Data Mining in Smart Systems Semestral Project Report

Zoltán Zele

Eötvös Lóránd University, Budapest, Hungary

Abstract. The following report summarizes the the main ideas and principles of Gradient Boosting, its hyper-parameters, and gives a sample implementation for binary classification. The implemented classifier was ran on 61 datasets through a grid search to find the best hyper-parameter configuration for each dataset.

1 Theory of Gradient Boosting

1.1 The supervised learning problem

Given some data points $x_1, x_2, \ldots, x_n \in X$ and corresponding labels $y_1, y_2, \ldots, y_n \in Y$, our goal is to "learn" the mapping $F : X \to Y$ that will correctly predict the labels of the given data and unseen data points too. Finding the exact mapping is impossible in most cases, so we will approximate it with some parameterized function F^θ (model). To measure how good our model approximates the true mapping, given the data, we use a loss function $L(y, F^\theta(x))$. Then the problem is to find the best parameters θ^*:

$$\theta^* = \arg\min_\theta \sum_{i=1}^{n} L(y_i, F^\theta(x_i)).$$

The used loss function depends on the specific problem at hand. If the labels y_i are continuous, then we say that the problem is a regression problem, if the labels y_i are members of a discrete set, then we say that the problem is a classification problem.

1.2 Gradient Descent

The gradient of a differentiable function f of n variables is a vector containing the partial derivatives of the function:

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right).$$

A very useful property of the gradient is that it points towards the steepest ascent, i.e where the function value increases the most.

There are many optimization problems where the minimizer of the objective function cannot be calculated explicitly. In those cases, first order information of the objective can be used, like the gradient, to make a guess where the minimizer could be. A very widely used algorithm called gradient descent does exactly this, chooses the negative gradient as the searching direction and finds a step length to step towards that direction. Given the previous approximation x_k and step length γ, the new approximation x_{k+1} is:

$$x_{k+1} = x_k - \gamma \nabla f(x_k). \tag{1}$$

1.3 Boosting

Boosting approximates $F : X \to Y$ by an additive expansion of the form[2]:

$$F^{\theta}(x) = \sum_{m=1}^{M} \beta_m h(x, \theta_m).$$

where the functions $h(x, \theta_m)$ (weak learners or base learners) are usually chosen to be simple functions of x with parameters $\theta = \{\theta_1, \theta_2, \dots\}$. The expansion coefficients β_m and the parameters θ_m are jointly fit to the training data in a forward stagewise manner[3]:

1. Initialize $F_0(x) = 0$.
2. For $m = 1$ to M:
   (a) Compute

$$(\beta_m, \theta_m) = \arg\min_{\beta, \theta} \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + \beta h(x_i, \theta)). \tag{2}$$

   (b) Set $F_m(x) = F_{m-1}(x) + \beta_m h(x, \theta_m)$.

1.4 Gradient Boosting

Gradient boosting approximately solves (2) for arbitrary differentiable loss functions $L(y, F(x))$ with a two step procedure[2]. First, the function $h(x, \theta)$ is fit by least-squares

$$\theta_m = \arg\min_{\theta, \rho} \sum_{i=1}^{n} [r_{im} - \rho h(x_i, \theta)]^2$$

to the current "pseudo"-residuals

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}. \tag{3}$$

Then, given $h(x, \theta_m)$, the optimal value of the coefficient β_m is determined

$$\beta_m = \arg\min_{\beta} \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + \beta h(x_i, \theta_m)). \tag{4}$$

Basically, the base learners are fit to approximate the negative gradient, and then a line search is performed to obtain the optimal step length (one that decreases the objective function the most). This way the step 2(b) of the algorithm in section (1.3) is analogous to the step of gradient descent defined at (1) in function space.

Gradient Tree Boosting Gradient tree boosting is using regression trees as base learners[2]. At every iteration m, a regression tree partitions the X-space into disjoint regions $R_{jm}, j = 1, 2, \ldots, J$, represented by the terminal nodes of the tree. J is the number of terminal nodes and it is usually a hyper-parameter of the model. A constant value $\bar{y}_{jm}$ is assigned to each region. Thus a tree can be formally expressed as

$$T(x, \{R_{jm}\}_1^J) = \sum_{j=1}^{J} \bar{y}_{jm} I(x \in R_{jm}). \tag{5}$$

Here $\bar{y}_{jm} = mean_{x_i \in R_{jm}}(r_{im})$ is the mean of (3) in each region R_{jm}.

With regression trees, (4) can be solved separately within each region R_{jm} defined by the corresponding terminal node j of the m^{th} tree. Because the tree (5) predicts a constant value $\bar{y}_{jm}$ within each region R_{jm}, the solution to (4) reduces to a simple "location" estimate based on the loss L:

$$\gamma_{jm} = \arg\min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma)$$

and the update rule becomes

$$F_m(x) = F_{m-1}(x) + \sum_{j=1}^{J} \gamma_{jm} I(x \in R_{jm}).$$

Gradient Tree Boosting Algorithm The general gradient tree boosting algorithm for a differentiable loss function L is the following[3]:

1. Initialize $F_0(x) = \arg\min_\gamma \sum_{i=1}^{n} L(y_i, \gamma)$
2. For $m = 1$ to M:
   (a) For $i = 1, 2, \ldots, n$ compute
   $$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F=F_{m-1}}.$$
   (b) Fit a regression tree to the targets r_{im} giving terminal regions R_{jm}, $j = 1, 2, \ldots, J_m$.
   (c) For $j = 1, 2, \ldots, J_m$ compute
   $$\gamma_{jm} = \arg\min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma).$$
   (d) Update $F_m(x) = F_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$
3. Output $\hat{F}(x) = F_M(x)$

2 Hyper-parameters

2.1 Number of Boosting Iterations

A hyper-parameter that arises naturally from the algorithm is the number of boosting iterations M. For large enough M, one can make the training loss arbitrarily small. However, this can lead to overfitting, which in turn reduces the power of the model to generalize to unseen data. The optimal number M^* is application dependent and have to be fine-tuned for the data at hand. As it was found[3] this parameter should not be tuned seperately, it's optimal value is strongly connected to the learning rate. See section (2.3).

2.2 Number of Terminal Nodes

According to [3], boosting was considered as a technique for combining models (trees for gradient boosting), and as such, the tree building algorithm was regarded as a primitive that produced models to be combined by the boosting process. The optimal size of the trees were determined seperately for each tree, starting with a very large one, and then pruning it until the approximately optimal number of terminal nodes were found. This procedure resulted in way too large trees (especially in the early stages), causing substantial performance degradation and increasing the computational cost. Fitting a full tree to the training data at each iteration is unnecessary and does not generalize well. As Josh Starmer says in his YouTube video[4] about gradient boosting: It is enough for each tree to make a prediction that is just slightly better than a random guess, since a lot of trees are built during the process.

The easiest way to avoid this problem is to restrict the trees to have the same number of terminal nodes, $J_m = J, \forall m$. At each iteration a J-terminal node tree is built. Thus J becomes a hyper-parameter of the boosting process that can be tuned for the data at hand.

Although in many applications $J = 2$ is insufficient, it is unlikely that $J > 10$ will be required[3]. Experience dictates that $4 \leq J \leq 8$ works well, with results being fairly insensitive in this range.

2.3 Learning Rate

The learning rate or "shrinkage" is a parameter $0 < \nu < 1$ that scales the contribution of each tree during boosting. This means that in the gradient boosting algorithm, the update rule becomes:

$$F_m(x) = F_{m-1}(x) + \nu \sum_{j=1}^{J} \gamma_{jm} I(x \in R_{jm}). \tag{6}$$

Smaller values of ν lead to larger values of M to achieve the same training loss, so there is a tradeoff between them. Empirically it has been found [3, 1] that smaller values of ν favor better test error, and require correspondingly larger values of M. In fact, the best strategy appears to be to set ν to be very small ($\nu < 0.1$) and then choose M by early stopping.

2.4 Subsampling

Jerome H. Friedman introduced the idea of subsampling or stochastic gradient boosting in his paper[2]. At each iteration a subsample of the training data is drawn at random (without replacement) from the full training dataset. This randomly selected subsample is then used, instead of the full dataset, to fit the base learner.

Let $\{y_i, x_i\}_1^n$ be the entire training data sample and $\{\pi(i)\}_1^n$ be a random permutation of the integers $\{1, 2, \dots, n\}$. Then a random sample of size $\tilde{n} < n$ is given by $\{y_{\pi(i)}, x_{\pi(i)}\}_1^{\tilde{n}}$. The stochastic gradient tree boosting algorithm is the then[2]

1. Initialize $F_0(x) = \arg\min_\gamma \sum\limits_{i=1}^{n} L(y_i, \gamma)$
2. For $m = 1$ to M:
   (a) For $i = 1, 2, \dots, \tilde{n}$ compute
   $$r_{\pi(i)m} = -\left[\frac{\partial L(y_{\pi(i)}, F(x_{\pi(i)}))}{\partial F(x_{\pi(i)})}\right]_{F=F_{m-1}}.$$
   (b) Fit a regression tree to the targets $r_{\pi(i)m}$ giving terminal regions R_{jm}, $j = 1, 2, \dots, J$.
   (c) For $j = 1, 2, \dots, J$ compute
   $$\gamma_{jm} = \arg\min_\gamma \sum_{x_{\pi(i)} \in R_{jm}} L(y_{\pi(i)}, F_{m-1}(x_{\pi(i)}) + \gamma).$$
   (d) Update $F_m(x) = F_{m-1}(x) + \nu \sum\limits_{j=1}^{J} \gamma_{jm} I(x \in R_{jm})$
3. Output $\hat{F}(x) = F_M(x)$

A typical value for $\tilde{n}$ can be $\frac{1}{2}$, although for large n, $\tilde{n}$ can be substantially smaller than $\frac{1}{2}$[3].

Subsampling reduces the computation time by the same fraction $\tilde{n}$, and in many cases it actually produces a more accurate model. It has been found[3] that subsampling along with shrinkage slightly outperforms the ones without it, but subsampling without shrinkage does poorly.

3 Implementation

3.1 Gradient Boosting for Binary Classification

The following will be a derivation of a concrete implementation of gradient boosting for binary classification, mainly based on Josh Starmer's video[4].

In binary classification the set of possible labels Y has a cardinality of 2, $|Y| = 2$. Let's assume that $Y = \{1, 0\}$, and let $p(x) = \Pr(y = 1|x)$ be the probability of the sample being in the first class.

Loss Function The loss function that we will use is the binomial negative log-likelihood (also called cross-entropy)

$$L(y, p(x)) = -[y \log(p(x)) + (1-y)\log(1-p(x))]$$

where $p(x)$ is the model's predicted probability of x being in the first class. We can convert this predicted probability into a log-odds prediction like

$$F(x) = log \frac{p(x)}{1-p(x)} = log \frac{Pr(y=1|x)}{Pr(y=0|x)}$$

and we can convert the log-odds prediction to the predicted probability like

$$p(x) = \frac{e^{F(x)}}{1+e^{F(x)}}. \tag{7}$$

Then the loss function in terms of the log-odds, after some simplification is

$$L(y, F(x)) = -yF(x) + log(1+e^{F(x)}).$$

Initializing the model Recall from the gradient boosting algorithm, that to initialize the model we have to solve the following equation:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma) = \arg\min_{\gamma} \sum_{i=1}^{n} (-y_i\gamma + log(1+e^{\gamma})).$$

To find γ, the initial log-odds prediction of our model, we have to take the derivative with respect to the predicted log-odds, set it equal to 0, and then solve for γ. The derivative of the loss function with respect to the log-odds is

$$\frac{\partial L(y, F(x))}{\partial F(x)} = -y + \frac{e^{F(x)}}{1+e^{F(x)}} = -y + p(x). \tag{8}$$

By using the predicted probability instead of the log-odds, the derivative simplifies, so let's calculate the initial predicted probability p and then convert it to the log-odds prediction γ.

$$\sum_{i=1}^{n}(-y_i + p) = 0 \iff \sum_{i=1}^{n} -y_i = -np \iff \frac{\sum_{i=1}^{n} y_i}{n} = p$$

and then the log-odds prediction is $\gamma = log\frac{p}{1-p}$.

Calculating Pseudo-Residuals We already calculated the derivative of the loss function at (8). Then the pseudo-residuals are

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F=F_{m-1}} = -(-y_i + \frac{e^{F_{m-1}(x_i)}}{1+e^{F_{m-1}(x_i)}}) = y_i - p_{m-1}(x_i) \tag{9}$$

where $p_{m-1}(x_i)$ is the previously predicted probability based on the previously predicted log-odds $F_{m-1}(x_i)$, using (7).

Computing the Output Values for Terminal Regions After building a regression tree to predict the pseudo-residuals, thus creating terminal regions R_{jm}, $j = 1, \ldots, J$, we have to compute

$$\gamma_{jm} = \arg\min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma) = \tag{10}$$

$$\arg\min_{\gamma} \sum_{x_i \in R_{jm}} -y_i(F_{m-1}(x_i) + \gamma) + log(1 + e^{F_{m-1}(x_i)+\gamma}).$$

In theory, we could take derivative with respect to γ, set it equal to 0, and then solve for γ, but it would be very complicated. Instead, let's approximate the loss function with it's second order Taylor expansion

$$L(y_i, F_{m-1}(x_i) + \gamma) \approx L(y_i, F_{m-1}(x_i)) + \frac{\partial L(y_i, F_{m-1}(x_i))}{\partial F_{m-1}(x_i)}\gamma + \tag{11}$$

$$\frac{1}{2}\frac{\partial^2 L(y_i, F_{m-1}(x_i))}{\partial F_{m-1}(x_i)^2}\gamma^2.$$

Then the derivative with respect to γ is

$$\frac{\partial L(y_i, F_{m-1}(x_i) + \gamma)}{\partial \gamma} \approx \frac{\partial L(y_i, F_{m-1}(x_i))}{\partial F_{m-1}(x_i)} + \frac{\partial^2 L(y_i, F_{m-1}(x_i))}{\partial F_{m-1}(x_i)^2}\gamma.$$

Substituting the loss function with (11) in (10), and then solving for γ, we get

$$\gamma = \frac{\sum_{x_i \in R_{jm}} -\frac{\partial L(y_i, F_{m-1}(x_i))}{\partial F_{m-1}(x_i)}}{\sum_{x_i \in R_{jm}} \frac{\partial^2 L(y_i, F_{m-1}(x_i))}{\partial F_{m-1}(x_i)^2}}.$$

The first derivative of the loss function is calculated at (8). The second derivative, after some simplification, is

$$\frac{e^{F_{m-1}(x_i)}}{1 + e^{F_{m-1}(x_i)}} \frac{1}{1 + e^{F_{m-1}(x_i)}} = p_{m-1}(x_i)(1 - p_{m-1}(x_i)).$$

Finally, the optimal value for γ is

$$\gamma = \frac{\sum_{x_i \in R_{jm}} -(-y_i + p_{m-1}(x_i))}{\sum_{x_i \in R_{jm}} p_{m-1}(x_i)(1 - p_{m-1}(x_i))} = \frac{\sum_{x_i \in R_{jm}} r_{im}}{\sum_{x_i \in R_{jm}} p_{m-1}(x_i)(1 - p_{m-1}(x_i))}$$

where $p_{m-1}(x_i)$ is derived from $F_{m-1}(x_i)$ using (7).

The other parts of the algorithm do not require more explanation, a new log-odds prediction have to be computed using (6), and then increment m and loop.

The summation in (6) is only there in case a sample end up in more than one terminal region. When we want to make a prediction on which class x belongs to, we compute $\hat{F}(x)$ and then convert it to a probability p, using (7). If $p > 0.5$ then the model will predict that x belongs to the first class, otherwise to the second class.

3.2 Implemented classifier

The implemented gradient boosting classifier can be found in the attached files as `GradientBoostingClassifier.py`. It's a python class that can be configured in it's constructor. Each constructor parameter refers to a hyper-parameter discussed in section (2). The `learning_rate` refers to ν, the `subsampling` refers to $\tilde{n}$, the `n_estimators` refers to M, and `max_leaf_nodes` refers to J. The classifier implements the stochastic gradient tree boosting algorithm mentioned in section (2.4), with the classification specific steps derived in the previous section (3.1). The base learners used in the algorithm are regression trees of `DecisionTreeRegressor` class implemented by scikit-learn[5]. The classifier have methods for fitting to training data and for predicting sample labels, and also for predicting probabilities of belonging to a class. The other methods are used internally by the fitting and predicting methods.

The methods of the class are tested in the `UnitTests.py` python script, except the fitting method, which is highly nondeterministic. It is assumed that the tree building algorithm of the used trees are correct. During fitting the loss can be monitored by setting the parameter `verbose` to `True`. This will print the loss at each iteration, which can be used for debugging if something goes wrong.

The `main.py` script handles the data loading and preprocessing steps. It takes, as the first command line argument, the path of the data file with `.arff` format, the number of trees (M), as the second command line argument, the learning rate (ν), as the third command line argument, the number of leaf nodes (J), as the fourth command line argument, and the subsampling rate ($\tilde{n}$), as the fifth command line argument.

As data preprocessing, the categorical attributes of the datasets are transformed in the following way: For each value that a categorical attribute can have, new attributes are created, and the original categorical attribute is deleted. Each new attribute is binary, and have the value 1 if the particular instance of the data takes that value of the possible values for that category, and 0 otherwise. E.g.: if a category `V1` has possible values $1, 2, 3$, then the the following attributes will be created: `V1_is_1`, `V1_is_2` and `V1_is_3`, each with values 0, or 1. This step prevents the trees from handling the categorical attributes as ordinal, and not creating splitting rules as `data.fruit` $<$ `'orange'`, as CART trees are not able to handle categorical attributes and will encode them as floats.

After preprocessing, the dataset is split into train and test set, with 80% of the data being in the training set and 20% being in the test set. The split is done in a stratified way, so that the distribution of the sample labels are the same in the training and test sets. This last step proved to be necessary, as the grid search algorithm presented in the next section is doing a lot of splits for each dataset to test for different parameter configurations, and if the ratio of the labels in the dataset is e.g. $4:1$, then there is a large chance that the train set will have most of the samples with labels that are less frequent and then the test set will consist of the the samples with the more frequent labels by far. This results in unrealistically high test accuracy, because classifying samples that belong to the less frequent class is more difficult, and with such test set it is not tested if the classifier learned how to properly classified these samples. Also the accuracy of the classifier will not depend on the parameters, instead on how the data is split. I stumbled upon this problem when I was monitoring the grid search, and most of the datasets had 100% accuracy. It was hard to believe, so i tested one of the found parameters and it showed around 80% accuracy for the same dataset, so clearly the split was not done right when it was tested in the first place.

4 Results

4.1 Grid Search

To find the best hyper-parameter settings for each dataset, the classifier was run through a grid search, trying out every possible hyper-parameter combination from the sets of possible parameter values.

The possible values for the learning rate was $\{0.1, 0.05, 0.01, 0.001\}$, for the number of leaf nodes $\{2, 3, 6, 11\}$, for the number of trees $\{100, 200, 500, 1000\}$, and for the subsampling rate $\{0.6, 0.7, 0.8, 1.0\}$. I only considered 4 possible value for each parameter because more possibilities would increase the execution time exponentially. Even like this, when there are 4 possible values for 4 different parameters, the number of combinations is $4^4 = 256$. Assuming that the classifier will run for only 1 second for each possible combination through 61 datasets, the execution time is 15616 seconds $\approx$ 260 minutes $\approx$ 4.3 hours. But in practice the classifier run for a longer time (much longer for big datasets when building 1000 trees).

The `gridsearch.sh` shell script implements the grid search algorithm. It reads the datasets from the `data` directory and runs the `main.py` script on each dataset with each combination of parameters. If the tried current configuration produces better accuracy than the so far best, then it saves that configuration. At the end it saves the found parameters in the `results.txt` file.

After running this script for more than 12 hours, it was only ready with about half of the datasets, so i created a second script `gridsearchParallel.sh` that creates 4 processes, where each process can run the grid search on different datasets in parallel. This time the grid search was ready in 16 hours (I ran it on a low/medium end laptop with a 4 core processor, if someone has access to more than this, then it is easy to configure the script to run it on more processes). The results are written in the `results` directory, where each file has the name of the dataset, and contains the best found parameters with the corresponding accuracy.

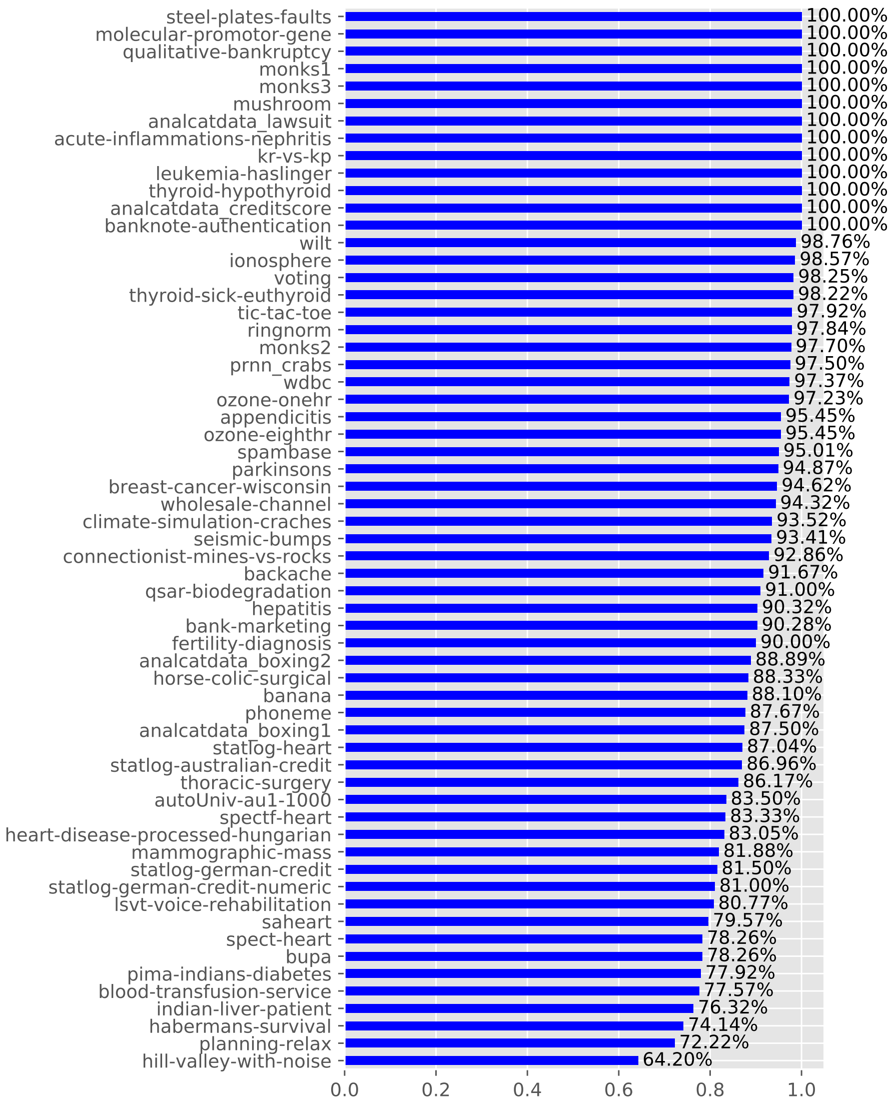

Fig. 1. The accuracy of the model with the best found hyper-parameter settings for each dataset.

4.2 Analysis

Fig. 1. shows the accuracy of the model with the best parameters for each dataset. Fig. 2. shows the distribution of the hyper-parameters considering all of the datasets. We can see that in most cases a learning rate of 0.1 was sufficiently small, and 100 trees were enough. The distribution of leaf nodes was roughly uniform, while lower subsampling rates were giving better results in most cases.

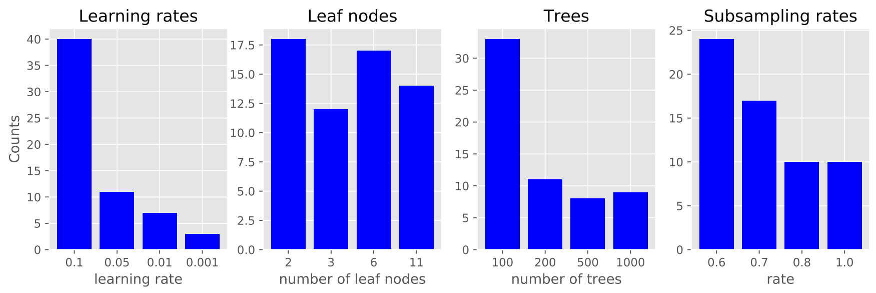

Fig. 2. The distribution of the hyper-parameters across all datasets.

Now we explore how the distribution changes if we look at datasets that have less than 500 samples and datasets with more than 500 samples. Fig. 3 shows the distribution for datasets with less than 500 samples and Fig. 4. shows the distribution for datasets with more than 500 samples. The biggest difference is that the classifier for datasets that have less than 500 samples built trees with 2 leaf nodes the most, while for datasets with more than 500 samples, built trees with 6 leaf nodes the most. Also for bigger datasets, 500 $\leq$ trees often needed.

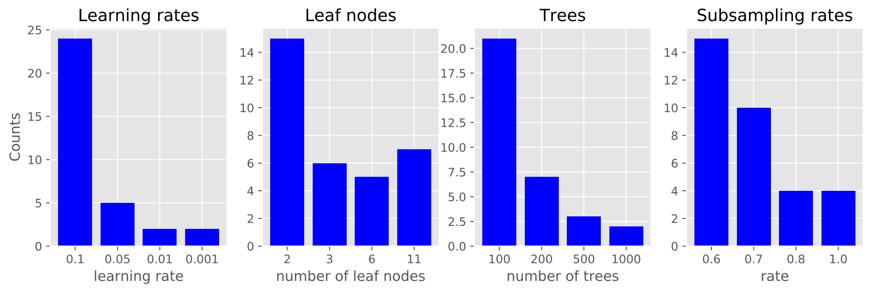

Fig. 3. The distribution of the hyper-parameters with less than 500 sample datasets.

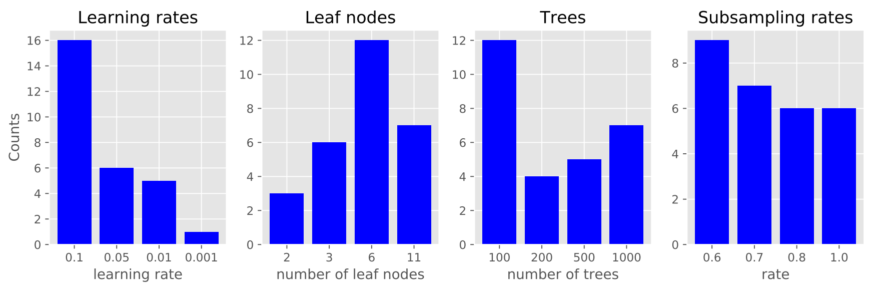

Fig. 4. The distribution of the hyper-parameters with more than 500 sample datasets.

In section (2.3) we discussed that there is a tradeoff between the learning rate and the number of trees. Fig. 5. shows the relation between the learning rate and the number of trees. There are far more datasets where the best learning rate was 0.1, so i would not make any assumptions about correlation from this data. There were only 3 datasets where the best learning rate was 0.001.

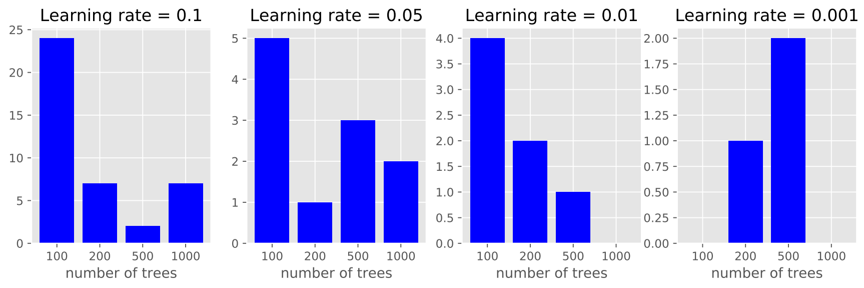

Fig. 5. The distribution of the number of trees for each learning rate.

4.3 Remarks

The way the grid search was implemented is that it started with smaller values and then tried the bigger ones (one exception is the learning rate, where the value gets lower in time, but there is an ordering still). This may have introduced some bias in the data towards lower values since if the accuracy is the same for two parameter settings, then the grid search would keep the one that it found first (for smaller datasets this happens multiple times). Also for the datasets where the classifier reached 100% accuracy, the grid search is stopped since we can't find a better result, and most of the times it found these parameters just after a few iterations, so it kept the smaller valued parameter configurations. On the other hand if our goal is to find the best parameter settings, then lower values mean simpler model, which is desirable for e.g. performance considerations. The figures were created using the `stats.py` script.

References

1. J. Friedman, Greedy Function Approximation: A Gradient Boosting Machine, The Annals of Statistics, Vol. 29, No. 5, 2001.
2. J. Friedman, Stochastic Gradient Boosting, 1999
3. T. Hastie, R. Tibshirani and J. Friedman. Elements of Statistical Learning Ed. 2, Springer, 2009.
4. "Gradient Boost Part 4: Classification Details", YouTube, uploaded by StatQuest with Josh Starmer, 22 Apr 2019, `https://www.youtube.com/watch?v=StWY5QWMXCw`.
5. scikit-learn Homepage, `https://scikit-learn.org/`. Last accessed 27 May 2019