.0010	0.0686
	30	-0.0523	0.9479	4.3763	0.2756	0.0016	0.1319
	40	-0.7946	0.9494	3.7671	0.2043	0.0025	0.1661
	50	0.0005	0.9479	**3.7130**	0.1925	0.0019	0.1540
	80	-0.8638	0.9471	4.1000	0.1959	0.0019	0.1545

Table 24: Results of Random Forest to predict the time taken by each algorithm and using **Linear Regression** to predict the proxy.

One can see in the Table 25 that the prediction of the *General algorithm* time have a R^2 test that is significant when using a Random Forest for the time prediction and a Decision Tree for the prediction of the number of non-zero elements in x. This score is equal to 0.7133. The previous highest R^2 test score obtained was when using a Linear Regression with only the size of b as feature, this regression has a R^2 test score of 0.6520. However, the $MAPE$ does not decrease significantly compared to previous model. It dropped from 0.1300 to 0.1175. For the regression of the *Two-Phase algorithm* time and the *One-Phase algorithm* the lowest average $MAPE$ of the test folds are higher than the one obtained with both Decision Tree for the prediction of the time and the proxy prediction. When using a Random Forest to predict the time of the *Two-Phase algorithm* the lowest $MAPE$ test obtained is 9.4258 whereas the lowest one obtained is equal to 5.7619. The lowest average $MAPE$ of the test folds for the *One-Phase algorithm* is 1.5749 and the lowest one obtained in the table is 2.0576 with a maximal depth of 20.

The Table 26 shows the result when using the Random Forest model to predict the proxy and when also using a Random Forest to predict the algorithm computation time. One can see in that table that the scores of the *General algorithm* is very similar to the score obtained with the same time predictor but with a Decision Tree as proxy regressor. It has a slighly lower $MAPE$ test than the previous best $MAPE$ test with 0.1173 instead of 0.1175. The lowest avrerage $MAPE$ test over the test folds of the *Two-Phase algorithm* time regression in the table is 8.3770 and it is not lower than the lowest $MAPE$ test obtained so far that had a value equal to 5.7619. The lowest average $MAPE$ of the test folds of the *One-Phase algorithm* in the table is also higher than the lowest one. This $MAPE$ test of the *One-Phase algorithm* time regression is obtained with a depth of 80 and its $MAPE$ is equal to 2.2202 whereas the lowest $MAPE$ test obtained was 1.5749, this value is obtained with a Decision Tree to predict the time and also a Decision Tree as proxy predictor.

Algorithm	max_depth	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
General	10	0.7133	0.9032	**0.1175**	0.0892	0.0010	0.0537
	20	0.7285	0.9456	0.1485	0.0550	0.0014	0.0970
	30	0.7798	0.9512	0.1626	0.0373	0.0016	0.1323
	40	0.6959	0.9516	0.1683	0.0346	0.0020	0.1565
	50	0.7338	0.9515	0.1661	0.0346	0.0019	0.1515
	80	0.6715	0.9516	0.1650	0.0346	0.0017	0.1374
Two-Phase	10	-4.9467	0.9710	39.5086	33.6275	0.0009	0.0471
	20	-5.0350	0.9801	9.8703	0.7827	0.0011	0.0619
	30	-4.8475	0.9802	9.6542	0.4702	0.0016	0.1313
	40	-4.9395	0.9804	**9.4258**	0.2319	0.0021	0.1641
	50	-4.9525	0.9801	9.8523	0.1494	0.0023	0.1734
	80	-4.9161	0.9804	9.4506	0.0899	0.0024	0.1800
One-Phase	10	-0.2602	0.9337	2.3815	0.6853	0.0009	0.0416
	20	-0.3247	0.9447	2.3103	0.2200	0.0012	0.0862
	30	-0.3290	0.9446	2.3622	0.1151	0.0019	0.1484
	40	-0.2821	0.9459	**2.2412**	0.0773	0.0023	0.1722
	50	-0.3623	0.9481	2.3264	0.0708	0.0024	0.1810
	80	-0.3330	0.9464	2.2202	0.0703	0.0022	0.1679

Table 25: Results of Random Forest to predict the time taken by each algorithm and using also a **Decision Tree** but with a maximal depth of 50 to predict the proxy.

The Table 27 shows the results of the Random Forest with AdaBoost to predict the number of non-zero elements in the solution. In that table, the lowest average $MAPE$ of the test folds is obtained with a depth of 10 for the *General algorithm* time regression and this regression has a $MAPE$ of 0.1174. The lowest $MAPE$ for the *Two-Phase algorithm* time regression is obtained with a depth of 20 with a value of 5.7233 which is very close to the previous lowest one obtained with a value of 5.7097 but the one of the Table 27 has a higher R^2 test score which has a value of -0.0603 instead of -0.2395. If a model have an higher R^2 but a very close $MAPE$ score, it suggests that the model is better. Even if the regression of the *Two-Phase algorithm* time regression seem better with a Random Forest to predict the time and AdaBoost to predict the proxy, the regression of the *One-Phase algorithm* time seems worse than the best previous one. Indeed, it has a $MAPE$ test equal to 2.1637 when using a depth of 20 whereas the best regression had a $MAPE$ test of 1.5749 for the *One-Phase algorithm* time predicted. This regression is predicted using a Decision Tree to predict the time and a Random Forest to predict the number of non-zero elements in the solution.

Algorithm	max_depth	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
General	10	0.7372	0.9031	**0.1173**	0.0892	0.0011	0.0850
	20	0.6259	0.9448	0.1548	0.0550	0.0016	0.1185
	30	0.7388	0.9509	0.1658	0.0373	0.0021	0.1587
	40	0.7215	0.9507	0.1718	0.0347	0.0021	0.1690
	50	0.7539	0.9506	0.1665	0.0344	0.0022	0.1659
	80	0.7623	0.9509	0.1645	0.0343	0.0022	0.1712
Two-Phase	10	-4.8636	0.9680	38.6339	34.5725	0.0010	0.0728
	20	-4.7731	0.9776	8.6679	1.1343	0.0013	0.0955
	30	-4.7149	0.9780	**8.3770**	0.7732	0.0019	0.1497
	40	-4.8858	0.9779	8.8907	0.4096	0.0023	0.1805
	50	-4.9257	0.9776	8.6910	0.4039	0.0027	0.1915
	80	-4.8845	0.9778	9.3595	0.3531	0.0025	0.1837
One-Phase	10	-0.9720	0.9342	2.0807	0.8492	0.0011	0.0839
	20	-1.0714	0.9476	**2.0297**	0.3692	0.0015	0.1091
	30	-1.1024	0.9475	2.0484	0.2032	0.0021	0.1587
	40	-1.0364	0.9457	2.0907	0.1522	0.0023	0.1803
	50	-1.0399	0.9476	2.0781	0.1443	0.0026	0.1837
	80	-0.9718	0.9477	2.1002	0.1433	0.0026	0.1878

Table 26: Results of Random Forest to predict the time taken by each algorithm and using **Random Forest** with 2 estimators and a maximal depth of 20 to predict the proxy.

Algorithm	max_depth	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
General	10	0.7350	0.9030	**0.1174**	0.0892	0.0029	0.1975
	20	0.7881	0.9456	0.1490	0.0551	0.0033	0.2055
	30	0.6752	0.9512	0.1654	0.0373	0.0037	0.2131
	40	0.7261	0.9513	0.1694	0.0347	0.0043	0.2176
	50	0.6754	0.9512	0.1676	0.0345	0.0043	0.2217
	80	0.6701	0.9514	0.1712	0.0345	0.0042	0.2198
Two-Phase	10	-0.1383	0.9708	36.3432	34.3883	0.0030	0.1994
	20	-0.0603	0.9798	**5.7233**	0.9028	0.0032	0.2035
	30	-0.1406	0.9797	6.1012	0.5533	0.0037	0.2116
	40	-0.0517	0.9800	6.6776	0.2698	0.0048	0.2249
	50	-0.0689	0.9798	6.6238	0.1761	0.0050	0.2304
	80	-0.0531	0.9802	6.6844	0.1320	0.0049	0.2288
One-Phase	10	-0.6542	0.9326	2.2068	0.8152	0.0030	0.1995
	20	-0.7571	0.9454	**2.1637**	0.3493	0.0033	0.2050
	30	-0.6687	0.9448	2.2220	0.1843	0.0039	0.2150
	40	-0.6144	0.9444	2.2486	0.1327	0.0048	0.2244
	50	-0.7220	0.9461	2.2759	0.1265	0.0050	0.2305
	80	-0.7139	0.9461	2.2778	0.1258	0.0048	0.2264

Table 27: Results of Random Forest to predict the time taken by each algorithm and using **AdaBoost** with 5 estimators and a maximal depth of 20 to predict the proxy.

With these regression algorithms with the best depth found, 7 classifiers were compared to the performances of **LRNNZB (b)**. The results are shown in Table 28. All these classifiers use a Random Forest with 2 estimators to predict the time taken by the *Two-Phase* and the *One-Phase* algorithm, the maximal depth of these estimators is adjusted to the proxy using the tables. The first model is called **RFLRProxy** because it uses a Random Forest to predict the time of each algorithm and it uses a Linear Regression to predict the number of non-zero elements in x. However, as in the previous section when using Decision Tree, the prediction of the *General algorithm* time was less well predicted when using a Random forest to predict the time combined with a Linear Regression to predict the proxy than when just using a Linear Regression only based on the size of b. Therefore, the simple Linear Regression with only the size of b was used to predict the *General algorithm* time, this model obtained an average computation time of $16.7543s$. Random Forest was also combiined with a Decision Tree to predict the proxy. The models **RFDTProxy** uses a Random forest to predict the algorithm time and a Decision Tree to predict the proxy. The difference between the model **RFDTProxy (a)** and **RFDTProxy (b)** is the regression of the *General algorithm* time. In **RFDTProxy (a)**, it is the Random Forest with a max depth of 10 that predicts the *General algorithm* time. For the model **RFDTProxy (b)**, the *General algorithm* time is predicted by the same Linear Regression used in **RFLRProxy**. Actually, the difference between all (a) and (b) is this *General algorithm* time regressor, the (a) models uses a Random Forest and (b) models uses a Linear Regression with the size of b as feature. One can see in Table 28 that overall, the "(b)" classifier obtained a lower average computation time fo the test folds despite having a lower **balanced-accuracy** than the "(a)" version. One can also see the model with the lowest average time computation, it is obtained with the model **RFAdaProxy (b)** which has an average time computation of $12.9462s$ which is slightly higher than the model *DTAdaProxy* obtained in previous section which have an average time of $12.4303s$. Furthermore, the metric $\gamma_{1,2}$ is very similar to all classifier.

Model	Average time (s)	Balanced accuracy	$\beta_{1,2}$	$\gamma_{1,2}$	$\gamma_{1,3}$
RFLRProxy	19.1151	0.6052	0.0038	0.2126	0.0000
RFDTProxy (a)	18.1238	0.6872	0.0045	0.2221	0.0000
RFDTProxy (b)	13.4330	0.6131	0.0041	0.2157	0.0000
RFRFProxy (a)	26.9035	0.6827	0.0042	0.2193	0.0000
RFRFProxy (b)	14.9314	0.6169	0.0040	0.2173	0.0000
RFAdaProxy (a)	16.4428	0.6419	0.0043	0.2213	0.0000
RFAdaProxy (b)	**12.9462**	0.5617	0.0041	0.2186	0.0000
LRNNZB (b)	27.3373	0.4589	0.0001	0.0054	0.0000

Table 28: Classifier comparison between the model obtained in this section with a Random Forest with a max depth which obtained the best $MAPE$ for the algorithm regressions and with the different proxy selected. The average time is the average of the total time needed to solve the test folds. The metrics average time, $\beta_{1,2}$, $\gamma_{1,2}$ and $\gamma_{1,3}$ includes all the time necessary to predict the fastest algorithm.

7.4 Extremely Randomized Trees

In this section, Extremely Randomized Trees was tested to predict the 3 algorithm computation times. Because the time window to predict the algorithm is low, the number of estimators used by the Extremely Randomized Trees should not be high. Therefore, in this section the number of estimators used in the Extremely Randomized Trees to predict the computation time is always set to 2. The parameter that changes to find the best regression is the maximal depth of these two estimators.

One can see in Table 29 that the Extremely Randomize Trees with a maximal depth of 10 using a Linear Regression predicted the *General algorithm* with the highest average R^2 score of the test folds so far. Its value is equal to 0.8429, the previous highest score was 0.7881. However, the $MAPE$ test is not the lowest one, the lowest value was 0.1173. For the *Two-Phase algorithm* regression, the depth that leads to the lowest average $MAPE$ of the test folds is equal to 80. The $MAPE$ for this regressor is 8.5867, this regressor does not seem to perform well compared to previous models tested because the best $MAPE$ test obtained was equal to 5.7619. The *Two-Phase algorithm* regression obtained a $MAPE$ test of 1.9818 which seems great compared to other models, but it is, actually, not the lowest.

The Table 30 shows that Extremely Randomized Trees with Decision Tree to predict the proxy with a depth of 10 obtained similar results for the *General algorithm* as the one with Linear Regression. This may be due to previous observed results. Indeed, in section Model based on the true number of non-zero elements in the solution, the General algorithm does not seem to highly depend on the number of non-zero elements in the solution. Therefore, having a more accurate predictor for this value should not improve the performance that much. The *Two-Phase algorithm* was predicted with a lowest $MAPE$ test of 7.2151 with a depth of 80, this model also did beat the lowest $MAPE$ test obtained. In the table, the *One-Phase algorithm* time was predicted with the lowest $MAPE$ test when a maximal depth of 30 was set to the trees, this $MAPE$ is equal to 1.8949. One can also see in Table 30 that the R^2 score is for the first time positive when predicting the *One-Phase algorithm* but it is still not great.

Algorithm	max_depth	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
General	10	0.8429	0.8846	**0.1477**	0.1308	0.0005	0.0216
	20	-0.2996	0.9639	0.1971	0.0638	0.0009	0.0505
	30	0.6967	0.9968	0.1720	0.0191	0.0014	0.0992
	40	0.7111	0.9981	0.2039	0.0034	0.0014	0.1124
	50	0.4701	0.9981	0.2073	0.0024	0.0017	0.1256
	80	0.7359	0.9981	0.1771	0.0023	0.0015	0.1168
Two-Phase	10	-2.9501	0.9466	119.3993	118.4647	0.0005	0.0204
	20	-0.5708	0.9983	33.1809	4.6062	0.0008	0.0421
	30	-0.6759	1.0000	11.4462	1.1697	0.0014	0.1123
	40	-3.4489	1.0000	16.4851	0.2429	0.0019	0.1581
	50	-4.2975	1.0000	26.9042	0.0408	0.0022	0.1742
	80	-0.2770	1.0000	**8.5867**	0.0076	0.0023	0.1749
One-Phase	10	0.5684	0.9434	30.3206	30.4541	0.0005	0.0183
	20	0.3228	0.9989	4.0906	3.1533	0.0008	0.0372
	30	0.0712	0.9999	**1.9818**	0.7526	0.0013	0.0966
	40	0.2832	1.0000	2.0380	0.1434	0.0019	0.1626
	50	0.1447	1.0000	4.7032	0.0223	0.0022	0.1730
	80	0.2011	1.0000	3.4523	0.0043	0.0023	0.1772

Table 29: Results of Extremely Randomized Trees to predict the time taken by each algorithm and using **Linear Regression** to predict the proxy.

Algorithm	max_depth	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
General	10	0.8028	0.8833	**0.1520**	0.1309	0.0010	0.0537
	20	0.7776	0.9624	0.1334	0.0643	0.0014	0.1130
	30	0.7852	0.9967	0.1694	0.0190	0.0019	0.1505
	40	0.6162	0.9981	0.1640	0.0034	0.0020	0.1583
	50	0.7627	0.9981	0.1623	0.0024	0.0020	0.1650
	80	0.6793	0.9981	0.1591	0.0023	0.0020	0.1639
Two-Phase	10	-0.8501	0.9625	136.3661	130.2248	0.0009	0.0470
	20	-1.3366	0.9996	7.7397	1.2600	0.0013	0.0838
	30	-0.7536	1.0000	8.3005	0.7007	0.0017	0.1439
	40	-0.8395	1.0000	7.7399	0.3703	0.0024	0.1777
	50	-1.0380	1.0000	8.9674	0.1028	0.0029	0.1941
	80	-0.5349	1.0000	**7.2151**	0.0076	0.0029	0.1974
One-Phase	10	0.0996	0.9338	36.0429	34.3301	0.0009	0.0501
	20	0.1850	0.9979	2.5261	1.0586	0.0013	0.0913
	30	0.1057	0.9999	**1.8949**	0.2283	0.0018	0.1412
	40	0.2353	1.0000	1.9300	0.1004	0.0024	0.1827
	50	-0.1515	1.0000	2.0476	0.0277	0.0028	0.1940
	80	0.3580	1.0000	2.1052	0.0043	0.0029	0.1978

Table 30: Results of Extremely Randomized Trees to predict the time taken by each algorithm and using **Decision Tree** with a maximal depth of 50 to predict the proxy.

One can see in Table 31, the influence of the maximal depth of the Extremely Randomized Trees on the time regressions when using the selected Random Forest as proxy predictor. The *General algorithm* time regression that is obtained with a depth of 20 has an average $MAPE$ of the test folds of 0.1294 that is not far from the lowest one obtained so far which have a value of 0.1175. The *Two-Phase algorithm* has a lowest $MAPE$ test value with a depth of 80, its value equals to 8.5867. This performance is similar to the one obtained with the same proxy and a Random Forest to predict the time, however, Extremely Randomized Trees needed deeper trees resulting to a need of more computation time. The lowest average $MAPE$ of the test folds for *One-Phase algorithm* time regression was obtained with a depth of 40. For that depth, the $MAPE$ test is equal to 1.5552 which is, actually, the lowest value obtained so far for that regression, the previous lower value was very close with a number of 1.5749.

Algorithm	max_depth	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
General	10	0.8385	0.8839	0.1499	0.1312	0.0013	0.0841
	20	0.7049	0.9625	**0.1294**	0.0637	0.0017	0.1472
	30	0.6639	0.9967	0.1555	0.0184	0.0021	0.1702
	40	0.7115	0.9981	0.1723	0.0034	0.0022	0.1716
	50	0.7686	0.9981	0.1643	0.0023	0.0023	0.1799
	80	0.7009	0.9981	0.1651	0.0023	0.0022	0.1739
Two-Phase	10	-0.8783	0.9598	179.9604	176.3994	0.0012	0.0802
	20	-2.9170	0.9993	10.6914	2.1913	0.0015	0.1241
	30	-0.8725	1.0000	8.8241	0.8545	0.0020	0.1639
	40	-0.9593	1.0000	9.4561	0.3924	0.0026	0.1879
	50	-1.4023	1.0000	**8.0853**	0.1089	0.0032	0.2024
	80	-0.9516	1.0000	10.1392	0.0076	0.0034	0.2054
One-Phase	10	0.2817	0.9413	35.8867	34.4841	0.0012	0.0802
	20	-0.0399	0.9989	2.1220	0.7706	0.0015	0.1192
	30	-0.2638	0.9999	2.4926	0.3754	0.0020	0.1639
	40	-0.5817	1.0000	**1.5552**	0.1446	0.0027	0.1878
	50	-0.5454	1.0000	1.8147	0.0288	0.0031	0.2027
	80	-0.4751	1.0000	2.5273	0.0043	0.0035	0.2058

Table 31: Results of Extremely Randomized Trees to predict the time taken by each algorithm and using **Random Forest** with 2 estimators each having a maximal depth of 20 to predict the proxy.

The Table 32 shows the results for the Extremely Randomized Trees with AdaBoost to predict the number of non-zero elements in the solution. The lowest $MAPE$ for the regression of the *General algorithm* for this model is obtained with a depth of 10 and its values is equal to 0.1509. This value does not seem great compared to others models tested. Indeed, many other models could get a $MAPE$ around 0.13 and for much lower computation time, the $\gamma_{1,2}$ that represents the ratio of the samples which cannot win time if the best choice is made over the second best choice after making that regression without even making a regression over the two other algorithms. The regression of the *Two-Phase algorithm* achieved a lower average $MAPE$ of the test folds than the best one obtained in previously tested models. Indeed, the lowest $MAPE$ obtained in the table for the *Two-Phase algorithm* is 5.3612 and is obtained with a depth of 30 whereas the previously lowest value was 5.7619. Moreover, for the first time the R^2 test score is positive for the regression of this algorithm time. For the *One-Phase algorithm* time regression, the lowest $MAPE$ test is obtained with a depth of 30 and has a value of 1.8287 which is the lowest one obtained for this algorithm and with AdaBoost as proxy predictor.

Algorithm	max_depth	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
General	10	0.8194	0.8836	**0.1509**	0.1314	0.0030	0.2001
	20	0.6772	0.9604	0.1513	0.0643	0.0034	0.2073
	30	0.6399	0.9970	0.1565	0.0189	0.0040	0.2163
	40	0.6826	0.9981	0.1662	0.0035	0.0042	0.2178
	50	0.7870	0.9981	0.1565	0.0024	0.0040	0.2163
	80	0.6481	0.9981	0.1714	0.0023	0.0042	0.2205
Two-Phase	10	0.1514	0.9651	133.7584	130.5599	0.0030	0.1990
	20	0.0793	0.9997	8.7585	1.4146	0.0033	0.2056
	30	0.2525	1.0000	**5.3612**	0.7857	0.0038	0.2138
	40	0.1140	1.0000	5.8989	0.3920	0.0045	0.2229
	50	0.2042	1.0000	5.3973	0.1006	0.0047	0.2253
	80	0.0835	1.0000	5.7792	0.0077	0.0051	0.2326
One-Phase	10	0.4284	0.9383	35.9911	34.3465	0.0030	0.1990
	20	0.3395	0.9987	2.7718	1.2617	0.0033	0.2060
	30	0.4078	0.9999	**1.8287**	0.3772	0.0038	0.2142
	40	0.2909	1.0000	5.0703	0.1619	0.0046	0.2233
	50	0.3792	1.0000	2.2012	0.0364	0.0047	0.2267
	80	0.2460	1.0000	1.9765	0.0043	0.0050	0.2298

Table 32: Results of Extremely Randomized Trees to predict the time taken by each algorithm and using **AdaBoost** with 5 Decision Tree as estimators and with a maximal depth of 20 to predict the proxy.

With the max depth which minimize the $MAPE$ test obtained in this section, 8 classifiers are made and their results are shown in Table 33. The name of these models are chosen as in previous section, the first part of the name indicates the regression model used to predict the time algorithm and the second part indicates the proxy predictor and if there is a letter b after the model name, it means that the *General algorithm* is predicted using a Linear Regression with only the size of b as feature. Models with the letter a after their names signify that the *General algorithm* time is predicted with the model given by the first part of their name. For example, **ETLRProxy (a)** uses Extremely Randomized Trees to predict the 3 algorithm times and these regression models use the proxy predicted by a linear regression. In Table 33, similarly, as model using Random Forest to predict the algorithm times, the model using a Linear Regression with only the size of b as feature leads to lower average computation time. The model using a Linear Regression to predict the proxy has the highest computation time and models which uses better predictor for the proxy seem to have a lower computation time which may indicate that the proxy is important for the prediction of the best algorithm. The lowest average computation time is not as good as previous models. Indeed, when using Decision Tree to predict the algorithm times, the lowest average computation time was $12.43s$ and when these regressions are made with Random Forest, this time becomes $12.95s$ whereas the best one in Table 33 has an average computation time of the test folds equal to $14.26s$. This value is obtained with **ETDTProxy (b)**.

Model	Average time (s)	Balanced accuracy	$\beta_{1,2}$	$\gamma_{1,2}$	$\gamma_{1,3}$
ETLRProxy (a)	31.6717	0.7133	0.0041	0.2210	0.0000
ETLRProxy (b)	28.0744	0.6376	0.0037	0.2108	0.0000
ETDTProxy (a)	18.1180	0.6854	0.0068	0.2624	0.0000
ETDTProxy (b)	**14.2553**	0.6160	0.0056	0.2363	0.0000
ETRFProxy (a)	27.0545	0.6976	0.0046	0.2251	0.0000
ETRFProxy (b)	15.4451	0.6311	0.0048	0.2286	0.0000
ETAdaProxy (a)	17.5896	0.6616	0.0063	0.2543	0.0000
ETAdaProxy (b)	14.9077	0.5839	0.0052	0.2317	0.0000
LRNNZB (b)	27.3384	0.4589	0.0001	0.0070	0.0000

Table 33: Classifier comparison between the models obtained in this section with a Extremely Randomized Trees with a maximal depth that obtained the best $MAPE$ for the algorithm regressions and with the different proxy selected. The average time is the average of the total time needed to solve test folds. The metrics average time, $\beta_{1,2}$, $\gamma_{1,2}$ and $\gamma_{1,3}$ includes all the time necessary to predict the fastest algorithm

7.5 AdaBoost

In this section, AdaBoost is used to predict the 3 algorithm computation times. As with Extremely Randomized Trees, the time window is low and because the model is used 3 times, the number of trees used by the model is limited to 2. This limitation is on all the regression models trained in this section. The parameters that will be changed to find the best regressors is the maximal depth given to the two estimators.

The Table 34 shows the score of AdaBoost when predicting the time taken by the 3 algorithm with different maximal depth parameters of the Decision Tree used as an estimator and with the use of a Linear Regression to predict the proxy. In that table, the lowest average $MAPE$ of the test folds for the *General algorithm* is obtained with a depth of 10, its value is equal to 0.16012 which is the worst obtained for the regression of the *General algorithm.* For the regression of the *Two-Phase algorithm* time, the $MAPE$ test was also high compared to the one obtained in with previous regression. Indeed, with a depth of 20, the model obtained a $MAPE$ of 12.4458 which is one of the highest obtained for this algorithm. The $MAPE$ test of the *One-Phase algorithm* is surprisingly good compared to other regressors. This score is obtained with a maximal depth of 20 and is equal to 1.4904 which is the lowest obtained so far.

The result of the algorithm time regressions when using AdaBoost and the supposed best Decision Tree to predict the proxy is shown in Table 35. The lowest average $MAPE$ of the test folds for the regression of the *General algorithm* is obtained with a depth of 10 and is equal to 0.1266 and one can see that the R^2 seem great, these value are not bad compared to other models but they are not the best seen so far. The $MAPE$ test for the regression of the *Two-Phase algorithm* is equal to 9.7425 when using the *max_depth* parameter of the decision tree estimator set to 30. This value is far from the lowest value obtained for this algorithm, the lowest value is 5.3612. The regression of the *One-Phase algorithm* which obtained the lowest $MAPE$ test on the table was the one with depth of 20. This model obtained a $MAPE$ over the test folds of 1.9443 which is still higher than the lowest one obtained so far for this algorithm. Indeed, the lowest one obtained a value of 1.5552 and the regressor was obtained with Extremely Randomized Trees for the time regression and Random Forest for the proxy prediction.

Algorithm	max_depth	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
General	10	0.1100	0.8921	**0.1612**	0.0926	0.0011	0.0706
	20	-0.0533	0.9125	0.1986	0.0584	0.0014	0.1124
	30	-0.3086	0.9170	0.2216	0.0393	0.0018	0.1500
	40	-0.4029	0.9183	0.2235	0.0355	0.0019	0.1582
	50	0.0142	0.9173	0.2584	0.0353	0.0021	0.1659
	80	-2.8881	0.9221	0.2631	0.0362	0.0020	0.1678
Two-Phase	10	-1.7954	0.9507	14.9136	11.3013	0.0010	0.0533
	20	-2.0265	0.9614	**12.4458**	1.9720	0.0014	0.1040
	30	-2.1669	0.9679	135.4954	1.0312	0.0018	0.1500
	40	-5.9059	0.9661	140.8121	0.8347	0.0023	0.1765
	50	-7.0811	0.9639	44.3259	0.5886	0.0025	0.1874
	80	-4.9772	0.9661	20.6734	0.8924	0.0027	0.1892
One-Phase	10	-0.0747	0.9148	5.7727	5.6348	0.0010	0.0569
	20	0.2644	0.9234	**1.4904**	0.7883	0.0014	0.1067
	30	-1.5894	0.9198	3.4392	0.3118	0.0020	0.1628
	40	-1.4008	0.9187	5.3052	0.2289	0.0023	0.1765
	50	-1.6148	0.9261	6.7486	0.2438	0.0025	0.1853
	80	-2.2709	0.9146	4.8373	0.2337	0.0027	0.1908

Table 34: Results of AdaBoost to predict the time taken by each algorithm and using **Linear Regression** to predict the proxy.

Algorithm	max_depth	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
General	10	0.7769	0.8921	**0.1266**	0.0928	0.0016	0.1207
	20	0.7140	0.9128	0.1746	0.0591	0.0020	0.1608
	30	0.7215	0.9155	0.1693	0.0387	0.0027	0.1924
	40	0.7255	0.9191	0.1795	0.0361	0.0025	0.1838
	50	0.7126	0.9195	0.2132	0.0360	0.0026	0.1857
	80	0.7403	0.9218	0.1724	0.0360	0.0024	0.1797
Two-Phase	10	-6.5470	0.9598	38.1644	31.3656	0.0014	0.1109
	20	-3.6551	0.9680	9.7455	0.8006	0.0019	0.1542
	30	-6.8279	0.9727	**9.7425**	0.4857	0.0026	0.1863
	40	-5.6895	0.9691	10.6426	0.2449	0.0028	0.1920
	50	-6.4752	0.9729	9.8618	0.1601	0.0031	0.1993
	80	-7.8365	0.9661	10.3926	0.0915	0.0029	0.1974
One-Phase	10	-0.2885	0.9187	2.4333	0.7457	0.0014	0.1049
	20	-0.6259	0.9352	**1.9443**	0.2317	0.0019	0.1608
	30	-0.5300	0.9289	2.3593	0.1233	0.0029	0.1956
	40	0.1169	0.9349	2.6021	0.0850	0.0030	0.1983
	50	-0.8843	0.9262	2.2619	0.0748	0.0032	0.2018
	80	-1.1058	0.9234	2.3555	0.0768	0.0029	0.1951

Table 35: Results of AdaBoost to predict the time taken by each algorithm and using **Decsision Trees** to predict the proxy.

The Table 36 shows the results for different depth of the Decision Trees used by AdaBoost to predict the time of the algorithm and with the proxy predicted with Random Forest. The maximal depth which obtained the best regression for *General algorithm* in the table is equal to 10 with a $MAPE$ of 0.1263 which is similar to what can be obtained with other models. For the *Two-Phase* *algorithm* the lowest average $MAPE$ of the test folds is equal to 6.6193 which is higher than the lowest one obtained. The best value for this algorithm in the table is achieved with a depth of 30. The lowest $MAPE$ test for *One-Phase algorithm* time regression in the table is equal to 1.4614 which is the lowest seen so far for this algorithm. However, the R^2 test score is negative, although, some models had an higher $MAPE$ with a positive R^2 score.

The last table showing the result of regression with different value of maximal depth for the Decision Tree used by AdaBoost is shown in Table 37. In this table the regression of the number of non-zero elements in the solution is also made with AdaBoost. The best regression of the *General algorithm* found with these regressors has a $MAPE$ test of 0.1263 which is, actually, equal to the one obtained with AdaBoost and the proxy regressor, which was Random Forest. The depth which obtained the lowest average $MAPE$ of the test folds for the *Two-Phase algorithm* time regression is equal to 20. Its $MAPE$ is equal to 5.6830, which is not the lowest one but it has a value close to it.Indeed, the lowest $MAPE$ test for this algorithm is equal to 5.3612. For the *One-Phase algorithm*, the average $MAPE$ of the test folds is equal to 1.4614 which was obtained with a depth of 20. Actually, this value is the lowest one obtained for the regression of the *One-Phase algorithm* time but the R^2 test score is negative, where other model manage a similar $MAPE$ with positive R^2.

Algorithm	max_depth	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
General	10	0.8220	0.8923	**0.1263**	0.0928	0.0018	0.1421
	20	0.6918	0.9152	0.1522	0.0583	0.0021	0.1699
	30	0.6737	0.9155	0.1864	0.0394	0.0025	0.1855
	40	0.7416	0.9199	0.1777	0.0361	0.0025	0.1875
	50	0.6878	0.9223	0.2514	0.0359	0.0029	0.1971
	80	0.6588	0.9174	0.1873	0.0358	0.0045	0.2213
Two-Phase	10	-4.8445	0.9561	35.1183	30.6593	0.0017	0.1406
	20	-5.6836	0.9713	9.5328	1.1016	0.0019	0.1589
	30	-1.8797	0.9665	**6.6193**	0.7139	0.0024	0.1800
	40	-0.3567	0.9695	7.7181	0.4232	0.0029	0.1947
	50	-4.7562	0.9712	9.6111	0.2836	0.0036	0.2093
	80	-5.1089	0.9656	8.7228	0.2276	0.0060	0.2416
One-Phase	10	-0.9386	0.9157	1.8054	0.8685	0.0017	0.1398
	20	-1.0556	0.9244	**1.4614**	0.3791	0.0020	0.1685
	30	-1.6794	0.9328	1.5585	0.2142	0.0026	0.1876
	40	-1.6908	0.9308	1.4906	0.1481	0.0030	0.1973
	50	-0.8564	0.9169	1.6521	0.1487	0.0035	0.2087
	80	-1.3786	0.9315	1.9535	0.1409	0.0053	0.2293

Table 36: Results of AdaBoost to predict the time taken by each algorithm using **Random Forest** with 2 estimators and with a maximal depth of 20 to predict the proxy.

Algorithm	max_depth	R^2 test	R^2 train	MAPE test	MAPE train	$\beta_{1,2}$	$\gamma_{1,2}$
General	10	0.7909	0.8912	**0.1263**	0.0929	0.0042	0.2177
	20	0.7055	0.9151	0.1833	0.0580	0.0044	0.2253
	30	0.5233	0.9226	0.2051	0.0391	0.0043	0.2211
	40	0.7427	0.9170	0.1772	0.0360	0.0047	0.2256
	50	0.6874	0.9184	0.1838	0.0360	0.0046	0.2243
	80	0.7093	0.9172	0.1764	0.0354	0.0046	0.2240
Two-Phase	10	-0.3267	0.9595	36.0455	34.0427	0.0042	0.2163
	20	-0.5293	0.9677	**5.6830**	0.9245	0.0041	0.2203
	30	0.0879	0.9692	24.4041	0.5714	0.0042	0.2187
	40	0.2612	0.9696	5.7660	0.2848	0.0050	0.2296
	50	-0.3127	0.9716	5.9178	0.1829	0.0051	0.2302
	80	-0.5445	0.9639	6.2715	0.1330	0.0053	0.2325
One-Phase	10	-0.7538	0.9116	2.2011	0.8836	0.0042	0.2181
	20	-0.8843	0.9266	**2.0115**	0.3607	0.0043	0.2228
	30	-0.6783	0.9213	2.1221	0.1973	0.0045	0.2219
	40	-0.7754	0.9269	2.2321	0.1451	0.0051	0.2304
	50	-1.1621	0.9268	2.1965	0.1338	0.0051	0.2308
	80	-1.0858	0.9220	2.1679	0.1312	0.0052	0.2310

Table 37: Results of AdaBoost to predict the time taken by each algorithm and using **AdaBoost** with 5 estimators and with a maximal depth of 20 to predict the proxy.

As in previous section, the Table 38 shows the result of the classification if AdaBoost is used to predict the algorithm time and with different proxy. In the model name, for example in **AdaLRProxy (b)**, the first part "Ada" is because AdaBoost is used for the algorithm time regression, "LRProxy" is because Linear Regression is used for the proxy regression and if there is a "(b)", it means that the *General algorithm* is predicted with a Linear Regression using the size of b as unique feature. As previous models, models using the Linear Regression with only the size of b to predict the *General algorithm* have lower average predicted times even if their *balanced-accuracy* is lower. The model obtaining the lowest average time in the table is **AdaDTProxy (b)** with a time of $13.7499s$ which is higher than the one of **RFAdaProxy (b)** and **DTAdaProxy** which obtained an average time of $12.9462s$ and $12.4303s$ respectively.

Model	Average time (s)	Balanced accuracy	$\beta_{1,2}$	$\gamma_{1,2}$	$\gamma_{1,3}$
AdaLRProxy (a)	16.9415	0.6562	0.0040	0.2163	0.0000
AdaLRProxy (b)	14.6581	0.5776	0.0027	0.1924	0.0000
AdaDTProxy (a)	17.7854	0.6677	0.0045	0.2244	0.0000
AdaDTProxy (b)	**13.7499**	0.5930	0.0038	0.2142	0.0000
AdaRFProxy (a)	24.2992	0.6853	0.0052	0.2327	0.0000
AdaRFroxy (b)	23.4228	0.6080	0.0040	0.2173	0.0000
AdaAdaProxy (a)	24.5938	0.6380	0.0062	0.2433	0.0000
AdaAdaProxy (b)	22.9493	0.5611	0.0053	0.2317	0.0000
LRNNZB (b)	27.3382	0.4589	0.0001	0.0061	0.0000

Table 38: Classifier comparison between the models obtained in this section with AdaBoost using Decision Trees with a maximal depth which obtained the best $MAPE$ for the algorithm regressions and with the different proxy predictor selected. The average time is the average of the total time needed to solve the test folds. The metrics average time, $\beta_{1,2}$, $\gamma_{1,2}$ and $\gamma_{1,3}$ includes all the time necessary to predict the fastest algorithm.

7.6 Comparison of the best models

In the classifiers tested, the regressions of the algorithm times use always the same regression models. One could try different combinations of regressors to predict the algorithm times. However, the noticeable difference of $MAPE$ test for the time regressions of the *Two-Phase algorithm* is when using AdaBoost as proxy predictor and Extremely Randomized Trees for the time regressions, this one obtained the lowest $MAPE$ with a value of 5.3612. The lowest $MAPE$ for this algorithm when removing all models using AdaBoost as proxy predictor is 6.68193, thus, to have the lowest $MAPE$, AdaBoost should be used to predict the proxy. For the *One-Phase algorithm* time regression, the $MAPE$ is better predicted when the proxy regressor is a Random Forest or a Decision Tree. Indeed, when AdaBoost predicts the proxy, the best $MAPE$ test for the *One-Phase algorithm* time regression obtained is 1.8287 whereas when other proxy predictors are used the lowest $MAPE$ is 1.5552. Therefore, having AdaBoost to predict the proxy should lead to better *Two-Phase algorithm* time regression and to have the best regression for the *One-Phase algorithm* time, the proxy should be one of the model. Thus, two proxy models should be used resulting to one more prediction and, thus, to more computation time. Therefore, to avoid adding more computation time, these other combinations were not explored.

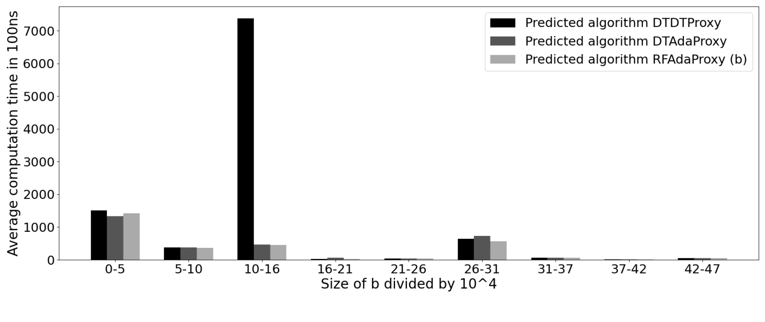


Figure 32: Bar plot of the average computation time taken by the predicted algorithm of the models **DTDTProxy**, **DTAdaProxy** and **RFAdaProxy (b)**.

The best models which are selected are the one which achieved the lowest average computation time of the test folds, this value is around $12s$, these models are **DTAdaProxy** and **RFAdaProxy (b)**. Another model that will be tested is **DTDTProxy** which obtained an average computation time of $14.57s$. This model is tested because it obtained a $\gamma_{1,2} = 0.1043$ which is much lower compared to the other one obtained. The two other models have a $\gamma_{1,2}$ equal to 0.2129 and 0.2186 respectively.

One can see in Figure 32 that the time that is lost by the model **DTDTProxy** compared to other models is mostly due to samples that have size of b in between 10×10^4 and 16×10^4. The Figure 33 shows the proportion of the algorithm predicted with different b size ranges. In this figure, the **DTDTProxy** does not seem to have a much different proportions of prediction for the problematic range. One problem that may occur is that some algorithm prediction are swapped. Indeed, if an algorithm is predicted in the place of another one and the other one is predict at the place of first algorithm then the proportion stays the same but the algorithm are not well classified. Because of the problem in that range of b sizes, **DTDTProxy** will not be the chosen as final model. In the Figure 32, for a size of b between 16×10^4 and 21×10^4, the prediction of the *Two-Phase algorithm* seem a bit too high compared to the prediction of **RFAdaProxy (b)**. In the Figure 32, one can see that **RFAdaProxy (b)** proportions of algorithm predictions is close to the ground truth. This means that the model **DTAdaProxy** was wrongly predicting the *Two-Phase algorithm* without impacting to much the average computation time. This wrong classification could lead to problems, therefore, the final model chosen is **RFAdaProxy (b)**. In the Figure 34, one can see that **RFAdaProxy (b)** predictions leads to a time much closer to the ground truth than **LRNNZB (b)**.

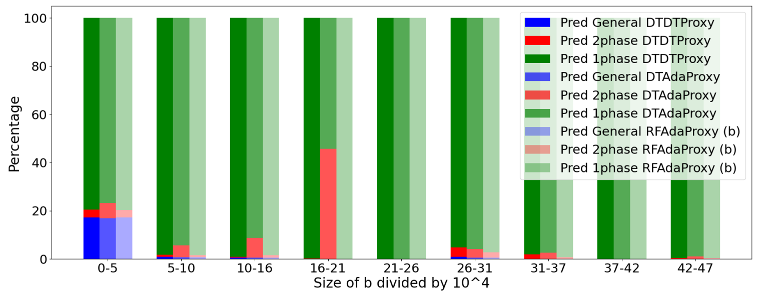


Figure 33: Bar plot of the average computation time taken by the predicted algorithm of the models **DTDTProxy**, **DTAdaProxy** and **RFAdaProxy (b)**.

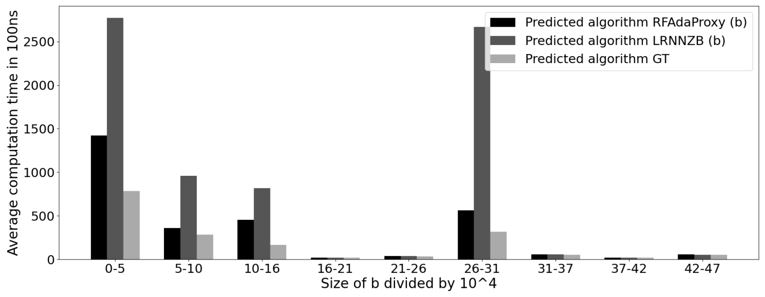


Figure 34: Bar plot with the proportion of time each algorithm is predicted for different rang of b size. The models shown are **RFAdaProxy (b)**, **LRNNZB (b)** and the ground truth called **GT**.

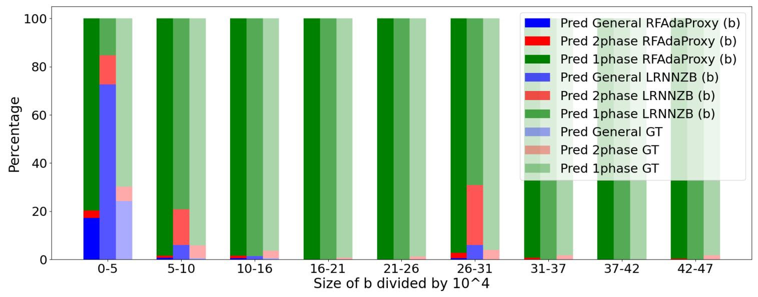


Figure 35: Bar plot of the average computation time taken by the predicted algorithm of the models **RFAdaProxy (b)**, **LRNNZB (b)** and the ground truth called **GT**.

8 Model assessment and conclusion

Now that a model has been chosen, its performances must be tested with the part of the dataset which was not used before. This part represent 20% of the samples. The model **RFAdaProxy (b)** will be trained with all the folds which were used in the model selection part, this part represents the remaining 80% of the data. Then, the performance of the model will be computed using the unseen test set. The final step, if the model performs well, is to train the model **RFAdaProxy (b)** with all the dataset.

The model which would always preditct the fastest algorithm would complete the solve of the test set with a time of $19.95s$. The model **RFAdaProxy (b)** obtained a total computation time of $34.78s$ on the unseen 20% of the dataset. In order to understand the performance of the model **RFAdaProxy (b)**, **LRNNZB (b)** was also tested on the test set. The algorithms chosen by **LRNNZB (b)** completed the solve of these problems with a time of $44.85s$.

This results show that the model **RFAdaProxy (b)** still obtains results that improves the average computation time when solving multiple problems compared to the simple model using linear regressions.

Github

https://github.com/Gael-di-raimo/MasterThesis

References

[1] Dimitris Bertsimas and John N Tsitsiklis. *Introduction to linear optimization*, volume 6. Athena scientific Belmont, MA, 1997.

[2] John JH Forrest and John A Tomlin. Updated triangular factors of the basis to maintain sparsity in the product form simplex method. *Mathematical programming*, 2(1):263–278, 1972.

[3] Leena M Suhl and Uwe H Suhl. A fast lu update for linear programming. *Annals of Operations Research*, 43(1):33–47, 1993.

[4] Timothy A Davis. *Direct methods for sparse linear systems*. SIAM, 2006.

[5] Pierre Geurts and Louis Wehenkel. Introduction to machine learning, faculty of applied sciences, university of liège, September 2020.

[6] Scikit-learn: Metrics and scoring: quantifying the quality of predictions. `https://scikit-learn.org/stable/modules/generated/sklearn.utils.validation.check_is_fitted.html`. Accessed: 2022-11-17.

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.