High Performance Computing

Year: 2018/2019

Final Project

BDT: Gradient Boosted Decision Tables for High Accuracy and Scoring Efficiency

---

28th May 2019

Hibraj Feliks 854342

Index

1 Introduction

1.1 Project overview

The project is about the design and performance evaluation of a proposed parallel implementation. In particular, the considered problem is that of **Gradient Boosted Decision Tables**, which are an alternative to the more used regression trees. A discussion about the usage of Decision Trees against that of Regression Trees can be found in [1]. Other implementation specific details are present in that paper too.

In this project, the **Multi-threading** approach has been chosen to deliver the parallel implementation.
Moreover, backfitting strategies have been implemented, in particular:

- **Cyclic backfitting**: This is the standard backfitting algorithm; we go back to the first cut and re-optimize all cuts in sequential "cyclic" order.
- **Random backfitting**: We randomly select a cut in the current decision table to backfit.
- **Greedy backfitting**: This is the greedy stagewise approach where we select a cut to backfit that gives the largest reduction in variance.

2 Tricks considered for the project

2.1 Inverted Index

An important observation for making the whole algorithm faster is to use an inverted index to store the documents having the same value for a certain feature. That is important since in the calculation of the best cut for a certain level of the decision table, we are going to consider all possible cuts and choose the one that provides the best MSE. So, having the possible values of a certain feature ordered in decreasing order makes it easier to immediately have the documents which are going to change class (in a tree point of view, they move to the right sibling leave).
We can see the proposed structure in Fig. 1.

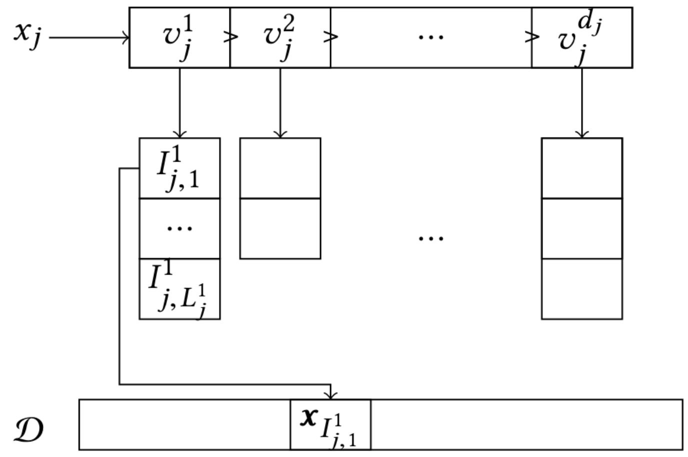


Figure 1: Structure to store indices based on their value of feature j.

A more convenient way of storing indices for a certain feature x_j is shown in Fig. 2.

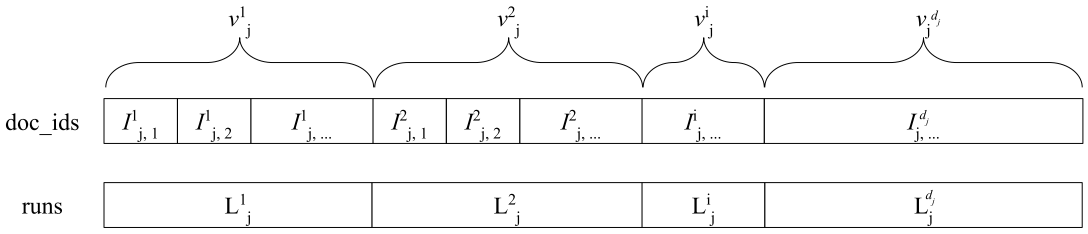


Figure 2: Proposed structure to sequentially store indices according to their value of feature x_j.

As we can see, there is no need to have a separated structure of arbitrary length to store all indices of value v_j^1 of feature x_j, but instead we sequentially store all indices ordered as in the first case. The difference is that we store the number of indices belonging to the same value, for example L_j^1 is the number of documents having value v_j^1, which are stored in the first L_j^1 positions of array **doc_ids**. They are followed by the documents having the second largest value v_j^2 which are going to occupy the next L_j^2 positions of the array. And so on.

This structure is going to help in the calculations, since it is more cache friendly (it's a simple vector of integers). Which means that it should favour spatial locality.

This poses the problem that we can't divide the workload in different threads, since the algorithm is based on sequential assumptions. That is, we sequentially move documents as the considered cut becomes smaller. So we can't give a thread the job of calculating the gain starting from value

v^i_j, since it should first read the id of all documents having a larger value than v^i_j to calculate the division of documents to classes.

2.2 Dealing with empty leaves

While deciding cuts of the tree we might end up with some leaves of the tree being empty, which means that we can't assign a proper response value to that leave. This problem can be mitigated by passing to such leaves the value of their father. This seems to be the more suitable choice since assigning any other value would mean corrupting the result in some sense.

3 Parallel Version

As anticipated in the introduction, the work done on this project as far as concerns the parallelization, is brought by means of multi-thread implementation. In particular, we are going to discuss parts of the code which are elegible for parallelization, that are:

- **Finding best feature**: To find the next cut of the current decision table we have to test possible cuts on each feature, trying all of them. We can think of giving each thread a specific feature to check for the best gain.
- **Update after feature selection**: After finding the best feature-cut couple, we have to update the actual leave in which objects reside, that is to apply the new cut.
- **Update of residuals**: After adding a decision table to the model, we have to compute residuals for the next model training
- **Backfitting**: Backfitting strategies replace a specific cut of the tree by searching the most optimal feature-cut couple. So we can parallelize as we did for the search for the best feature.

3.1 Finding best feature (Gain)

The more time consuming job is that of calculating the best feature and the corresponding cut, which gives the best gain overall.

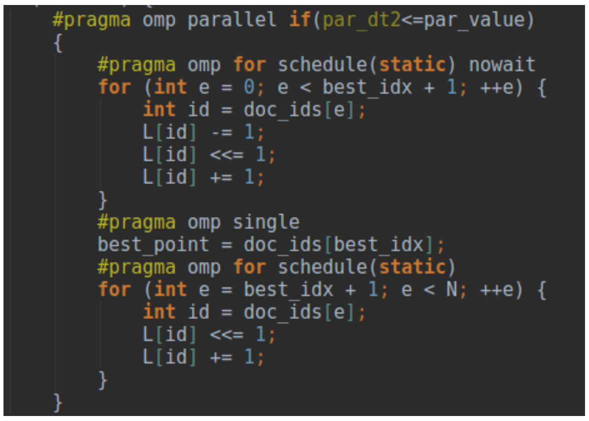
To check for the best value in a particular feature, we have to check all possible levels of that feature and for each of them, compute the split and calculate the gain. By itself, this operation is relatively not much time consuming, its complexity depends on the number of elements and on the possible levels for that feature.
Thinking of parallelizing the task at this level seems to be counterproductive, since we have to call this function as many times as there are features. So the algorithm would spend a considerable amount of time in creating threads, instead of using them for the actual computation.

In general, the number of features is expected to be considerably lower than the number of elements.
For that reason, in order to divide the job in different sub-tasks, we can look for the best combination for each feature separately. That is, we can give each thread the task of computing the best gain for a specified chunk of features.
The structure proposed in Fig.2 is useful to ease the calculations. We expect to have features with varying number of levels, graphically, in Fig.2 we expect to have a varying number of runs (d_j). So we expect to have a different workload for different features. This suggests to not distribute features equally because it is likely to have some threads finishing before others. In order to push for a more balanced work, we should prefer the usage of a dynamic assignment of features

to threads. In particular, by making use of **OpenMP**, we can use the dynamic scheduling of for cicles to distribute the work on different chunks and threads will ask for the next chunk to execute as soon as they become idle.

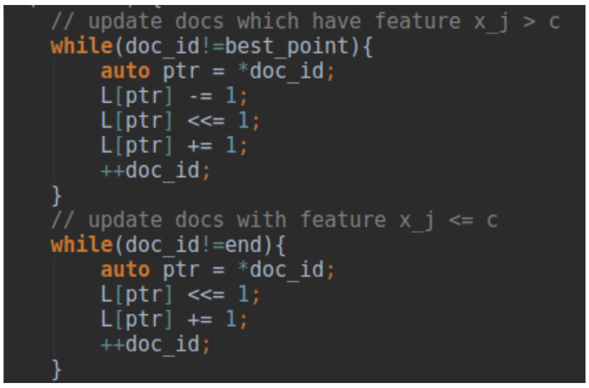
3.2 Update after feature selection

As noted previously, after finding the actual best cut, we have to update the leave membership for all items. The calculations are eased by the fact that for each feature, we have an array containing document ids sorted by feature value (array **doc_ids** in Fig.2). So, when updating the leave membership, we can easily distinguish documents having a value lower than the cut from the others. The task becomes more time consuming because we use these ids to access the vector containing leave memberships in a not sequential order, which makes it hard to use cache locality.



```
#pragma omp parallel if(par_dt2<=par_value)
{
    #pragma omp for schedule(static) nowait
    for (int e = 0; e < best_idx + 1; ++e) {
        int id = doc_ids[e];
        L[id] -= 1;
        L[id] <<= 1;
        L[id] += 1;
    }
    #pragma omp single
    best_point = doc_ids[best_idx];
    #pragma omp for schedule(static)
    for (int e = best_idx + 1; e < N; ++e) {
        int id = doc_ids[e];
        L[id] <<= 1;
        L[id] += 1;
    }
}
```

(a) Parallel code



```
// update docs which have feature x_j > c
while(doc_id!=best_point){
    auto ptr = *doc_id;
    L[ptr] -= 1;
    L[ptr] <<= 1;
    L[ptr] += 1;
    ++doc_id;
}
// update docs with feature x_j <= c
while(doc_id!=end){
    auto ptr = *doc_id;
    L[ptr] <<= 1;
    L[ptr] += 1;
    ++doc_id;
}
```

(b) Sequential code

Figure 3: The code for updating leave membership of all items, based on the optimal cut just found. (a) parallelized code; (b) sequential code with pointer access.

We can think of parallelizing the job, by assigning each thread the task of taking a chunk of these ids and doing the calculations for them only. Given the fact that the workload for each chunk is expected to be equally distributed, we can use the OpenMP static scheduling clause. A code snippet is shown in Fig. 3(a) for the parallel version, while in 3(b) we can see the sequential code to achieve the same task.

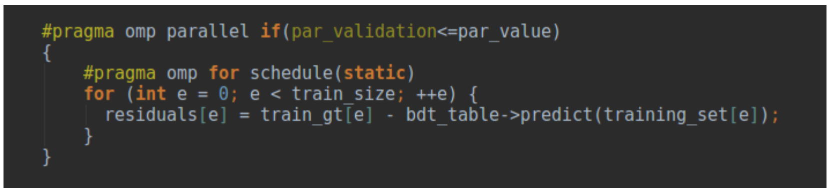
Without parallelization, the code consisting of 2 **for** loops is slightly slower than that one using **while** loops and pointer access. Simple experiments show that there is an improvement of 17% in execution time switching from the **for** version to the **while** one. But we would expect the first one to bring a good improvement in the parallel context, since it is a simple **for** loop, which indexes can be split among different threads.

It turns out that still with 32 threads, the **for** version is slower than the sequential **while** version. Numerical values are going to be given in Sec.4.
Reasons for that result can be found on the fact that the access to vector **L** still remains random and in the code shown in Fig.3(a) there is no sign to indicate to the compiler that accesses can be made independently, that is, that there is no concurrence in accessing **L**. Thus, threads are not accessing it concurrently, bringing the execution time to the same level as a sequential code. For the code in Fig.3(b) there is a slightly better sequential execution time since there is a memory pointer access to the vector containing doc ids, thus making one less memory request.

3.3 Update of residuals

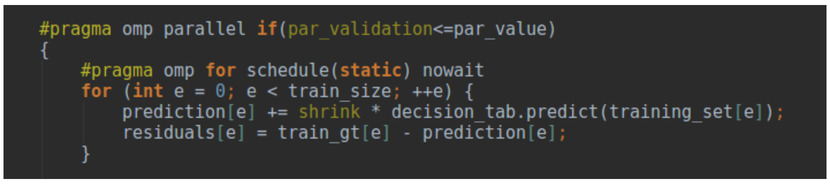
As we know, after adding a weak learner (decision table) to the model, we have to calculate the residuals for each element of the training set and train the next decision table upon these residuals.

For large datasets, even if the calculation is straightforward, some execution time is required for that computation.
Given the fact that this can be written in a simple **for** loop with invariant boundary values, and that vector containing the response (**train_gt**) is constant, a good performance is expected in the parallelized case. The code can be seen in Fig.4.



```
#pragma omp parallel if(par_validation<=par_value)
{
    #pragma omp for schedule(static)
    for (int e = 0; e < train_size; ++e) {
      residuals[e] = train_gt[e] - bdt_table->predict(training_set[e]);
    }
}
```

Figure 4: Code snippet needed to update residuals. First version.



```
#pragma omp parallel if(par_validation<=par_value)
{
    #pragma omp for schedule(static) nowait
    for (int e = 0; e < train_size; ++e) {
        prediction[e] += shrink * decision_tab.predict(training_set[e]);
        residuals[e] = train_gt[e] - prediction[e];
    }
```

Figure 5: Code snippet needed to update residuals. Last version.

In the parallel case, in a simple setup composed of a single machine with dual core, the parallel version turns out to be effective. Indeed a Speedup of 2.4 is achieved, as we can see from the results exposed in Fig.6, where the first bar (**0:Sequential**) is the sequential code exposed in Fig.4, while the second bar (**1:Parallel**) is its parallel execution.

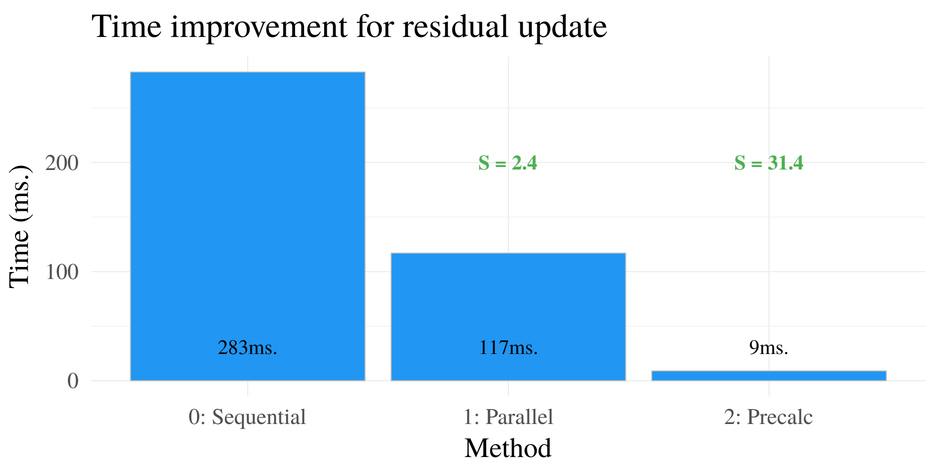


Figure 6: Execution time for updating residuals.

The last improvement to this part of the code is represented in the third bar (**2:Precalc**), which consists in having a vector containing the actual predictions. We can see the code in Fig.5.

Instead of calling the model everytime, we can compute residuals and keep a vector of the actual prediction that our model does, which we update as soon as a new decision table is added to the model. This refinement brings a Speedup of **31.4** on the same dual core machine.

Probably the compiler is not able to inline the call to **bdt→predict** shown in the code in Fig.4, so it is not processed as a single value per each execution of the line. So, it has to create the environment in order for the function call to be executed and only at the end return the value and close the external environment created. While in this version the calculation is trivial and the call to **decision_tab.predict** is probably inlined since it is simple.

3.4 Backfitting

Backfitting is a considerable part of the overall execution time. The same observations as in the case of finding the best cut (gain) holds also for backfitting. Indeed, after removing a particular cut, we have to search all possible features for the best a posteriori cut. The same idea of parallelization is applied here too. That is, the loop on all features is parallelized by assigning each thread a chunk of them. **Dynamic scheduling** offered by **OpenMP** has been used, which means that a thread asks for the next chunk as soon as he finishes with the actual one. This is brought by the fact that we don't know how many levels for each feature we have, so we don't know the workload for each feature.

4 Experiments

Dataset: Million Song Dataset (MSD)

This dataset [2] is the largest one taken into consideration to stress the algorithm to work on *big data*. The original dataset consists of 1 million songs, thus the name, but the one used in the experiments is composed of nearly half of them and it is retrieved from UCI repository at `https://archive.ics.uci.edu/ml/datasets/yearpredictionmsd`.

- **Number of instances**: 515'345
- **Number of attributes**: 90
  - 12 attributes: timbre average
  - 78 attributes: timbre covariance
- **Response**: year in which the song was produced (1922-2011)

The following experiments are done with parameters:

- Depth of decision trees $d = 4$ ($2^4 = 16$ leaves)
- Number of tables $t = 20$
- Number of processors/threads $p = 32$

The results are computed as the mean of 10 runs, by shuffling the data for each run.

- Train size 64% ($329'822$)
- Validation size 16%
- Test size 20%

4.1 Parallelization effect

The first plot shows how different levels of parallelization affect the execution time. We can see it in Fig. 7. The first column is for the sequential case, while the others:

- **1: par_dt**: parallelization of the procedure for searching the best cut, as explained in Sec 3.1
- **2: par_bt**: parallelization of backfitting strategy as discussed in Sec 3.4
- **3: par_val**: parallel code for residual updating, Sec 3.3
- **4: par_upd**: parallel code to update predictions of the tree, after adding a cut
- **5: par_dt2**: the parallel code in Sec 3.2

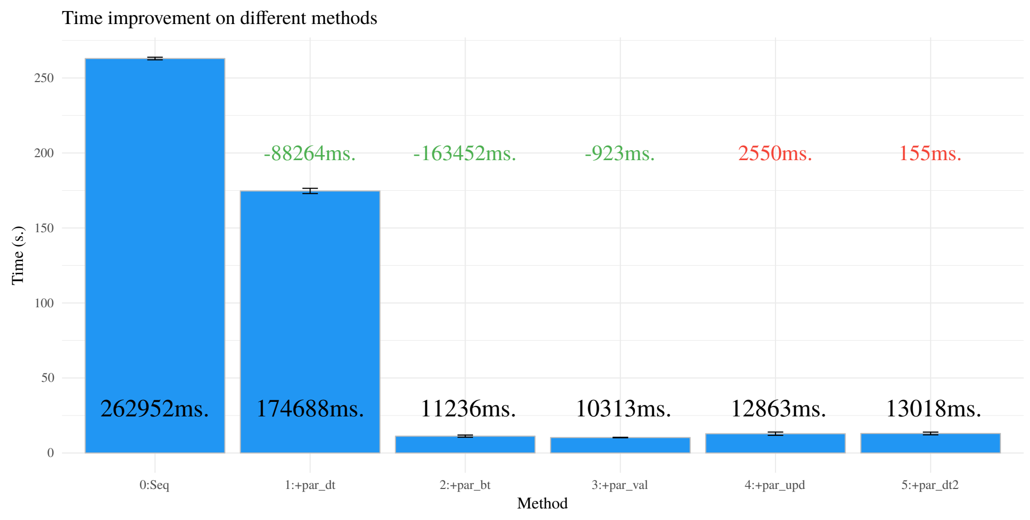


Figure 7: Execution time with standard deviation, using different levels of parallelization. Improvement in green, worsening in red.

As we can clearly see from the barplot, there is a significant improvement in parallelizing the search for the best (feature, cut) couple to add to the actual decision table, indeed it brings a **88264ms.** improvement. Same goes for the parallelization of the backfitting procedure, which simply applies the same parallelization technique for searching the best cut at each level of the tree. In this case, there is an additional **163452ms.** improvement. The other parts of the code bring a relatively small improvement when executed on 32 processors. Actually, the last two (4:*par_upd*, 5:*par_dt2*) lead to a small worsening with respect to method 3:*par_val*, which is the reason for which we should stop at this parallelization level. It is probably due to the fact that the portion of code parallelized in these last 2 methods includes simple instrictions on many data points, that when parallelized become too fast to be executed, thus the overhead for thread creation becomes a factor. Numerically, if we have to process 300'000 instances, when split among 32 threads, each thread has to process less than 10'000 entries. Which is too few for simple tasks.

We can better notice this effect by looking at the speedup.

In terms of Speedup we achieve:

- ◇ **1: par_dt**: **1.5**
- ◇ **2: par_bt**: **23.4**
- ◇ **3: par_val**: **25.5**
- ◇ **4: par_upd**: **20.4**
- ◇ **5: par_dt2**: **20.2**

which again suggests using the first 3 methods, achieving a Speedup of **25.5**, and avoid the last 2 which would lead to a worsening down to **20.4**. On the rest of experiments the first 3 methods are used.

4.2 Speedup based on threads

With the same experimental setup described above, we analyse what happens as the number of threads varies.

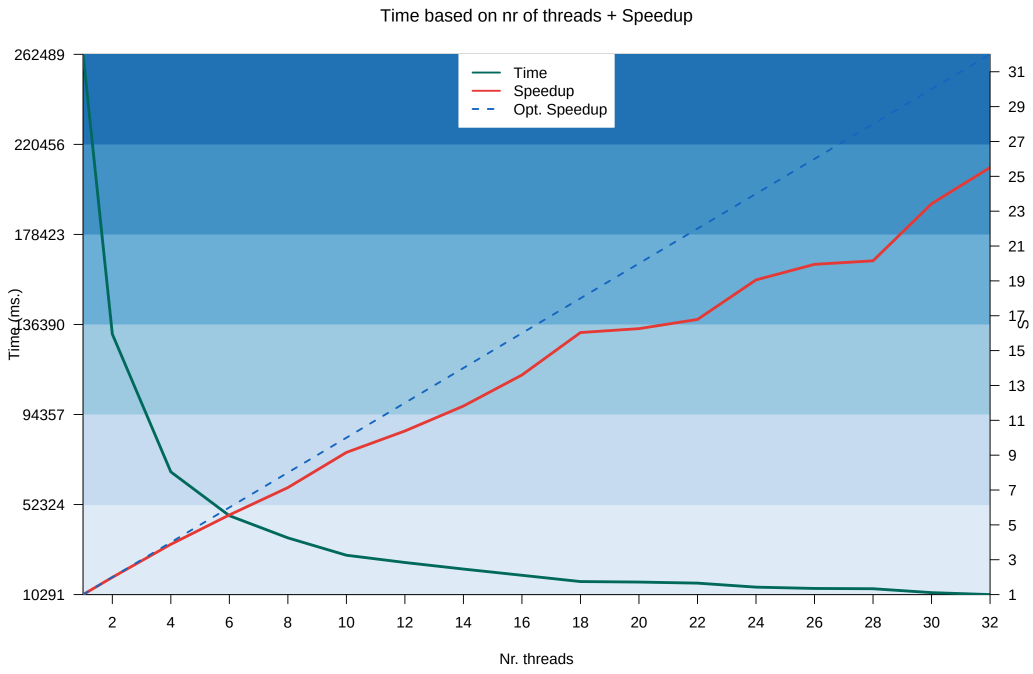


Figure 8: Execution time (left axis) and Speeup (right axis) as the number of threads varies.

As we would expect, the execution time decreases as the number of processing units grows. In Fig. 8 we can also note how Speedup tends to be really close to the optimal one for $2 \leq p \leq 18$, where the optimal Speedup is given by T_S/p, also called Linear Speedup.
For values $p > 18$ we note how it tends to grow slower, and it starts getting away from the optimal line.

From the analysis brought in Sec.4.1 we understood that the most effective improvement is given by the parallelization of the computations of gain on features. Computing such gain for a single feature has a complexity $\mathcal{O}(n)$ dominated by the number of items in the dataset, since we have to test all possible cuts by checking how MSE varies by moving instances.

Having $p = 22$ means that we are distributing the computation of all 90 features on 22 processing units. That is, 4 features per thread. Moreover, given the fact that **dynamic scheduling** is used, we can think of a thread having less than 3 features to elaborate and ask for the next chunk once finished. This might imply that we are reaching a **Fine-grained parallelism**, that is, we are distributing the job among many threads, which can be counterproductive. It might result in the overhead being more impactful on the overall time. To give a better idea, some simple experiments showed that the computation time for a single feature is lower than **70ms.** on the local machine. Keeping in mind that the machines on which experiments have been carried out are way faster, we can expect a thread to complete the task in approx. **100ms.** before asking for the next chunk. So the time needed for thread management becomes influencial.

This means that we would probably have better results for bigger datasets or higher number of features.

4.3 Accuracy of the model

The following experiment shows what happens to the accuracy of the model as the number of tables grows.

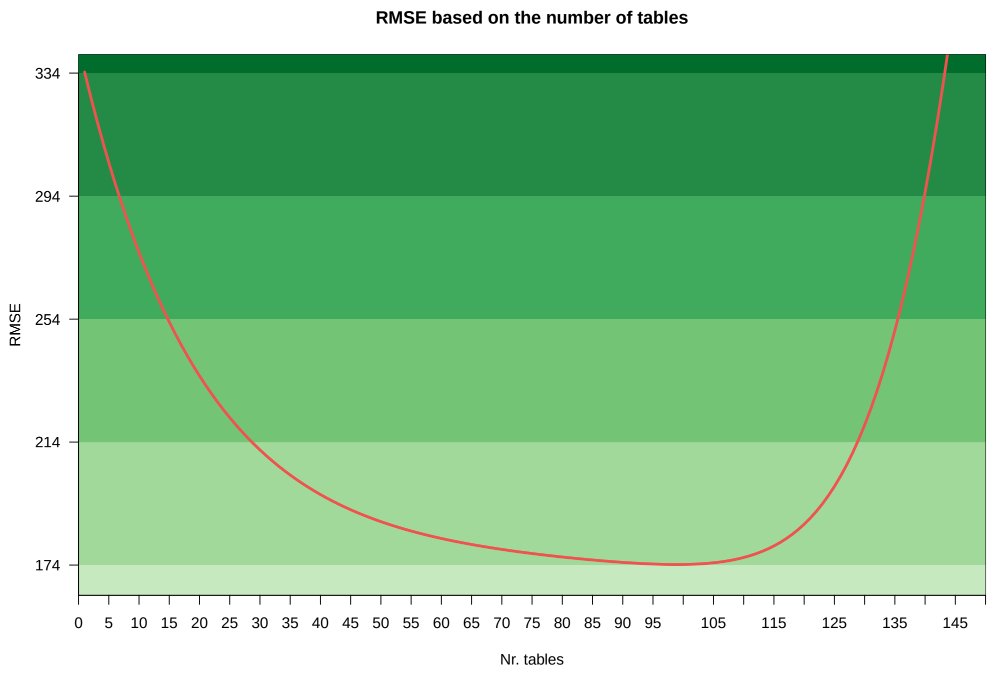


Figure 9: RMSE variation based on the number of decision tables added to the model

In Fig. 9 we can see the experimental results showing the standardized Root Mean Square Error (RMSE) on a model of growing dimension, with the same setup as before, testing on approx. $100'000$ instances, where the response is a discrete variable indicating the year of production of songs, ranging $1922 - 2011$, rounding operation has not been applied to the response of the model.

As we would expect, the error decreases as the number of decision tables added to the model grows, indeed we start from an initial rmse of 334 and end up by halving it when we reach $t = 100$ tables. For succesive added complexity, the model starts to overfit, and looses the generalization ability, ending up with an error increasing considerably.

4.4 Testing time

We can also take a look at the execution time for predicting a new value.
The code is expected to be faster. Since the execution time of the prediction of a single instance is less than $1ms$. for a not so large number of tables, it seems unreasonable to parallelize the predicting function itself. Instead, we would better choose to parallelize the call to the function, that is, in our case we have more than $100'000$ instances to test, we can divide them into chunks and statically assign them to threads.

The following experiment is run on 16 threads. Still 10 runs are considered, but in this case the number of tables varies. Running times are shown for the prediction of the entire test set.

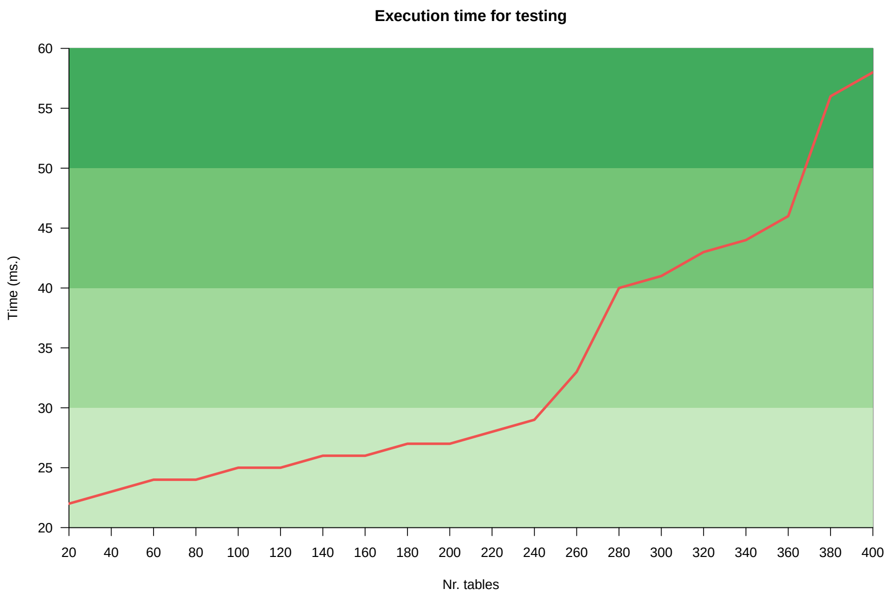


Figure 10: Testing time on testset, using 16 threads.

In Fig.10 we note how the execution time depends on the number of tables of the model. Execution is particularly fast, indeed we spend **58ms.** to test all 103'068 instances of the test set on a model composed of 400 tables.
Nothing particular to discuss here.

4.5 Parallelization limit

Recalling that T_s is the sequential time and T_p the parallel part executed on p copies, the Amdahl's Law states that:

$$S = \frac{T_s}{T_p} = \frac{T_s}{fT_p + (1-f)\frac{T_s}{p}} = \frac{1}{f + \frac{(1-f)}{p}}$$

$$S \leq \frac{1}{f}$$

Ending up with the fact that Speedup S is upperbounded by a fraction $1/f$ where f is the fraction of the code which is sequential.

We can use this result to get an estimate of the fraction of code which is not parallelized. That is, given the fact that empirical analysis suggests that Speedup reaches at most a factor of approx. **25** (see Fig.11), we can combine it with the formula $S \leq 1/f$, ending up with

$$f = 0.04$$

This means that approximately 4% of the code is not parallelized.

We should keep in mind the final discussion carried out in Sec.4.2, that is, we might have reached the point in which the dataset size (or number of features) is not large enough to saturate the usage of that many working units.
Thus, the stabilization of speedup seen in Fig.11 might also be affected by this factor.

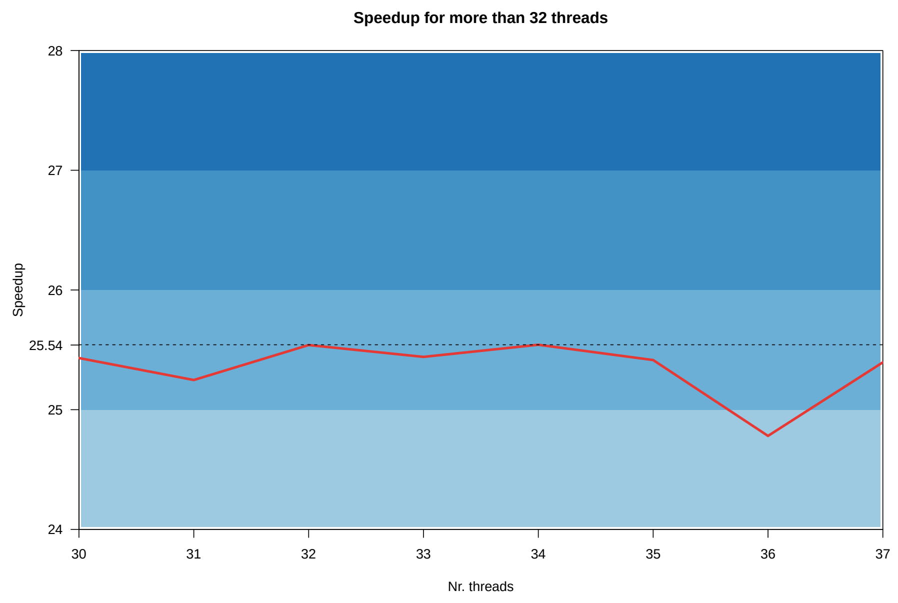


Figure 11: Dark side of the plot.

5 Future Work

There are some adjustements which can be added to the project to take care of some particular problems which are not essential for the purposes of this project target.

A problem whith the actual structure may arise in cases of contineous features which would cause the evaluation of almost all possible splits of the data for that particular feature. This is due to the fact that probably each document has a different value for that feature, since it is contineous, thus the vector **runs** is going to have N entries equal to 1. Upon evaluation, taking all possible cuts, means taking all $N - 1$ possible splits of documents. This would make the algorithm more time consuming.

A trivial fix is to group chunks of values together, making it a discete variable. We may think about fixed range groups, whose range is related to the values present for that feature.

In order to have a more balanced split of the workload upon calculation of the best cut, when all features have to be taken into consideration, we can think of computing some statistics about the levels that features posses. We can use it to make a better split of the work among threads.

The current algorithm doesn't stop until the maximum number of tables is reached. Only in the end it considers the optimal number of tables, based on calculations carried out on the validation set after each table added to the model.

An experiment on increasing number of instances can be carried out.
We can find a larger dataset on which we can make some experiments to analyse what happens as $n \to \infty$.

References

[1] Yin Lou and Mikhail Obukhov. Bdt: Gradient boosted decision tables for high accuracy and scoring efficiency. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1893–1901. ACM, 2017.

[2] Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. The million song dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.