A better method to enforce monotonic constraints in regression and classification trees

Auguste, Charles Smirnov, Ivan Malory, Sean
Grassl, Thomas

July 24, 2019

Abstract

In this report we present two new ways of enforcing monotone constraints in regression and classification trees. One yields better results than the current LightGBM, and has a similar computation time. The other one yields even better results, but is much slower than the current LightGBM. We also propose a heuristic that takes into account that greedily splitting a tree by choosing a monotone split with respect to its immediate gain is far from optimal. Then, we compare the results with the current implementation of the constraints in the LightGBM library, using the well known Adult public dataset. Throughout the report, we mostly focus on the implementation of our methods that we made for the LightGBM library, even though they are general and could be implemented in any regression or classification tree.

The best method we propose (a smarter way to split the tree coupled to a penalization of monotone splits) consistently beats the current implementation of LightGBM. With small or average trees, the loss reduction can be as high as 1% in the early stages of training and decreases to around 0.1% at the loss peak for the Adult dataset. The results would be even better with larger trees. In our experiments, we didn't do a lot of tuning of the regularization parameters, and we wouldn't be surprised to see that increasing the performance of our methods on test sets.

1 Results

1.1 Summary of the important results

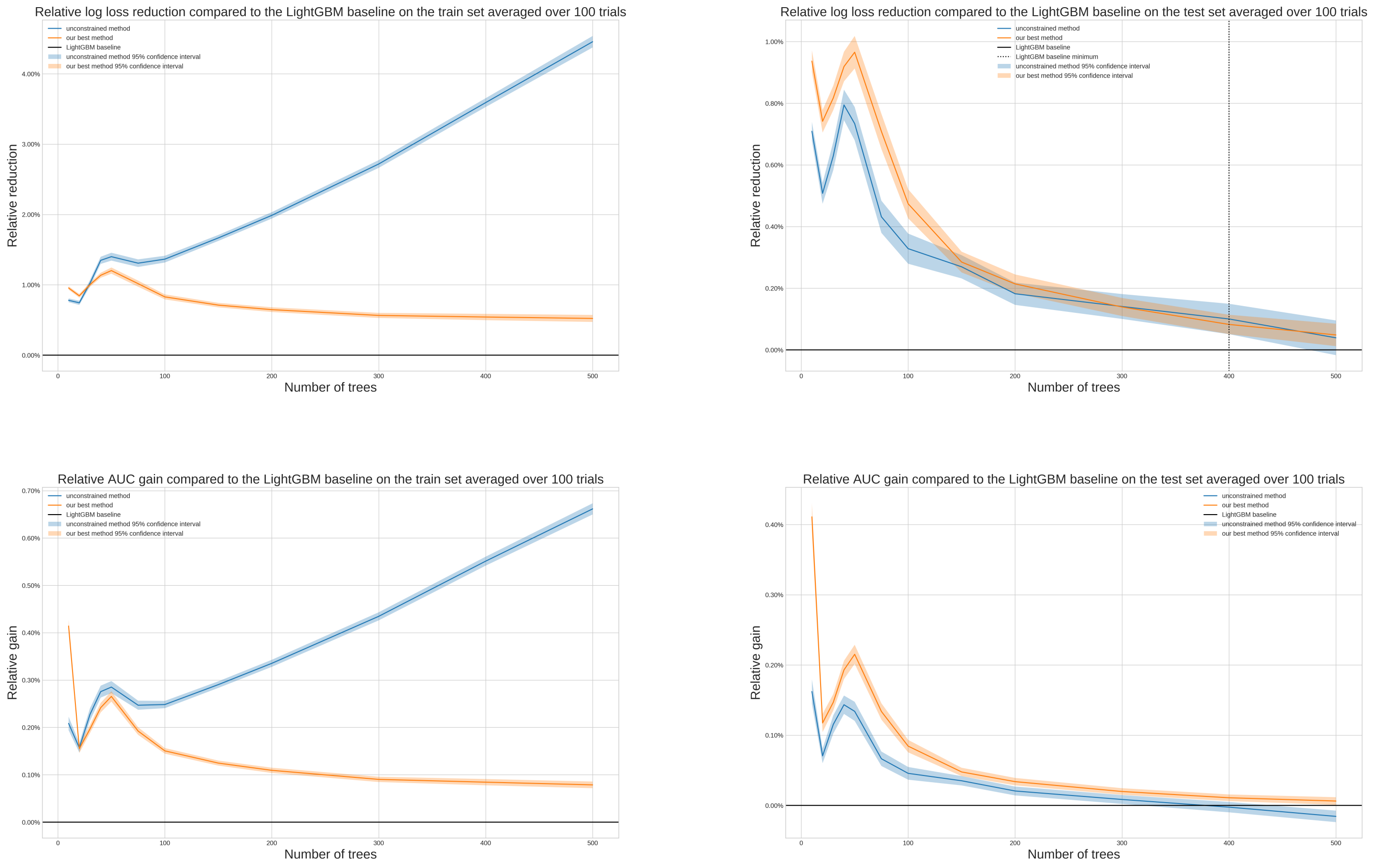


Figure 1: Loss and AUC vs. number of iterations relative to the constrained LightGBM baseline, for our best method and the unconstrained method, on the train and test sets

On figure 1 are represented the most important results of the report. Overall, our best method generates a consistent and significant loss reduction, which also helps us improve common metrics. Therefore we think our method should become the standard for enforcing monotone constraints, and replace the current LightGBM method.

1.2 Dataset and preprocessing

The Adult dataset is a well known public dataset extracted by Barry Becker from the 1994 Census database [1]. The task of this dataset is classification. We need to classify individuals depending on whether or not they earn more than 50,000$ a year. The dataset is fairly imbalanced as for around 75% of entries, people earn less than 50,000$ a year (which doesn't mean that around 75% of people earned less than 50,000$ in 1994, because entries have to be weighted by the survey weights). If a person earns more than 50,000$ a year, its label will be 1. Otherwise, it will be 0.

Description of the data The description of the data is extremely important as we need to know which variable are locally monotone. Table 1 summarizes for each variable its type (continuous or discrete) and if the labels should be locally monotone with respect to it.

Variable	Type	Labels' relationship to the variable
`age`	continuous	Monotonically increasing
`workclass`	discrete	None
`fnlwgt`	continuous	None
`education`	discrete	None
`education_num`	continuous	Monotonically increasing
`marital_status`	discrete	None
`occupation`	discrete	None
`relationship`	discrete	None
`race`	discrete	None
`sex`	discrete	None
`capital_gain`	continuous	None
`capital_loss`	continuous	None
`hours_per_week`	continuous	Monotonically increasing
`native_country`	discrete	None

Table 1: Summary of the Adult dataset variables

Treatment of the data The variables `education` and `education_num` are bijective. We chose to remove the `education` variable as the labels are monotonically increasing with respect to the other one. Then, we chose to one-hot encode all the categorical variables.

In order to test the different algorithms, de decided to perform Monte-Carlo cross validation (or repeated random sub-sampling validation). For each exper-

[1] Adult data set Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

iment, we perform N trials where we randomly split the set into a training set and a testing set. The results are then averaged across all the trials. The ratios we are using are shown on table 2.

Set	Ratio	Number of data points
Training set	65%	31747
Testing set	35%	17095
Total	100%	48842

Table 2: Split ratio of the data into training and testing

Parameters used The parameters we used for our experiments were set as stated in table 3. The remaining parameters not mentioned in table 3 were set to LightGBM's default values[2]. Finally, we set the variables `age`, `education_num` and `hours_per_weeks` to be monotonically increasing. We decided to perform gradient boosting. We picked the specific depth 5 to have trees that wouldn't overfit too much, otherwise it would make our new methods perform better than what they actually do on the training set and worse on the test set (because they will be able to overfit better as they are less "constraining").

Parameter	Value
objective	`"binary"`
boosting	`"gbdt"`
num_leaves	32
depth	5
min_data_in_leaf	100
seed	42
bagging_seed	42
feature_fraction_seed	42
drop_seed	42
data_random_seed	42

Table 3: Parameters chosen

1.3 Loss and metrics vs. iterations

The unconstrained method represents an upper bound of the achievable performance on the training set (we cannot learn better with constraints than without). We use the current constrained LightGBM implementation as a baseline and plot our results according to it.

[2]LightGBM parameters

On figure 2, it can be seen that our new methods learn more efficiently than the LightGBM baseline and consistently beat it on the training set for the loss and for all metrics. On the testing set, we also mostly beat the baseline. However, with many iterations we start to overfit, and therefore some metrics can get below the baseline, but this could be offset by tuning the regularization parameters of the gradient boosting scheme.

On these figures, it seems like the slow method is not that useful as a combination of the fast method and penalization achieves almost the same results. However, for different datasets, and different parameters, especially if we were trying to build bigger trees, then the slow method could achieve significantly better results.

1.4 Stability of the loss for with respect to the penalization parameter

On figure 3, it can be observed that our heuristic penalty is able to reduce the loss when it is set correctly. Precise tuning is not required as the results seem pretty stable, and the improvement is consistent (even though it is offset by the boosting effect). However, one must be careful to not set the penalization parameter too high, otherwise, it may prevent any monotone split in the whole tree. Figure 3 only depicts the results for the fast method. However, the slow method yields very similar results.

1.5 Computation time for each method

On figure 4, we report the computational cost for each method. We measured it using the `timeit` module in Python (so the results are not very precise, as the standard deviation can be as high as 1ms). These statistics were created using either 2000, 10000, or close to 50000 random entries. The execution times were measured over 1 boosting iteration, using a maximum depth of 10 and a maximum number of leaves of 40, and the trees were filled with the maximum number of leaves every time. The results are averaged across 5000 runs.

Even though these results are a bit noisy, we can draw very useful insights from them. Overall, whether we are using LightGBM or our new fast method, the difference in computational costs is not very significant. The slow method is, however, much slower than the current LightGBM. Nonetheless, the difference mitigates when the dataset is bigger and building histogram is a more important task.

1.6 Plots of the first trees generated by each method

On figures 5, 6, 7, 8, 9 and 10, we plotted the first trees generated by our different methods to see how they would differ. Here are a few things to notice to understand the tree visualizations,

- The trees are all compared to the unconstrained tree. A red leaf means that the leaf does not appear in the unconstrained tree. A red node means that the node does not appear in the unconstrained tree (either the splitting feature is different or the threshold is different, or the parent of the node is different). A red gain means that the gain for the same node in the unconstrained tree is different;
- A green node means that the node is monotonically increasing;
- Blue numbers on the nodes show the order in which the nodes have been split.

Here are a few observations coming from the comparison of the trees,

- As anticipated, the monotone features are extremely important, because they appear in many nodes in the first trees;
- Up to a certain point (until the first monotone split), all trees are similar to the unconstrained tree;
- For the penalized method, we can indeed see that no monotone split happened in the first levels of the tree;
- The outputs of the first tree are all negative because the dataset is imbalanced and there are a lot more 0's than 1's;
- The trees get even more different as the algorithms keep running;
- Gains for the unconstrained method seem generally greater than for the rest of the methods, which makes sense. Also, the unconstrained method has more leaves on this example, which is also coherent.

2 Explanations of the new methods

2.1 New ways of constraining trees

We developed 2 new ways of constraining trees. Both yield better results than the current LightGBM constraining method. One of the method is about as fast as the current LightGBM. The other method is much slower (can be as much as twice as slow in extreme worst case scenarii), but is also more accurate. In this section, we will present both.

Both methods originally stem from the observation that the current LightGBM constraining method is over-constraining the trees. In our methods, when we make a monotone split, instead of imposing constraints on both children that make the outputs of all their descendants mutually exclusive, as LightGBM currently does by setting a mid point, we chose to set up only the necessary constraints every time so that the next split would not break the monotone rules.

More precisely, this means that if the split is monotonically increasing, then the left child gets the right output as a maximum constraint, and the right child gets the left output as a minimum constraint. However, is the split is monotonically decreasing, then it means that the left child gets the right output as a minimum constraint and the right child gets the left output as a maximum constraint constraint.

Then, when we make any split (monotone or not) in a branch having a monotone node as a parent somewhere, after making the split, we need to check that the new outputs are not violating any constraint on other leaves of the tree. The general idea is that we should start from the node where a split was just made, go up the tree, and every time a monotone node is encountered, we should go down in the opposite branch and check that the constraints and the new outputs from the new split are compatible. If they are not, then the constraints need to be updated. Therefore making a split in a branch can very well update the constraints of other leaves in another branch.

2.1.1 The computationally efficient way of constraining trees

In this method, only one minimum constraint and one maximum constraint need to be stored for each leaf. A constraint is computed as the extremum of all constraints applied sequentially on a leaf during the construction of the tree. That makes the constraints very easy to compute, because when a constraint of a leaf has to be updated, it can be done only by performing a minimum or a maximum. This method is better than the current LightGBM because it doesn't create "gaps" between leaves, and it is still efficient, because the only additional thing we need is going through the tree and recomputing some split sometimes, but usually not that much (especially when we compare it to the time spent building histograms).

2.1.2 The slower yet more accurate way of constraining the trees

This method is based on the same principle except that for each leaf, for each feature, for each threshold, we are going to store one minimum and one maximum constraints. By doing that, we are going to have many different constraints for every leaf (at most number of features times number of thresholds times 2 for minimum and maximum constraints per leaf, but usually, it is quite a lot less than that). Then when a split has to be made with respect to a feature, we can have different left and right constraints for the children depending on where the constraints were applied on the original leaf. Additionally, when a split is made, and we start updating the constraints of other leaves, we take into account the fact that the split may very well have "unconstrained" some leaves (as well as constrained others). By doing that we are sometimes able to split leaves that would not be splittable using the previous methods. Moreover, since we compute the exact constraints needed everywhere at every iteration of building the trees, we are not over-constraining any leaf, and therefore trees can get much more accurate. However, when using this method, we need to store lots of constraints that are going to be updated very often. Furthermore, when a leaf is "unconstrained", then all the constraints need to be computed from the beginning (otherwise we would need to store which leaves the constraints come from, and keep them sorted, which, in our opinion, would not be better). Because of that, this method is quite a lot slower, even though we tried to implement many optimizations so that it would remain tractable.

2.2 Theoretical example

On figure 11, we imagined a simple situation demonstrating the improvements of our methods. Let's imagine that we have a regression problem, with the data being represented on figure 11a. The only constraint is that the output has to be monotonically increasing horizontally. The true labels of the points are given on the right of figure 11a. On figure 11a, we take it as given that the first split is the same with all methods and that it yields the values written on the graph. On figure 11b, we assume that the second split is again the same with all methods and that it yields the values written on the graph. The 3 following figures depict what would happen for the following split with each method.

- On figure 11c, with LightGBM, since the values on the graph from figure 11a were 0.3 and 0.6, the mid point is 0.45. Therefore, the blue dots are upper bounded and labelled as 0.45.
- On figure 11d, with the fast method, once the first split is made, the leaf on the left is given an upper bound of 0.6. Once the second split is made, every leaf on the left is given an upper bound of 0.5. Therefore, the blue dots are labelled as 0.5
- On figure 11e, with the slow method, once the first split is made, the leaf on the left is given an upper bound of 0.6. Once the second split is made, the top part of the left leaf is given an upper bound of 0.5, and the lower part of the left leaf is given an upper bound of 0.8. Therefore, the blue dots are labelled as 0.7, their true value.

We can therefore see that our methods are better at this specific regression problem, the slow method achieving optimal results with only one tree. Similar examples happen very often both in regression and classification tasks, when building trees. Therefore, our methods should consistently yield better results than the current LightGBM implementation.

2.3 A heuristic penalty for monotone splits

2.3.1 Why penalizing monotone splits is a good idea

The main drawback of adding monotone constraints to a tree is that it may end up over-constraining the tree because of the greediness of the tree splitting algorithm, and therefore loose in predictive power. By building a tree in a greedy fashion, we don't know how much the constraints of monotone splits are going to affect the following nodes (it may reduce the gain of the children by a lot). For our new methods, we don't know either by how by much a new split is going to constrain other surrounding leaves and their future children (in the event that the leaf we are splitting is part of a branch growing from a monotone node).

When using our new method, even though it already yields good results as we will see in section 1, both these effect happen in almost any tree with reasonable depth when making at least one monotone split. However, we do not quantify or take into account any of these 2 drawbacks at all in the algorithm. Yet they can be huge at the first levels of the trees, because the constraints are going to affect most of the leaves. Therefore, amongst other things, we are very likely to overestimate the gain generated by splitting a leaf on a monotone feature; especially early in the trees, where the constraints are going to impact many leaves.

Unfortunately, this reduction of gain can't be computed greedily. Indeed, we would need to know what is going to happen after to quantify it (or we could try to predict it but that is not an easy task either). However, we were able to develop a simple heuristic that improves the results, by penalizing monotone splits early when building trees.

The general idea of the heuristic is that we are likely to not want to make a monotone split in the first levels of a tree because this is going to constrain too many leaves. Then, generally, the deeper we go, the more inclined to monotone spits we are. This is a very general rule than can have many exceptions, but we believe it is true more often than not. Therefore, we designed a penalty function that is a function of depth, that will multiply the gain of monotone splits at a given depth. It is parametrised by only one parameter that we will call γ here for simplicity, and that allows monotone splits to be penalized the higher they appear in a tree.

2.3.2 Penalization formula and usage

The final penalty p that will multiply he gain of a node depends on the tunable parameter γ and on the depth of the node in the tree d, depth 0 being the root. So we have $0 \leq d \leq$ max depth. The penalty is computed according to the following formula,

$$p = \begin{cases} 0 & \text{if } \gamma \geq d+1 \\ 1 - \frac{\gamma}{2^d} & \text{if } \gamma \leq 1 \\ 1 - 2^{\gamma-1-d} & \text{else} \end{cases}$$

Moreover, when using the LightGBM framework, we add an extremely small ϵ to p, because only splits with a strictly positive gain can be performed, and in the event that we would only have monotone splits available, we still want split one of them.

On figure 12 we plotted what the penalty looks like for different penalization parameters, as a function of depth. There are some important things to mention about this penalty function:

- When $\gamma = 0$, then there is no penalty at all;
- When $\gamma \in [0; 1]$, then we gradually penalize all depths, and the closer to 1 they are, the more they are penalized;
- When $\gamma \in [1; 2]$, then the penalty for the first level is 0 (we prohibit any monotone split on this level), and we keep gradually penalizing monotone splits at all depths, and the closer to depth 2 they are, the more they are penalized;
- When $\gamma \in [2; 3]$, then the penalty for the first two levels is 0 (we prohibit any monotone split on these levels), and we keep gradually penalizing monotone splits at all depths, and the closer to depth 3 they are, the more they are penalized;
- ...

Therefore, the parameter γ is a very intuitive parameter that shouldn't confuse the end user.

2.3.3 Penalization vs. depth plot

On figure 12, we plot the penalty factor that will multiply the gain of monotone splits, for different depths and penalization parameters.

3 Conclusion

In this report, we propose two new methods to enforce monotonic constraints in regression and classification trees, as well as a heuristic to improve the results. During our tests, we found that our new methods consistently achieve better results than the current LightGBM constraining method. Our fastest method does not generate significant additional computational cost compared to the current LightGBM. Therefore, we believe that our work should replace the current LightGBM implementation. Finally, as mentioned previously, our work is not LightGBM-specific and could be implemented in any regression or classification tree.

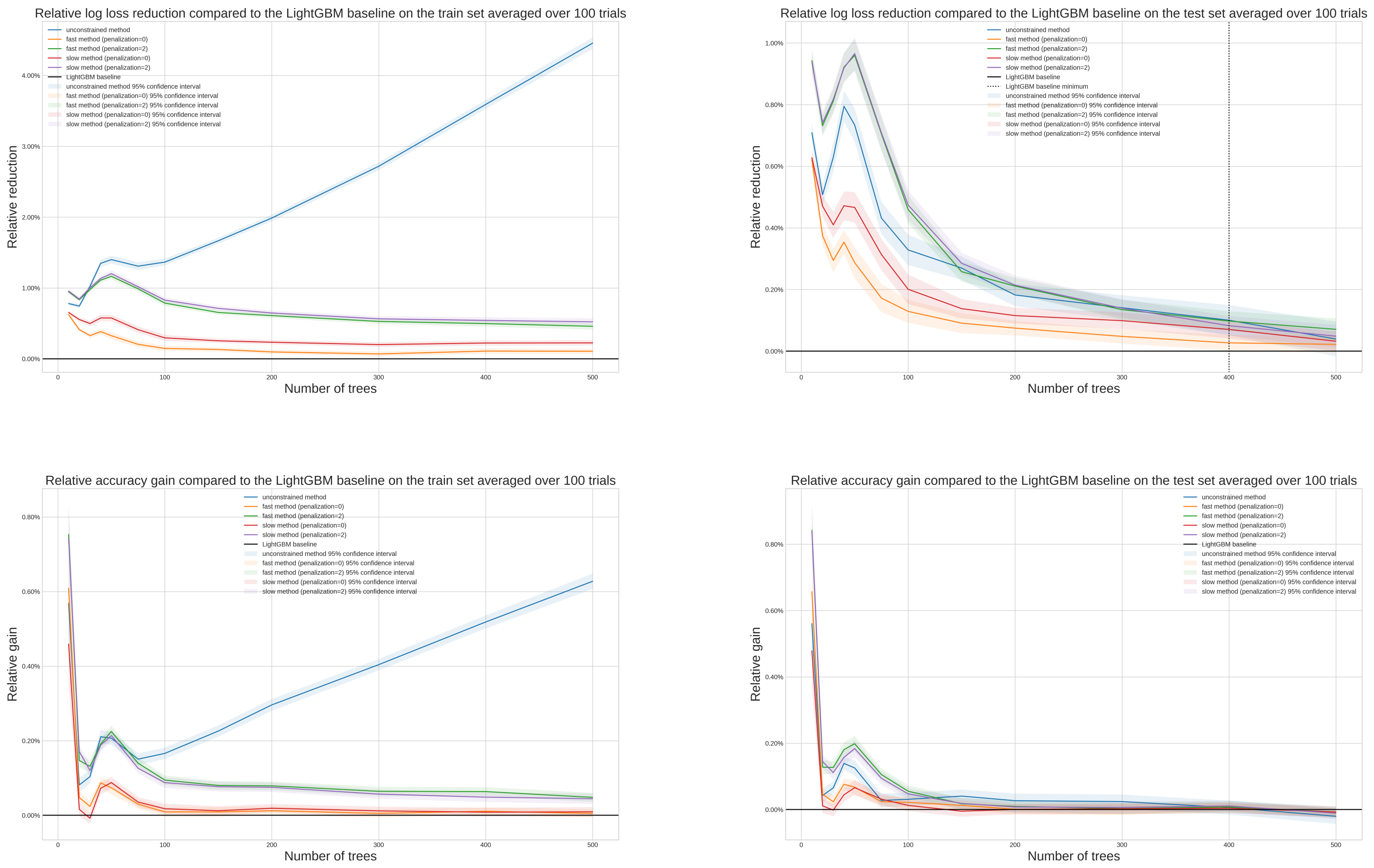


Figure 2: Loss and metrics vs. number of iterations relative to the constrained LightGBM baseline, for all methods, on the train and test sets

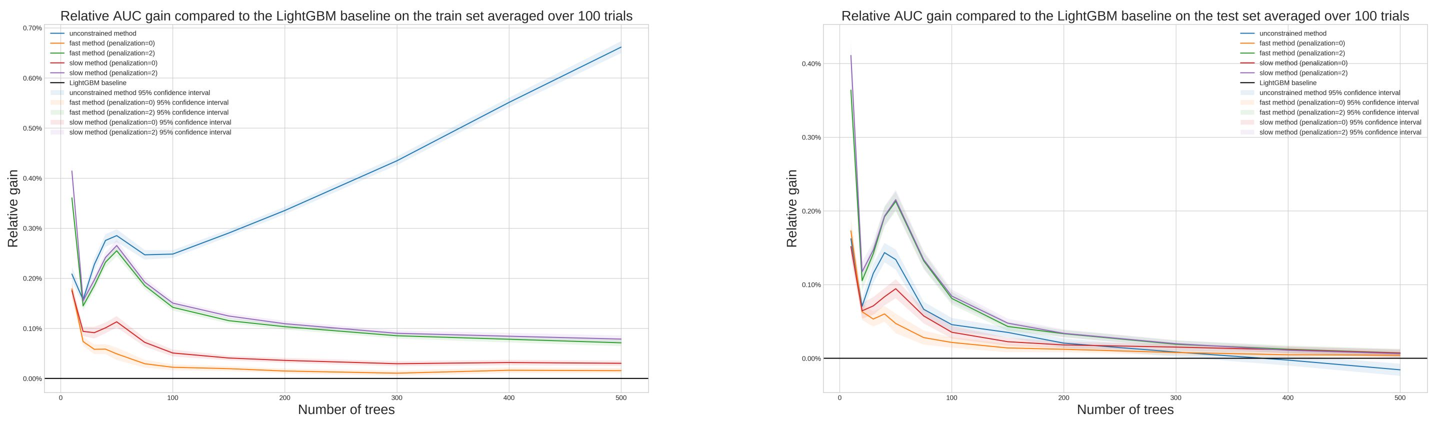


Figure 2: Loss and metrics vs. number of iterations relative to the constrained LightGBM baseline, for all methods, on the train and test sets

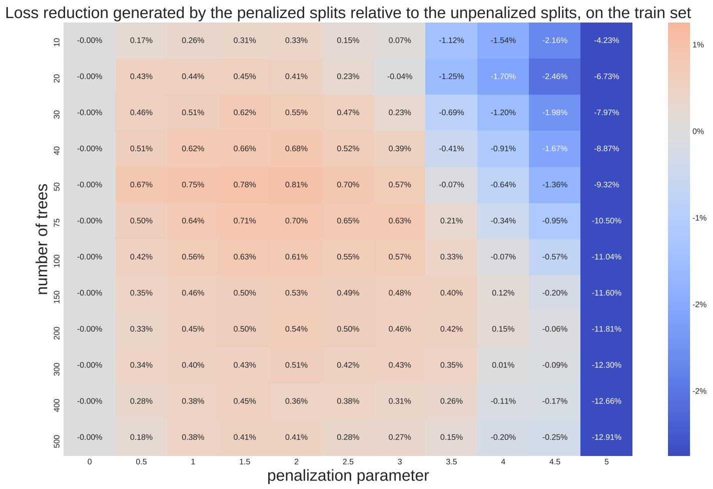


Figure 3: Relative loss reduction for penalized monotone splits, compared to unpenalized splits, for different penalization parameters, fot the fast method

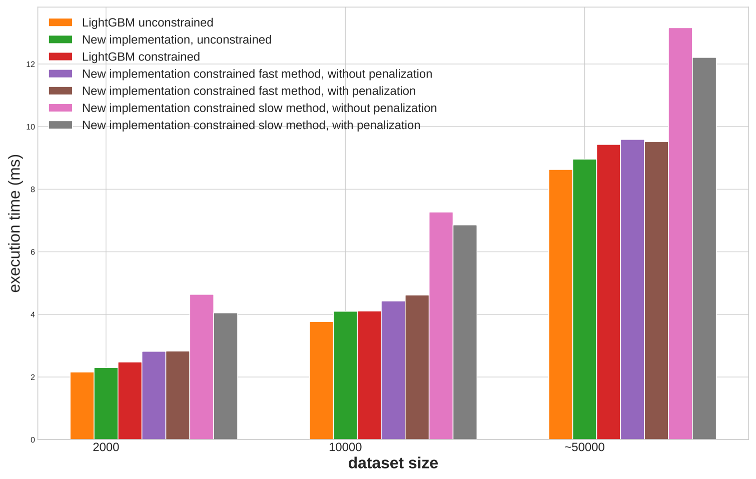


Figure 4: Mean execution time of the different methods for different dataset sizes averaged over 5000 runs

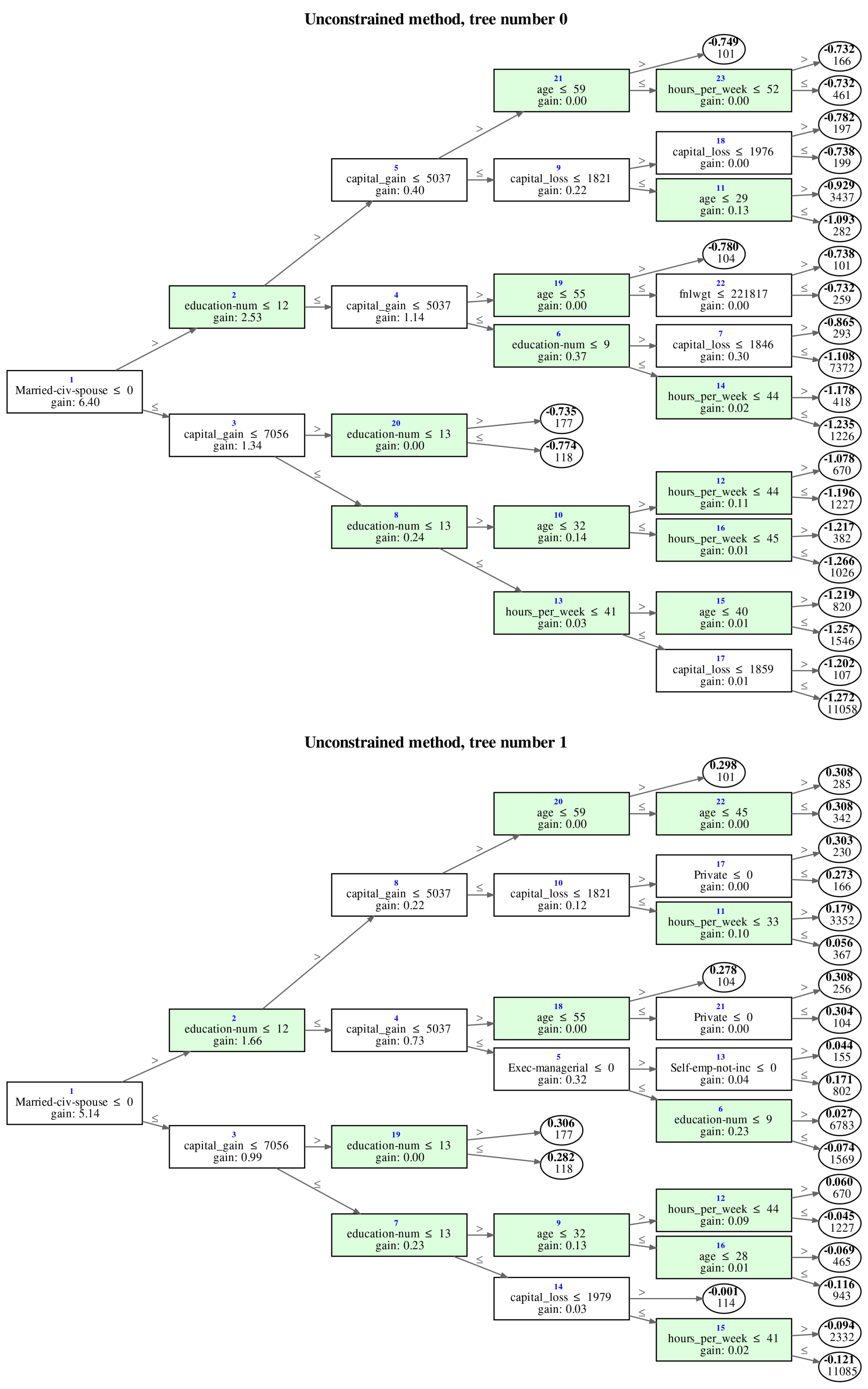

Unconstrained method, tree number 0
1 Married-civ-spouse ≤ 0 gain: 6.40
2 education-num ≤ 12 gain: 2.53
3 capital_gain ≤ 7056 gain: 1.34
4 capital_gain ≤ 5037 gain: 1.14
5 capital_gain ≤ 5037 gain: 0.40
6 education-num ≤ 9 gain: 0.37
7 capital_loss ≤ 1846 gain: 0.30
8 education-num ≤ 13 gain: 0.24
9 capital_loss ≤ 1821 gain: 0.22
10 age ≤ 32 gain: 0.14
11 age ≤ 29 gain: 0.13
12 hours_per_week ≤ 44 gain: 0.11
13 hours_per_week ≤ 41 gain: 0.03
14 hours_per_week ≤ 44 gain: 0.02
15 age ≤ 40 gain: 0.01
16 hours_per_week ≤ 45 gain: 0.01
17 capital_loss ≤ 1859 gain: 0.01
18 capital_loss ≤ 1976 gain: 0.00
19 age ≤ 55 gain: 0.00
20 education-num ≤ 13 gain: 0.00
21 age ≤ 59 gain: 0.00
22 fnlwgt ≤ 221817 gain: 0.00
23 hours_per_week ≤ 52 gain: 0.00
-0.749 101
-0.732 166
-0.732 461
-0.782 197
-0.738 199
-0.929 3437
-1.093 282
-0.780 104
-0.738 101
-0.732 259
-0.865 293
-1.108 7372
-1.178 418
-1.235 1226
-0.735 177
-0.774 118
-1.078 670
-1.196 1227
-1.217 382
-1.266 1026
-1.219 820
-1.257 1546
-1.202 107
-1.272 11058
Unconstrained method, tree number 1
1 Married-civ-spouse ≤ 0 gain: 5.14
2 education-num ≤ 12 gain: 1.66
3 capital_gain ≤ 7056 gain: 0.99
4 capital_gain ≤ 5037 gain: 0.73
5 Exec-managerial ≤ 0 gain: 0.32
6 education-num ≤ 9 gain: 0.23
7 education-num ≤ 13 gain: 0.23
8 capital_gain ≤ 5037 gain: 0.22
9 age ≤ 32 gain: 0.13
10 capital_loss ≤ 1821 gain: 0.12
11 hours_per_week ≤ 33 gain: 0.10
12 hours_per_week ≤ 44 gain: 0.09
13 Self-emp-not-inc ≤ 0 gain: 0.04
14 capital_loss ≤ 1979 gain: 0.03
15 hours_per_week ≤ 41 gain: 0.02
16 age ≤ 28 gain: 0.01
17 Private ≤ 0 gain: 0.00
18 age ≤ 55 gain: 0.00
19 education-num ≤ 13 gain: 0.00
20 age ≤ 59 gain: 0.00
21 Private ≤ 0 gain: 0.00
22 age ≤ 45 gain: 0.00
0.298 101
0.308 285
0.308 342
0.303 230
0.273 166
0.179 3352
0.056 367
0.278 104
0.308 256
0.304 104
0.044 155
0.171 802
0.027 6783
-0.074 1569
0.306 177
0.282 118
0.060 670
-0.045 1227
-0.069 465
-0.116 943
-0.001 114
-0.094 2332
-0.121 11085
>
≤

Figure 5: First two trees generated by the unconstrained method

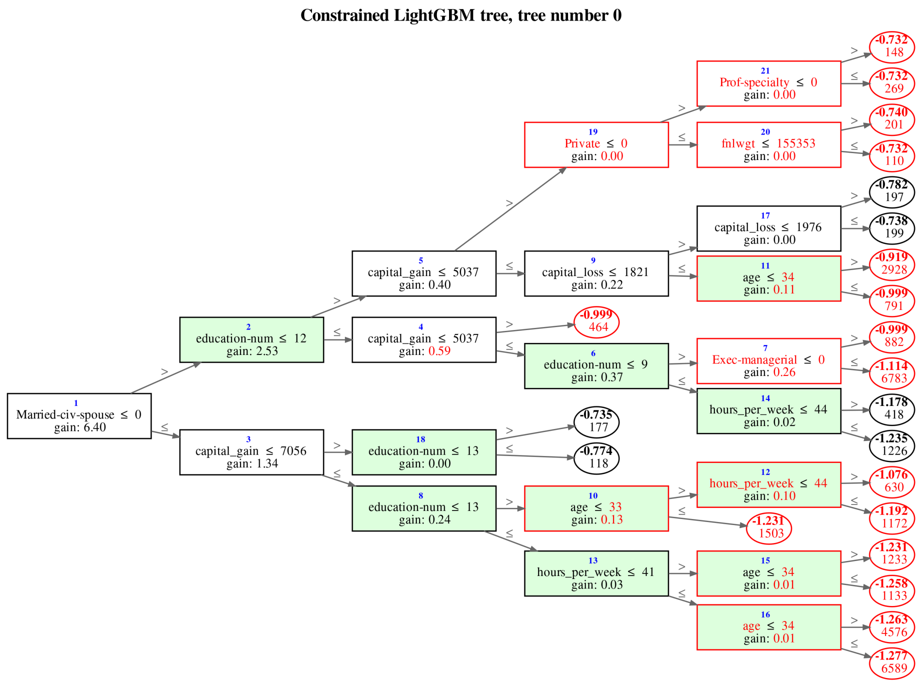
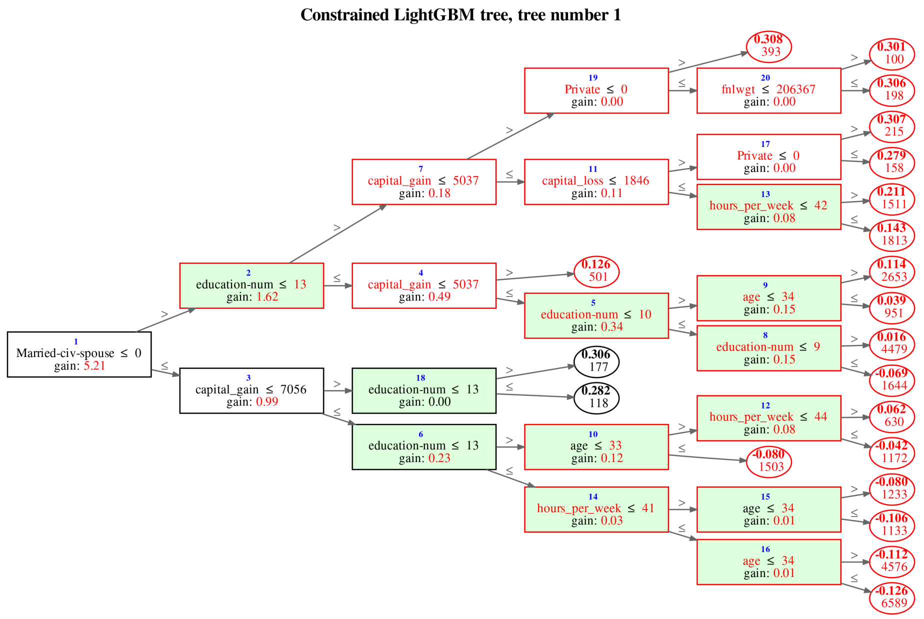
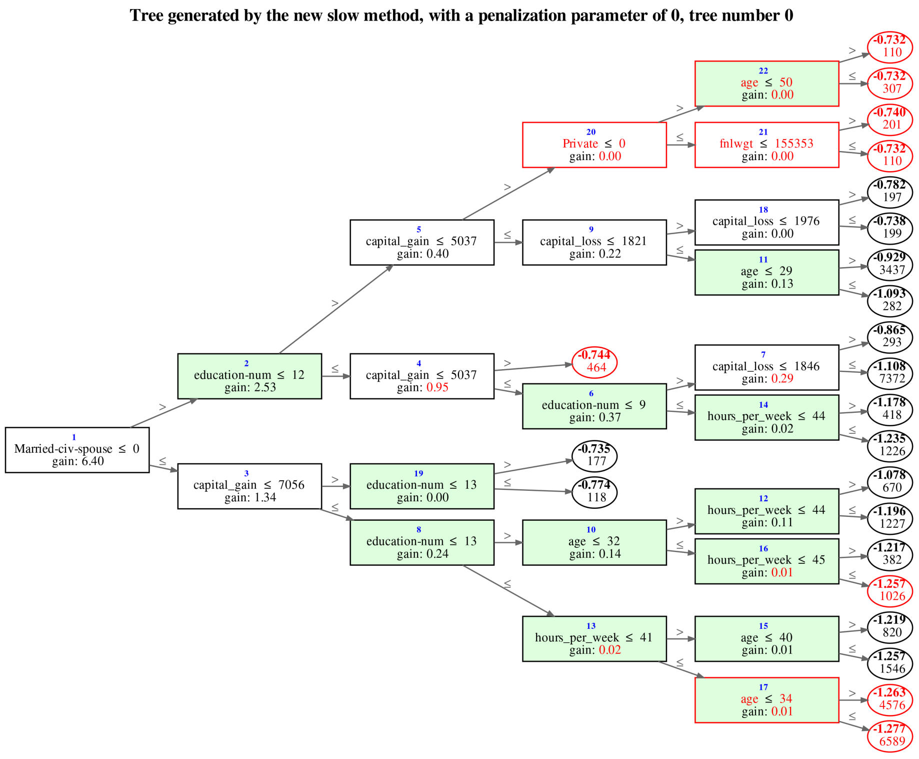

Constrained LightGBM tree, tree number 0
1 Married-civ-spouse ≤ 0 gain: 6.40
2 education-num ≤ 12 gain: 2.53
3 capital_gain ≤ 7056 gain: 1.34
5 capital_gain ≤ 5037 gain: 0.40
4 capital_gain ≤ 5037 gain: 0.59
18 education-num ≤ 13 gain: 0.00
8 education-num ≤ 13 gain: 0.24
19 Private ≤ 0 gain: 0.00
9 capital_loss ≤ 1821 gain: 0.22
6 education-num ≤ 9 gain: 0.37
10 age ≤ 33 gain: 0.13
13 hours_per_week ≤ 41 gain: 0.03
21 Prof-specialty ≤ 0 gain: 0.00
20 fnlwgt ≤ 155353 gain: 0.00
17 capital_loss ≤ 1976 gain: 0.00
11 age ≤ 34 gain: 0.11
7 Exec-managerial ≤ 0 gain: 0.26
14 hours_per_week ≤ 44 gain: 0.02
12 hours_per_week ≤ 44 gain: 0.10
15 age ≤ 34 gain: 0.01
16 age ≤ 34 gain: 0.01
-0.732 148
-0.732 269
-0.740 201
-0.732 110
-0.782 197
-0.738 199
-0.919 2928
-0.999 791
-0.999 464
-0.999 882
-1.114 6783
-1.178 418
-1.235 1226
-0.735 177
-0.774 118
-1.076 630
-1.192 1172
-1.231 1503
-1.231 1233
-1.258 1133
-1.263 4576
-1.277 6589

Constrained LightGBM tree, tree number 1
1 Married-civ-spouse ≤ 0 gain: 5.21
2 education-num ≤ 13 gain: 1.62
3 capital_gain ≤ 7056 gain: 0.99
7 capital_gain ≤ 5037 gain: 0.18
4 capital_gain ≤ 5037 gain: 0.49
18 education-num ≤ 13 gain: 0.00
6 education-num ≤ 13 gain: 0.23
19 Private ≤ 0 gain: 0.00
11 capital_loss ≤ 1846 gain: 0.11
5 education-num ≤ 10 gain: 0.34
10 age ≤ 33 gain: 0.12
14 hours_per_week ≤ 41 gain: 0.03
20 fnlwgt ≤ 206367 gain: 0.00
17 Private ≤ 0 gain: 0.00
13 hours_per_week ≤ 42 gain: 0.08
9 age ≤ 34 gain: 0.15
8 education-num ≤ 9 gain: 0.15
12 hours_per_week ≤ 44 gain: 0.08
15 age ≤ 34 gain: 0.01
16 age ≤ 34 gain: 0.01
0.308 393
0.301 100
0.306 198
0.307 215
0.279 158
0.211 1511
0.143 1813
0.126 501
0.114 2653
0.039 951
0.016 4479
-0.069 1644
0.306 177
0.282 118
0.062 630
-0.042 1172
-0.080 1503
-0.080 1233
-0.106 1133
-0.112 4576
-0.126 6589

Figure 6: First two trees generated by the baseline LightGBM method


Tree generated by the new slow method, with a penalization parameter of 0, tree number 0
1 Married-civ-spouse ≤ 0 gain: 6.40
2 education-num ≤ 12 gain: 2.53
3 capital_gain ≤ 7056 gain: 1.34
5 capital_gain ≤ 5037 gain: 0.40
4 capital_gain ≤ 5037 gain: 0.95
19 education-num ≤ 13 gain: 0.00
8 education-num ≤ 13 gain: 0.24
20 Private ≤ 0 gain: 0.00
9 capital_loss ≤ 1821 gain: 0.22
-0.744 464
6 education-num ≤ 9 gain: 0.37
-0.735 177
-0.774 118
10 age ≤ 32 gain: 0.14
13 hours_per_week ≤ 41 gain: 0.02
22 age ≤ 50 gain: 0.00
21 fnlwgt ≤ 155353 gain: 0.00
18 capital_loss ≤ 1976 gain: 0.00
11 age ≤ 29 gain: 0.13
7 capital_loss ≤ 1846 gain: 0.29
14 hours_per_week ≤ 44 gain: 0.02
12 hours_per_week ≤ 44 gain: 0.11
16 hours_per_week ≤ 45 gain: 0.01
15 age ≤ 40 gain: 0.01
17 age ≤ 34 gain: 0.01
-0.732 110
-0.732 307
-0.740 201
-0.732 110
-0.782 197
-0.738 199
-0.929 3437
-1.093 282
-0.865 293
-1.108 7372
-1.178 418
-1.235 1226
-1.078 670
-1.196 1227
-1.217 382
-1.257 1026
-1.219 820
-1.257 1546
-1.263 4576
-1.277 6589

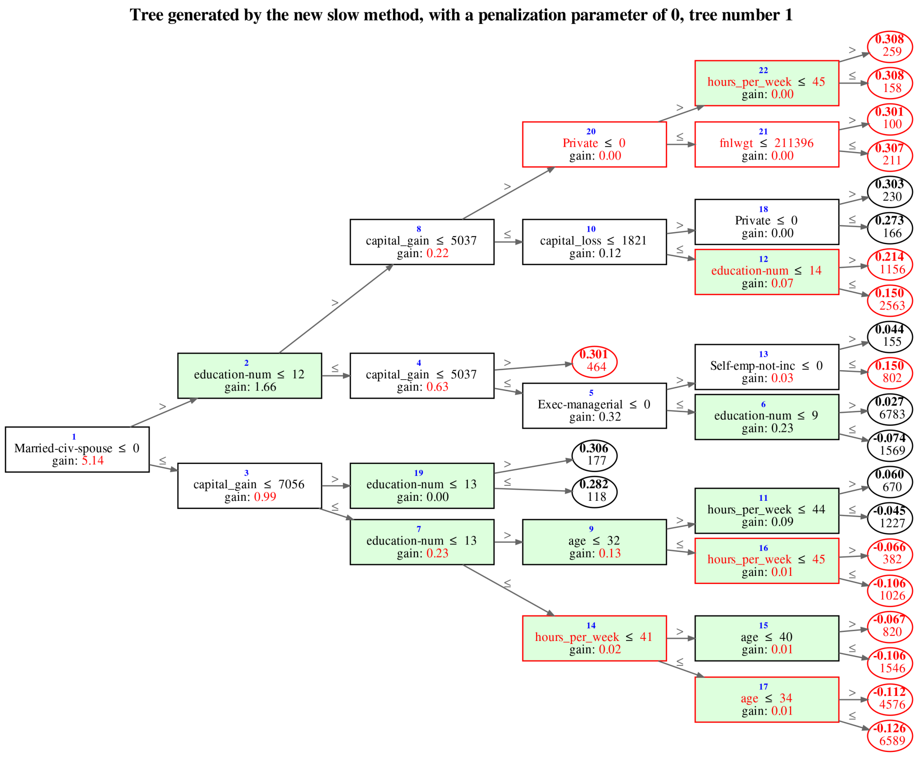
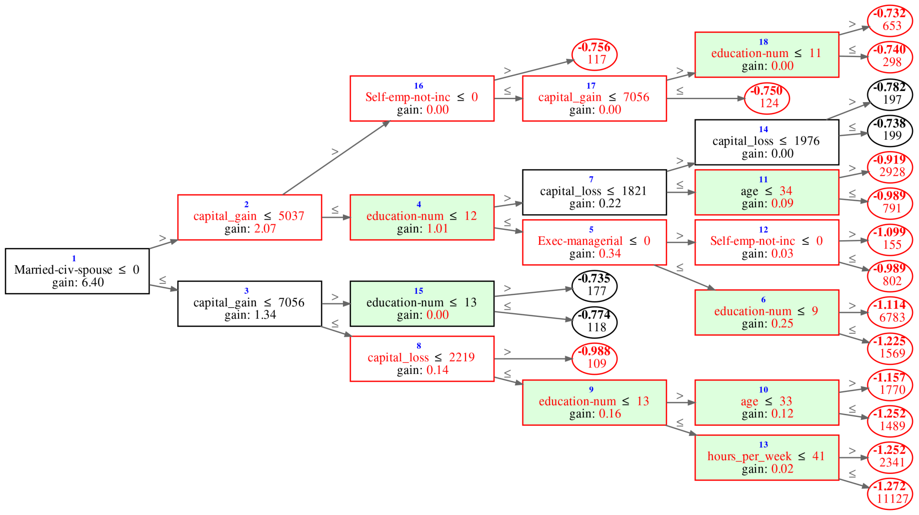

Tree generated by the new slow method, with a penalization parameter of 0, tree number 1
1 Married-civ-spouse ≤ 0 gain: 5.14
2 education-num ≤ 12 gain: 1.66
3 capital_gain ≤ 7056 gain: 0.99
8 capital_gain ≤ 5037 gain: 0.22
4 capital_gain ≤ 5037 gain: 0.63
19 education-num ≤ 13 gain: 0.00
7 education-num ≤ 13 gain: 0.23
20 Private ≤ 0 gain: 0.00
10 capital_loss ≤ 1821 gain: 0.12
0.301 464
5 Exec-managerial ≤ 0 gain: 0.32
0.306 177
0.282 118
9 age ≤ 32 gain: 0.13
14 hours_per_week ≤ 41 gain: 0.02
22 hours_per_week ≤ 45 gain: 0.00
21 fnlwgt ≤ 211396 gain: 0.00
18 Private ≤ 0 gain: 0.00
12 education-num ≤ 14 gain: 0.07
13 Self-emp-not-inc ≤ 0 gain: 0.03
6 education-num ≤ 9 gain: 0.23
11 hours_per_week ≤ 44 gain: 0.09
16 hours_per_week ≤ 45 gain: 0.01
15 age ≤ 40 gain: 0.01
17 age ≤ 34 gain: 0.01
0.308 259
0.308 158
0.301 100
0.307 211
0.303 230
0.273 166
0.214 1156
0.150 2563
0.044 155
0.150 802
0.027 6783
-0.074 1569
0.060 670
-0.045 1227
-0.066 382
-0.106 1026
-0.067 820
-0.106 1546
-0.112 4576
-0.126 6589

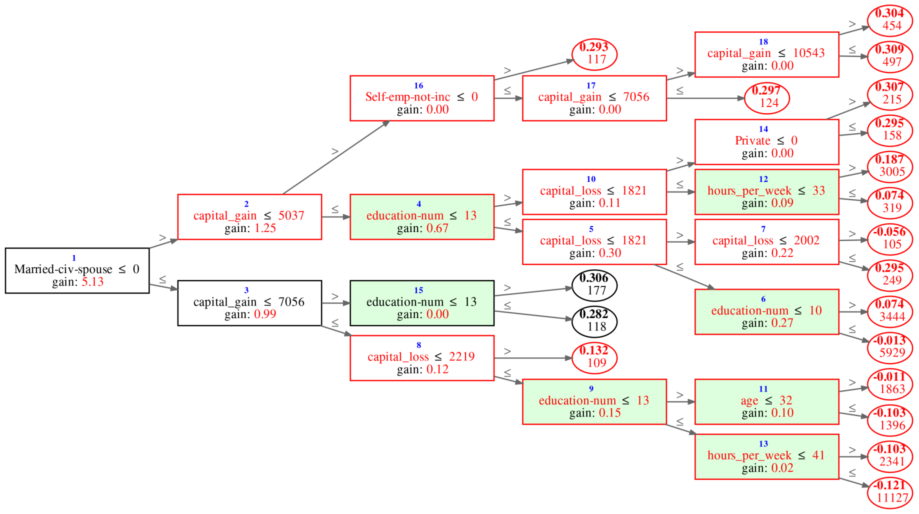
Figure 7: First two trees generated by our new slow method, with no penalization

Tree generated by the new slow method, with a penalization parameter of 2, tree number 0



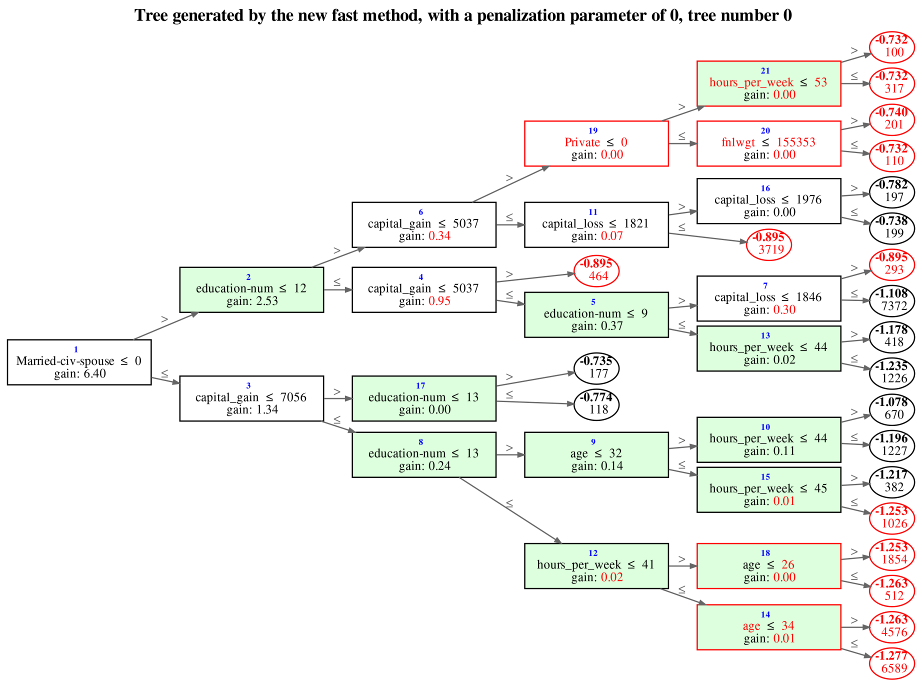
1 Married-civ-spouse ≤ 0 gain: 6.40
2 capital_gain ≤ 5037 gain: 2.07
3 capital_gain ≤ 7056 gain: 1.34
16 Self-emp-not-inc ≤ 0 gain: 0.00
4 education-num ≤ 12 gain: 1.01
15 education-num ≤ 13 gain: 0.00
8 capital_loss ≤ 2219 gain: 0.14
17 capital_gain ≤ 7056 gain: 0.00
7 capital_loss ≤ 1821 gain: 0.22
5 Exec-managerial ≤ 0 gain: 0.34
9 education-num ≤ 13 gain: 0.16
18 education-num ≤ 11 gain: 0.00
14 capital_loss ≤ 1976 gain: 0.00
11 age ≤ 34 gain: 0.09
12 Self-emp-not-inc ≤ 0 gain: 0.03
6 education-num ≤ 9 gain: 0.25
10 age ≤ 33 gain: 0.12
13 hours_per_week ≤ 41 gain: 0.02
-0.756 117
-0.750 124
-0.735 177
-0.774 118
-0.988 109
-0.732 653
-0.740 298
-0.782 197
-0.738 199
-0.919 2928
-0.989 791
-1.099 155
-0.989 802
-1.114 6783
-1.225 1569
-1.157 1770
-1.252 1489
-1.252 2341
-1.272 11127

Tree generated by the new slow method, with a penalization parameter of 2, tree number 1



1 Married-civ-spouse ≤ 0 gain: 5.13
2 capital_gain ≤ 5037 gain: 1.25
3 capital_gain ≤ 7056 gain: 0.99
16 Self-emp-not-inc ≤ 0 gain: 0.00
4 education-num ≤ 13 gain: 0.67
15 education-num ≤ 13 gain: 0.00
8 capital_loss ≤ 2219 gain: 0.12
17 capital_gain ≤ 7056 gain: 0.00
10 capital_loss ≤ 1821 gain: 0.11
5 capital_loss ≤ 1821 gain: 0.30
9 education-num ≤ 13 gain: 0.15
18 capital_gain ≤ 10543 gain: 0.00
14 Private ≤ 0 gain: 0.00
12 hours_per_week ≤ 33 gain: 0.09
7 capital_loss ≤ 2002 gain: 0.22
6 education-num ≤ 10 gain: 0.27
11 age ≤ 32 gain: 0.10
13 hours_per_week ≤ 41 gain: 0.02
0.293 117
0.297 124
0.306 177
0.282 118
0.132 109
0.304 454
0.309 497
0.307 215
0.295 158
0.187 3005
0.074 319
-0.056 105
0.295 249
0.074 3444
-0.013 5929
-0.011 1863
-0.103 1396
-0.103 2341
-0.121 11127

Figure 8: First two trees generated by our new slow method, with a penalization parameter of 2


Tree generated by the new fast method, with a penalization parameter of 0, tree number 0
1 Married-civ-spouse ≤ 0 gain: 6.40
2 education-num ≤ 12 gain: 2.53
3 capital_gain ≤ 7056 gain: 1.34
6 capital_gain ≤ 5037 gain: 0.34
4 capital_gain ≤ 5037 gain: 0.95
17 education-num ≤ 13 gain: 0.00
8 education-num ≤ 13 gain: 0.24
19 Private ≤ 0 gain: 0.00
11 capital_loss ≤ 1821 gain: 0.07
5 education-num ≤ 9 gain: 0.37
9 age ≤ 32 gain: 0.14
12 hours_per_week ≤ 41 gain: 0.02
21 hours_per_week ≤ 53 gain: 0.00
20 fnlwgt ≤ 155353 gain: 0.00
16 capital_loss ≤ 1976 gain: 0.00
7 capital_loss ≤ 1846 gain: 0.30
13 hours_per_week ≤ 44 gain: 0.02
10 hours_per_week ≤ 44 gain: 0.11
15 hours_per_week ≤ 45 gain: 0.01
18 age ≤ 26 gain: 0.00
14 age ≤ 34 gain: 0.01
-0.732 100
-0.732 317
-0.740 201
-0.732 110
-0.782 197
-0.738 199
-0.895 3719
-0.895 464
-0.895 293
-1.108 7372
-1.178 418
-1.235 1226
-0.735 177
-0.774 118
-1.078 670
-1.196 1227
-1.217 382
-1.253 1026
-1.253 1854
-1.263 512
-1.263 4576
-1.277 6589
>
≤

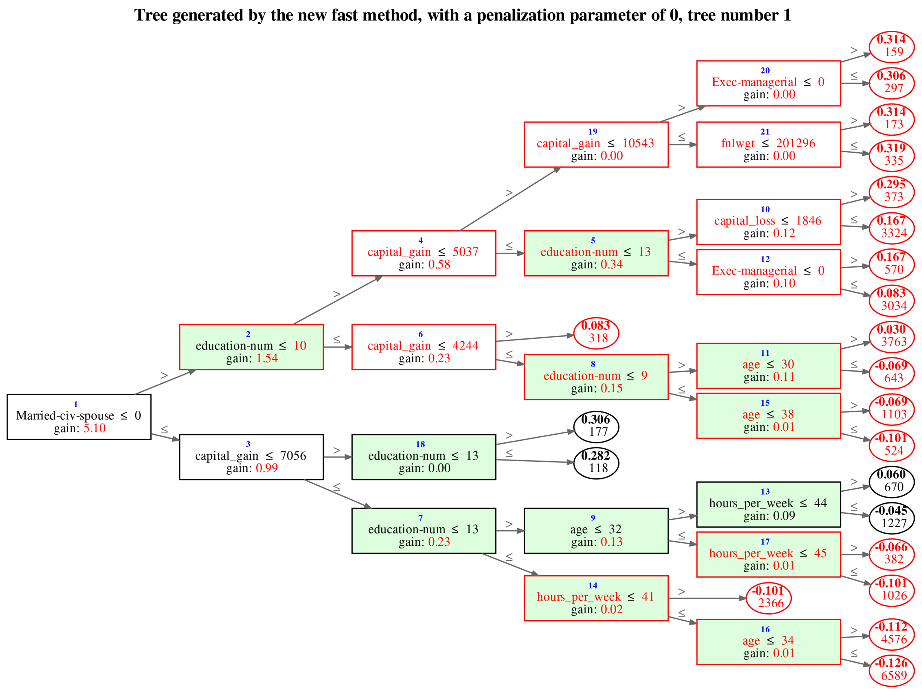
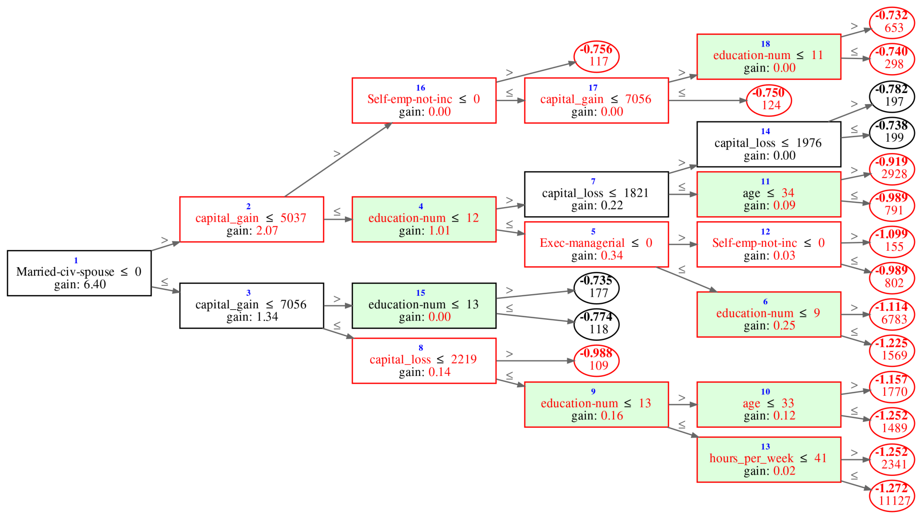

Tree generated by the new fast method, with a penalization parameter of 0, tree number 1
1 Married-civ-spouse ≤ 0 gain: 5.10
2 education-num ≤ 10 gain: 1.54
3 capital_gain ≤ 7056 gain: 0.99
4 capital_gain ≤ 5037 gain: 0.58
6 capital_gain ≤ 4244 gain: 0.23
18 education-num ≤ 13 gain: 0.00
7 education-num ≤ 13 gain: 0.23
19 capital_gain ≤ 10543 gain: 0.00
5 education-num ≤ 13 gain: 0.34
8 education-num ≤ 9 gain: 0.15
9 age ≤ 32 gain: 0.13
14 hours_per_week ≤ 41 gain: 0.02
20 Exec-managerial ≤ 0 gain: 0.00
21 fnlwgt ≤ 201296 gain: 0.00
10 capital_loss ≤ 1846 gain: 0.12
12 Exec-managerial ≤ 0 gain: 0.10
11 age ≤ 30 gain: 0.11
15 age ≤ 38 gain: 0.01
13 hours_per_week ≤ 44 gain: 0.09
17 hours_per_week ≤ 45 gain: 0.01
16 age ≤ 34 gain: 0.01
0.314 159
0.306 297
0.314 173
0.319 335
0.295 373
0.167 3324
0.167 570
0.083 3034
0.083 318
0.030 3763
-0.069 643
-0.069 1103
-0.101 524
0.306 177
0.282 118
0.060 670
-0.045 1227
-0.066 382
-0.101 1026
-0.101 2366
-0.112 4576
-0.126 6589
>
≤

Figure 9: First two trees generated by our new fast method, with no penalization

Tree generated by the new fast method, with a penalization parameter of 2, tree number 0


1 Married-civ-spouse ≤ 0 gain: 6.40
2 capital_gain ≤ 5037 gain: 2.07
3 capital_gain ≤ 7056 gain: 1.34
16 Self-emp-not-inc ≤ 0 gain: 0.00
4 education-num ≤ 12 gain: 1.01
15 education-num ≤ 13 gain: 0.00
8 capital_loss ≤ 2219 gain: 0.14
17 capital_gain ≤ 7056 gain: 0.00
7 capital_loss ≤ 1821 gain: 0.22
5 Exec-managerial ≤ 0 gain: 0.34
9 education-num ≤ 13 gain: 0.16
18 education-num ≤ 11 gain: 0.00
14 capital_loss ≤ 1976 gain: 0.00
11 age ≤ 34 gain: 0.09
12 Self-emp-not-inc ≤ 0 gain: 0.03
6 education-num ≤ 9 gain: 0.25
10 age ≤ 33 gain: 0.12
13 hours_per_week ≤ 41 gain: 0.02
-0.756 117
-0.750 124
-0.735 177
-0.774 118
-0.988 109
-0.732 653
-0.740 298
-0.782 197
-0.738 199
-0.919 2928
-0.989 791
-1.099 155
-0.989 802
-1.114 6783
-1.225 1569
-1.157 1770
-1.252 1489
-1.252 2341
-1.272 11127

Tree generated by the new fast method, with a penalization parameter of 2, tree number 1

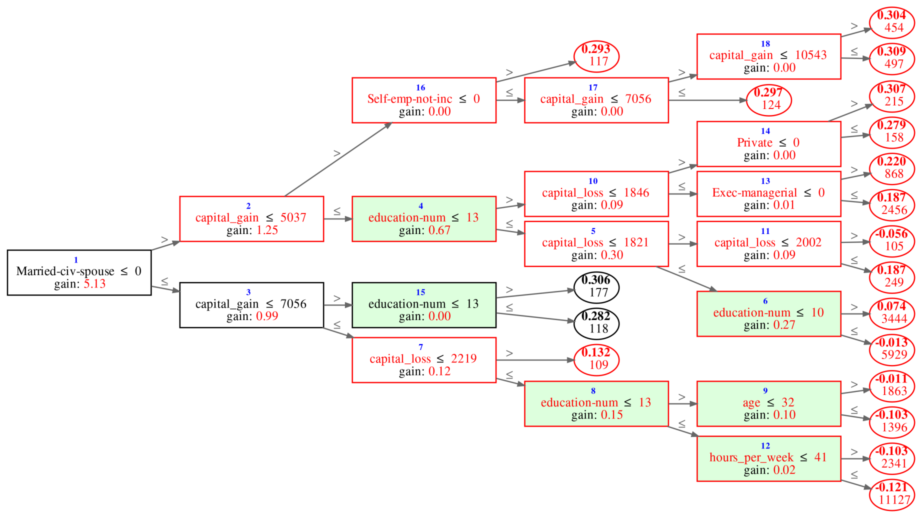
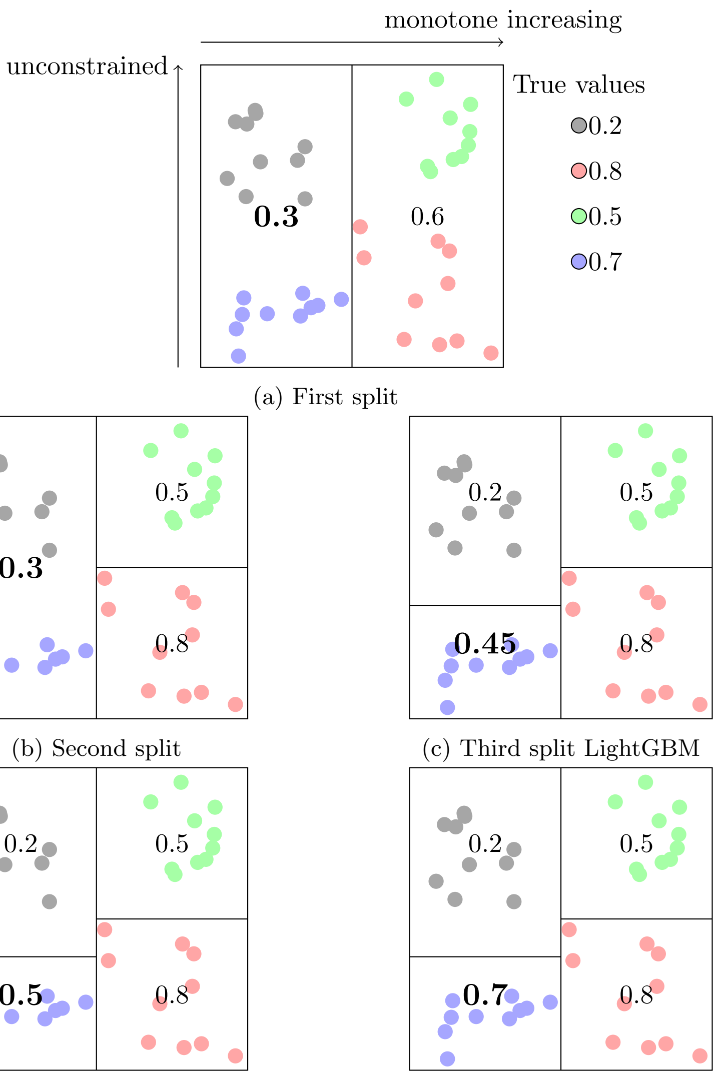

1 Married-civ-spouse ≤ 0 gain: 5.13
2 capital_gain ≤ 5037 gain: 1.25
3 capital_gain ≤ 7056 gain: 0.99
16 Self-emp-not-inc ≤ 0 gain: 0.00
4 education-num ≤ 13 gain: 0.67
15 education-num ≤ 13 gain: 0.00
7 capital_loss ≤ 2219 gain: 0.12
17 capital_gain ≤ 7056 gain: 0.00
10 capital_loss ≤ 1846 gain: 0.09
5 capital_loss ≤ 1821 gain: 0.30
8 education-num ≤ 13 gain: 0.15
18 capital_gain ≤ 10543 gain: 0.00
14 Private ≤ 0 gain: 0.00
13 Exec-managerial ≤ 0 gain: 0.01
11 capital_loss ≤ 2002 gain: 0.09
6 education-num ≤ 10 gain: 0.27
9 age ≤ 32 gain: 0.10
12 hours_per_week ≤ 41 gain: 0.02
0.293 117
0.297 124
0.306 177
0.282 118
0.132 109
0.304 454
0.309 497
0.307 215
0.279 158
0.220 868
0.187 2456
-0.056 105
0.187 249
0.074 3444
-0.013 5929
-0.011 1863
-0.103 1396
-0.103 2341
-0.121 11127

Figure 10: First two trees generated by our new fast method, with a penalization parameter of 2



(a) First split

(b) Second split

(c) Third split LightGBM

(d) Third split fast method

(e) Third split slow method

Figure 11: Theoretical example of the behavior of the different constraining methods

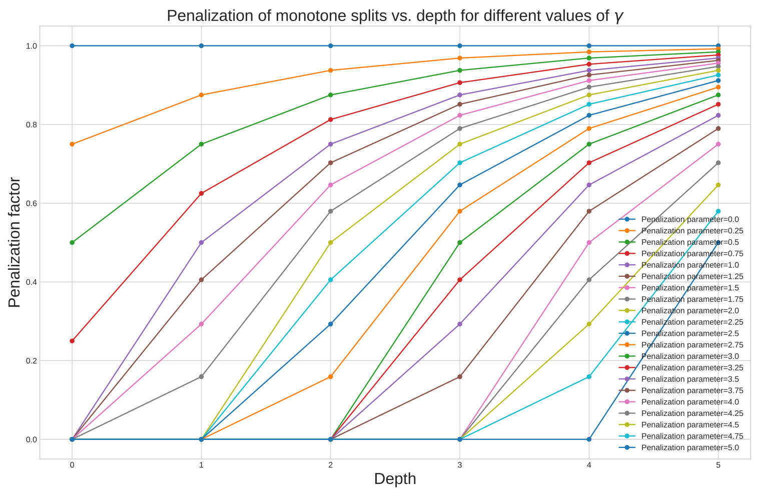


Figure 12: Penalty function for different penalization parameters