Anastasia Manokhina Follow
Data Scientist
Apr 16 · 18 min read

Open Machine Learning Course. Topic 10. Gradient Boosting

Hello everyone!

So far, we've covered 9 topics from Exploratory Data Analysis to Time Series Analysis in Python. Today we are going to have a look at one of the most popular and practical machine learning algorithms: gradient boosting. You can find more detailed math in the nbviewer format of the article.

Article Outline

1. Introduction and history of boosting

2. Gradient Boosting Machine algorithm
3. Loss functions
4. Assignment #10
5. Useful resources

1. Introduction and history of boosting

Almost everyone in machine learning has heard about gradient boosting. Many data scientists include this algorithm in their data scientist's toolbox because of the good results it yields on any given (unknown) problem.

Furthermore, XGBoost is often the standard recipe for winning ML competitions. It is so popular that the idea of stacking XGBoosts has become a meme. Moreover, boosting is an important component in many recommender systems; sometimes, it is even considered a brand. Let's look at the history and development of boosting.

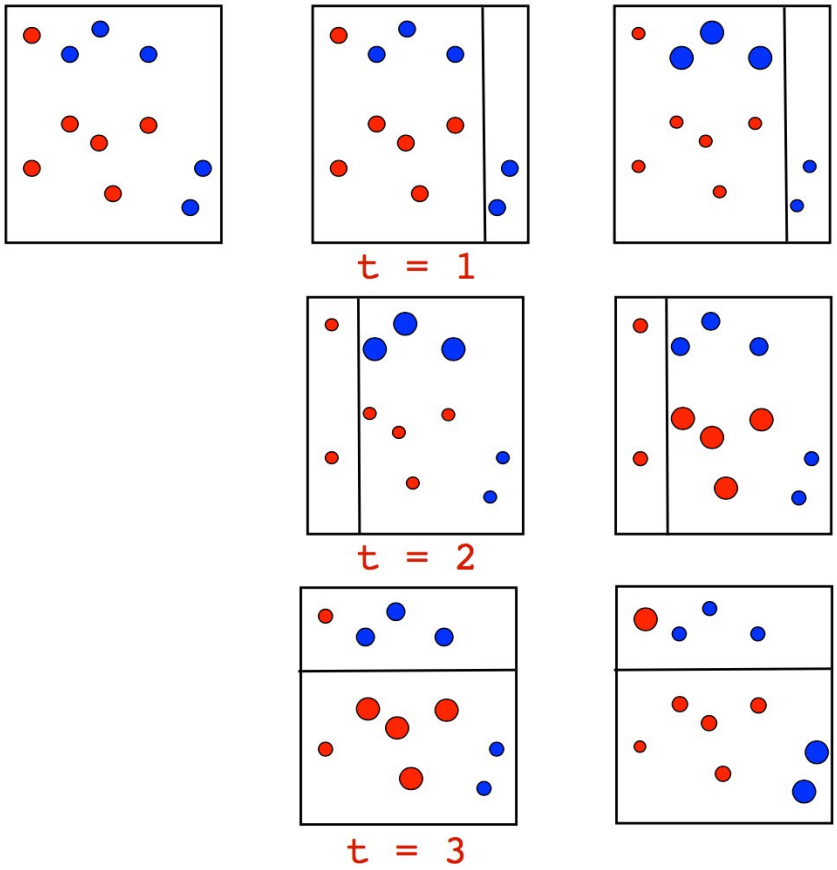
Boosting was born out of the question: is it possible to get one strong model from a large amount of relatively weak and simple models? By saying "weak models", we do not mean simple basic models like decision trees but models with poor accuracy performance, where poor is a little bit better than random.

A positive mathematical answer to this question was identified, but it took a few years to develop fully functioning algorithms based on this solution e.g. AdaBoost. These algoritms take a greedy approach: first, they build a linear combination of simple models (basic algorithms) by re-weighing the input data. Then, the model (usually a decision tree) is built on earlier incorrectly predicted objects, which are now given larger weights.

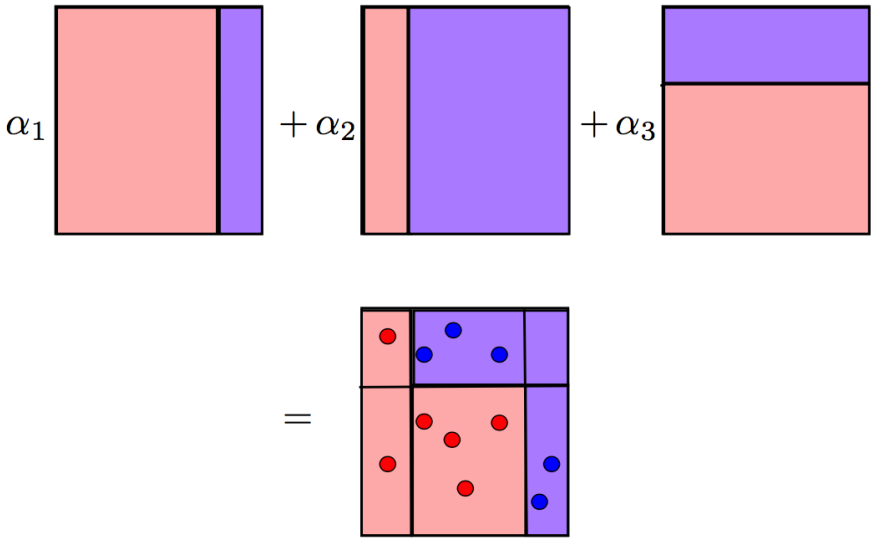
Many machine learning courses study AdaBoost—the ancestor of GBM (Gradient Boosting Machine). However, since AdaBoost merged with GBM, it has become apparent that AdaBoost is just a particular variation of GBM.

The algorithm itself has a very clear visual interpretation and intuition for defining weights. Let's have a look at the following toy classification problem where we are going to split the data between the trees of

depth 1 (also known as ‘stumps’) on each iteration of AdaBoost. For the first two iterations, we have the following picture:



The size of point corresponds to its weight, which was assigned for an incorrect prediction. On each iteration, we can see that these weights are growing—the stumps cannot cope with this problem. Although, if we take a weighted vote for the stumps, we will get the correct classifications:



Here is a more detailed example of AdaBoost where, as we iterate, we can see the weights increase, especially on the border between classes.

AdaBoost works well, but the lack of explanation for why the algorithm is successful sewed the seeds of doubt. Some considered it a super-algorithm, a silver bullet, but others were skeptical and believed AdaBoost was just overfitting.

The overfitting problem did indeed exist, especially when data had strong outliers. Therefore, in those types of problems, AdaBoost was unstable. Fortunately, a few professors in the statistics department at Stanford, who had created Lasso, Elastic Net, and Random Forest, started researching the algorithm. In 1999, Jerome Friedman came up with the generalization of boosting algorithms development—Gradient

Boosting (Machine), also known as GBM. With this work, Friedman set up the statistical foundation for many algorithms providing the general approach of boosting for optimization in the functional space.

CART, bootstrap, and many other algorithms have originated from Stanford's statistics department. In doing so, the department has solidified their names in future textbooks. These algorithms are very practical, and some recent works have yet to be widely adopted. For example, check out glinternet.

Not many video recordings of Friedman are available. Although, there is a very interesting interview with him about the creation of CART and how they solved statistics problems (which is similar to data analysis and data science today) more than 40 years ago. There is also a great lecture from Hastie, a retrospective on data analysis from one of the creators of methods that we use everyday.

In general, there has been a transition from engineering and algorithmic research to a full-fledged approach to building and studying algorithms. From a mathematical perspective, this is not a big change—we are still adding (or boosting) weak algorithms and enlarging our ensemble with gradual improvements for parts of the data where the model was inaccurate. But, this time, the next simple model is not just built on re-weighted objects but improves its approximation of the gradient of overall objective function. This concept greatly opens up our algorithms for imagination and extensions.

History of GBM

It took more than 10 years after the introduction of GBM for it to become an essential part of the data science toolbox. GBM was extended to apply to different statistics problems: GLMboost and GAMboost for strengthening already existing GAM models, CoxBoost for survival curves, and RankBoost and LambdaMART for ranking.

Many realizations of GBM also appeared under different names and on different platforms: Stochastic GBM, GBDT (Gradient Boosted Decision Trees), GBRT (Gradient Boosted Regression Trees), MART (Multiple Additive Regression Trees), and more. In addition, the ML community

was very segmented and dissociated, which made it hard to track just how widespread boosting had become.

At the same time, boosting had been actively used in search ranking. This problem was rewritten in terms of a loss function that penalizes errors in the output order, so it became convenient to simply insert it into GBM. AltaVista was one of the first companies who introduced boosting to ranking. Soon, the ideas spread to Yahoo, Yandex, Bing, etc. Once this happened, boosting became one of the main algorithms that was used not only in research but also in core technologies in industry.

ML competitions, especially Kaggle, played a major role in boosting's popularization. Now, researchers had a common platform where they could compete in different data science problems with large number of participants from around the world. With Kaggle, one could test new algorithms on the real data, giving algoritms oppurtunity to "shine", and provide full information in sharing model performance results across competition data sets. This is exactly what happened to boosting

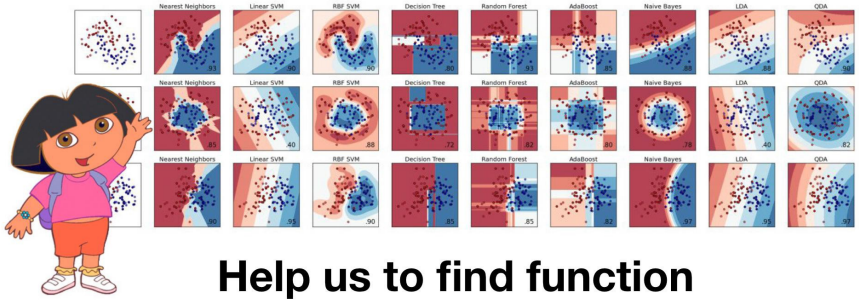
when it was used at Kaggle (check interviews with Kaggle winners starting from 2011 who mostly used boosting). The XGBoost library quickly gained popularity after its appearance. XGBoost is not a new, unique algorithm; it is just an extremely effective realization of classic GBM with additional heuristics.

This algorithm has gone through very typical path for ML algorithms today: mathematical problem and algorithmic crafts to successful practical applications and mass adoption years after its first appearance.

2. Gradient Boosting Machine algorithm

We are going to solve the problem of function approximation in a general supervised learning setting. We have a set of features X and target variables y which we use to restore the dependence $y = f(x)$. We restore the dependence by approximating $f(x)$ and by understanding which approximation is better when we use the loss function $L(y,f)$, which we want to minimize:

$$y \approx \hat{f}(x), \hat{f}(x) = \arg\min_{f(x)} L(y, f(x))$$



At this moment, we do not make any assumptions regarding the type of dependence $f(x)$, the model of our approximation, or the distribution of the target variable. We only expect that the function $L(y, f)$ is differentiable. Our formula is very general; let's define it for a particular data set with a population mean. Our expression for minimizing the loss of the data is the following:

$$\hat{f}(x) = \arg\min_{f(x)} \mathbb{E}_{x,y}[L(y, f(x))]$$

Unfortunately, the number of these functions is not just large, its functional space is infinite-dimensional. That is why it is acceptable for us to limit the search space by some family of functions. This simplifies the objective a lot because now we have a solvable optimization of parameter values.

Simple analytical solutions for finding the optimal parameters often do not exist, so the parameters are usually approximated iteratively. To start, we write down the empirical loss function that will allow us to evaluate our parameters using our data. Additionally, let's write out our approximation for a number of *M* iterations as a sum:

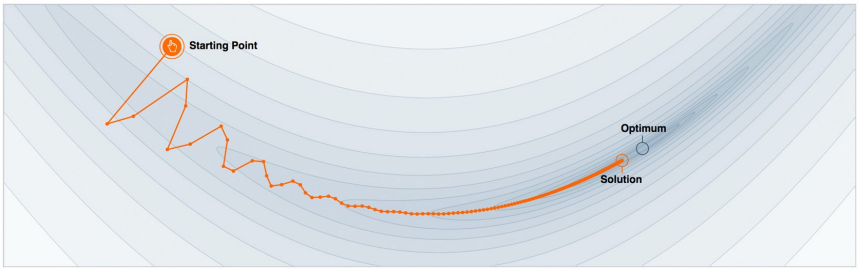
$$\hat{\theta} = \sum_{i=1}^{M} \hat{\theta}_i,$$
$$L_\theta(\hat{\theta}) = \sum_{i=1}^{N} L(y_i, f(x_i, \hat{\theta}))$$

Then, the only thing left is to find a suitable, iterative algorithm to minimize the last expression. Gradient descent is the simplest and most frequently used option. We define the gradient and add our iterative evaluations to it (since we are minimizing the loss, we add the minus sign). Our last step is to initialize our first approximation and choose the number of iterations *M*. Let's review the steps for this inefficient and naive algorithm:

1. Define the initial approximation of the parameters $\hat{\theta} = \hat{\theta}_0$
2. For every iteration $t = 1, \ldots, M$ repeat steps 3-7:
3. Calculate the gradient of the loss function $\nabla L_\theta(\hat{\theta})$ for the current approximation $\hat{\theta}$

$$\nabla L_\theta(\hat{\theta}) = \left[\frac{\partial L(y, f(x,\theta))}{\partial \theta}\right]_{\theta=\hat{\theta}}$$

4. Set the current iterative approximation $\hat{\theta}_t$ based on the calculated gradient $\hat{\theta}_t \leftarrow -\nabla L_\theta(\hat{\theta})$
5. Update the approximation of the parameters $\hat{\theta}$: $\hat{\theta} \leftarrow \hat{\theta} + \hat{\theta}_t = \sum_{i=0}^{t} \hat{\theta}_i$
6. Save the result of approximation $\hat{\theta}$: $\hat{\theta} = \sum_{i=0}^{M} \hat{\theta}_i$
7. Use the function that was found $\hat{f}(x) = f(x, \hat{\theta})$



Functional gradient descent

Let's imagine for a second that we can perform optimization in the function space and iteratively search for the approximations as functions themselves. We will express our approximation as a sum of incremental improvements, each being a function.

$$\hat{f}(x) = \sum_{i=0}^{M} \hat{f}_i(x)$$

Nothing has happened yet; we have only decided that we will search for our approximation not as a big model with plenty of parameters (as an example, neural network), but as a sum of functions, pretending we move in functional space.

In order to accomplish this task, we need to limit our search by some function family. There are a few issues here—first of all, the sum of models can be more complicated than any model from this family;

secondly, the general objective is still in functional space. Let's note that, on every step, we will need to select an optimal coefficient. For step t, the problem is the following:

$$\hat{f}(x) = \sum_{i=0}^{t-1} \hat{f}_i(x),$$

$$(\rho_t, \theta_t) = \arg\min_{\rho,\theta} \mathbb{E}_{x,y}[L(y, \hat{f}(x) + \rho \cdot h(x, \theta))],$$

$$\hat{f}_t(x) = \rho_t \cdot h(x, \theta_t)$$

Here is where the magic happens. We have defined all of our objectives in general terms, as if we could have trained any kind of model for any type of loss functions. In practice, this is extremely difficult, but, fortunately, there is a simple way to solve this task.

Knowing the expression of loss function's gradient, we can calculate its value on our data. So, let's train the models such that our predictions will be more correlated with this gradient (with a minus sign). In other words, we will use least squares to correct the predictions with these residuals. For classification, regression, and ranking tasks, we will minimize the squared difference between pseudo-residuals r and our predictions. For step t, the final problem looks the following:

$$\hat{f}(x) = \sum_{i=0}^{t-1} \hat{f}_i(x),$$

$$r_{it} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x)=\hat{f}(x)}, \quad \text{for } i = 1, \ldots, n,$$

$$\theta_t = \arg\min_{\theta} \sum_{i=1}^{n} (r_{it} - h(x_i, \theta))^2,$$

$$\rho_t = \arg\min_{\rho} \sum_{i=1}^{n} L(y_i, \hat{f}(x_i) + \rho \cdot h(x_i, \theta_t))$$

Friedman's classic GBM algorithm

We can now define the classic GBM algorithm suggested by Jerome Friedman in 1999. It is a supervised algorithm that has the following components:

- a dataset;
- number of iterations *M*;
- choice of loss function with a defined gradient;
- choice of function family of base algorithms with the training procedure;
- additional hyperparameters (for example, in decision trees, the tree depth);

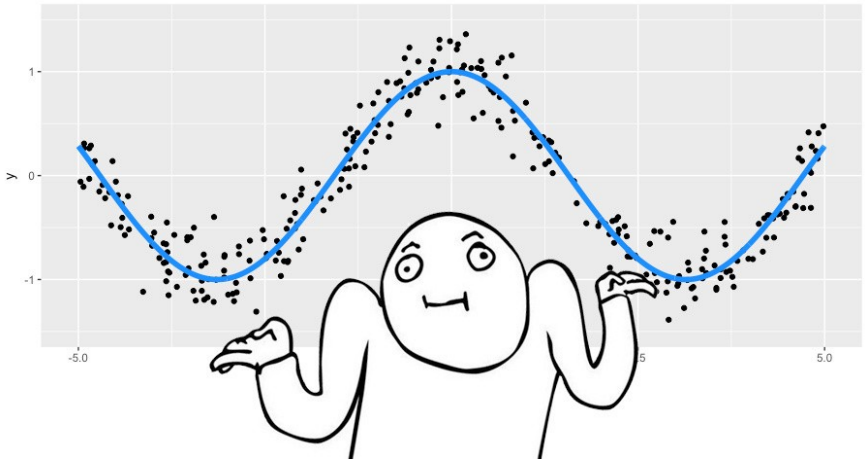
The only thing left is the initial approximation. For simplicity, for an initial approximation, a constant value *gamma* is used. The constant value, as well as the optimal coefficient, are identified via binary search or another line search algorithm over the initial loss function (not a gradient). So, we have our GBM algorithm described as follows:

1. Initialize GBM with constant value $\hat{f}(x) = \hat{f}_0, \hat{f}_0 = \gamma, \gamma \in \mathbb{R}$
$$\hat{f}_0 = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma)$$
2. For each iteration $t = 1, \dots, M$, repeat:
3. Calculate pseudo-residuals r_t
$$r_{it} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x)=\hat{f}(x)}, \quad \text{for } i = 1, \dots, n$$
4. Build new base algorithm $h_t(x)$ as regression on pseudo-residuals $\{(x_i, r_{it})\}_{i=1,\dots,n}$
5. Find optimal coefficient ρ_t at $h_t(x)$ regarding initial loss function
$$\rho_t = \arg\min_{\rho} \sum_{i=1}^{n} L(y_i, \hat{f}(x_i) + \rho \cdot h(x_i, \theta))$$
6. Save $\hat{f}_t(x) = \rho_t \cdot h_t(x)$
7. Update current approximation $\hat{f}(x)$: $\hat{f}(x) \leftarrow \hat{f}(x) + \hat{f}_t(x) = \sum_{i=0}^{t} \hat{f}_i(x)$
8. Compose final GBM model $\hat{f}(x)$: $\hat{f}(x) = \sum_{i=0}^{M} \hat{f}_i(x)$
9. Conquer Kaggle and the rest of the world

Step-By-Step example: How GBM Works

Let's see an example of how GBM works. In this toy example, we will restore a noisy function

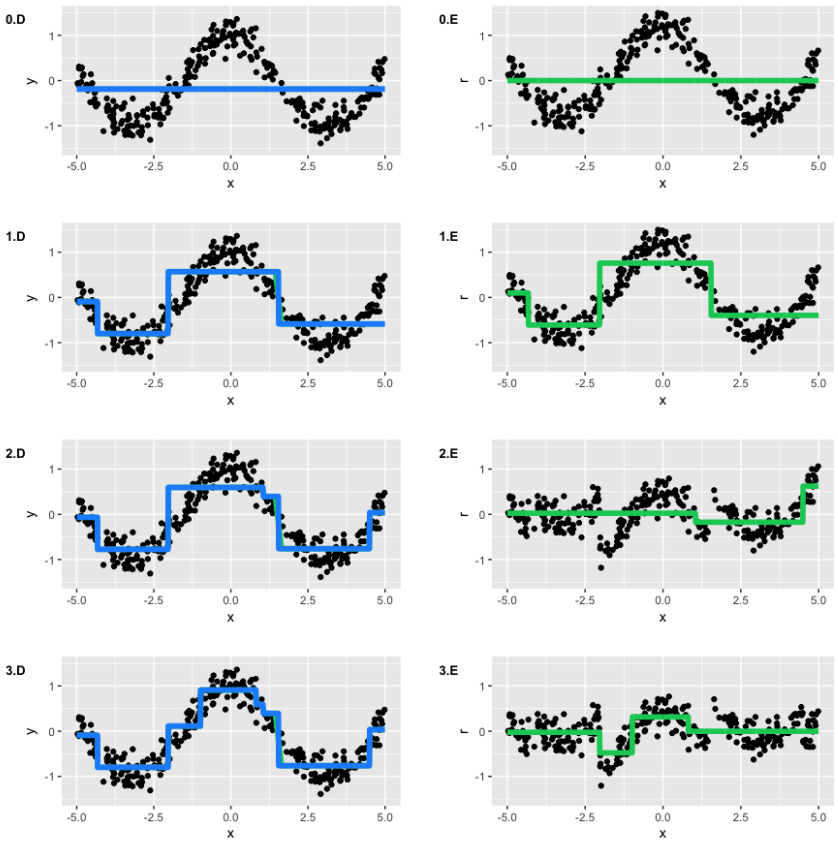
$$y = cos(x) + \epsilon, \epsilon \sim \mathcal{N}(0, \frac{1}{5}), x \in [-5, 5]$$



This is a regression problem with a real-valued target, so we will choose to use the mean squared error loss function. We will generate 300 pairs of observations and approximate them with decision trees of depth 2. Let's put together everything we need to use GBM:

- Toy data
- Number of iterations $M = 3$
- The mean squared error loss function $L(y, f) = (y - f)^2$
- Gradient of $L(y, f)$ is just residuals $r = (y-f)$
- Decision trees as base algorithms
- Hyperparameters of the decision trees: trees depth is equal to 2

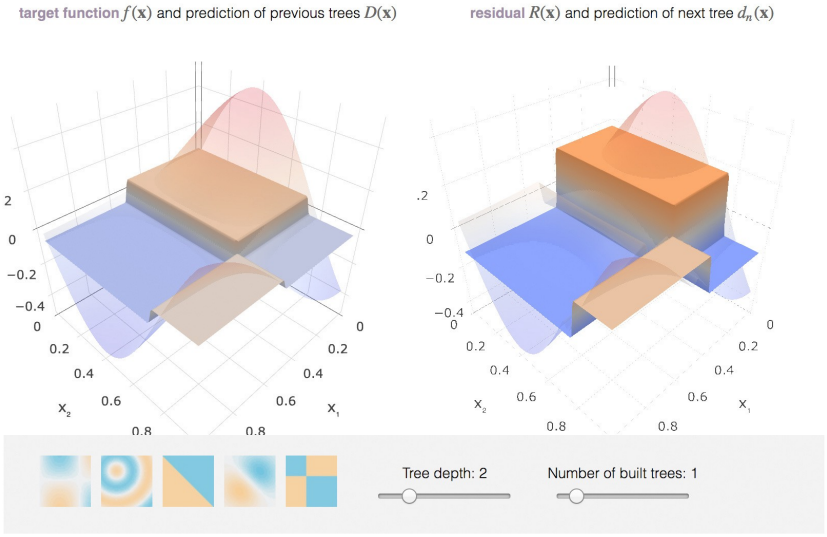
We will run GBM and draw two types of graphs: the current approximation (blue graph) and every tree built on its pseudo-residuals (green graph). The graph's number corresponds to the iteration number:



By the second iteration, our trees have recovered the basic form of the function. However, at the first iteration, we see that the algorithm has built only the "left branch" of the function. This was due to the fact that our trees simply did not have enough depth to build a symmetrical

branch at once, and it focused on the left branch with the larger error. Therefore, the right branch appeared only after the second iteration.

The rest of the process goes as expected—on every step, our pseudo-residuals decreased, and GBM approximated the original function better and better with each iteration. However, by construction, trees cannot approximate a continuous function, which means that GBM is not ideal in this example. To play with GBM function approximations, you can use the awesome interactive demo in this blog called Brilliantly wrong:



3. Loss functions

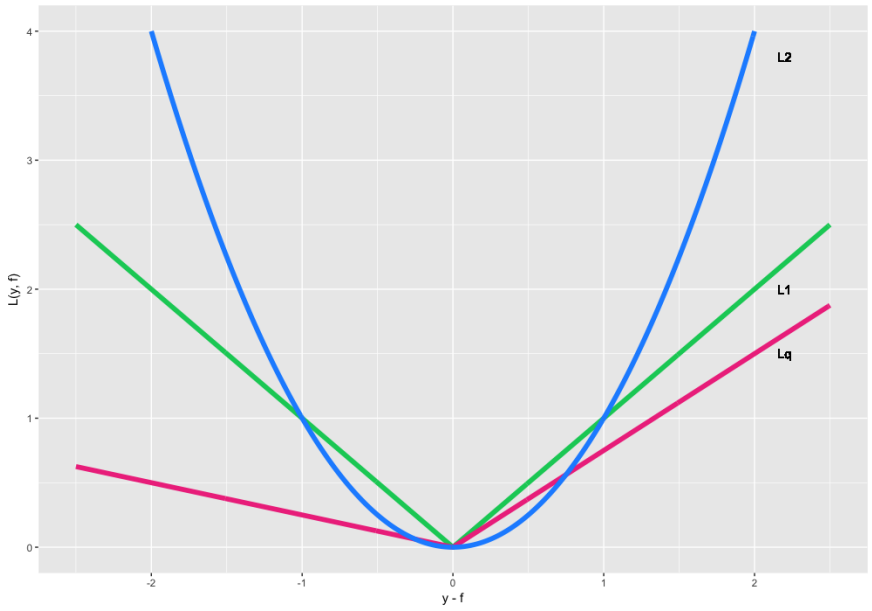
If we want to solve a classification problem instead of regression, what would change? We only need to choose a suitable loss function *L(y, f)*. This is the most important, high-level moment that determines exactly how we will optimize and what characteristics we can expect in the final model.

As a rule, we do not need to invent this ourselves—researchers have already done it for us. Today, we will explore loss functions for the two most common objectives: regression and binary classification.

Regression loss functions

Let's start with a regression problem for *y,* a real number. In order to choose the appropriate loss function, we need to consider which of the properties of the conditional distribution *(y|x)* we want to restore. The most common options are:

- $L(y, f) = (y—f)^2$ a.k.a. *L2* loss or Gaussian loss. It is the classical conditional mean, which is the simplest and most common case. If we do not have any additional information or requirements for a model to be robust, we can use the Gaussian loss.
- $L(y, f) = |y—f|$ a.k.a. *L1* loss or Laplacian loss. At the first glance, this function does not seem to be differentiable, but it actually defines the conditional median. Median, as we know, is robust to outliers, which is why this loss function is better in some cases. The penalty for big variations is not as heavy as it is in *L2*.
- *Lq* loss or Quantile loss. Instead of median, it uses quantiles. We can see that this function is asymmetric and penalizes the observations which are on the right side of the defined quantile.



Let's use loss function *L_q* on our data. The goal is to restore the conditional 75%-quantile of cosine. Let us put everything together for GBM:

- Toy data $\{(x_i, y_i)\}_{i=1,\dots,300}$ ✓
- A number of iterations $M = 3$ ✓;
- Loss function for quantiles

$$L_{0.75}(y, f) = \begin{cases} 0.25 \cdot |y - f|, & \text{if } |y - f| \le 0 \\ 0.75 \cdot |y - f|, & \text{if } |y - f| > 0 \end{cases}$$ ✓;

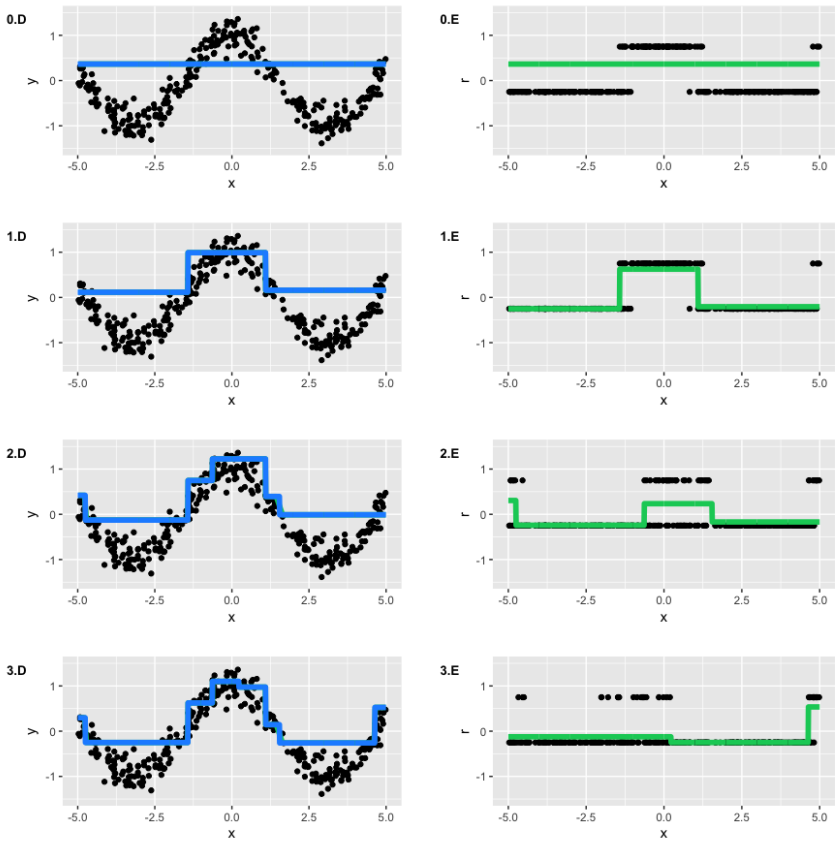
- Gradient $L_{0.75}(y, f)$ - function weighted by $\alpha = 0.75$. We are going to train tree-based model for classification: $r_i = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x)=\hat{f}(x)} =$

$$= \alpha I(y_i > \hat{f}(x_i)) - (1 - \alpha) I(y_i \le \hat{f}(x_i)),$$ ✓;

$$\text{for } i = 1, \dots, 300$$

- Decision tree as a basic algorithm $h(x)$ ✓;
- Hyperparameter of trees: depth = 2 ✓;

For our initial approximation, we will take the needed quantile of y. However, we do not know anything about optimal coefficients, so we'll use standard line search. The results are the following:

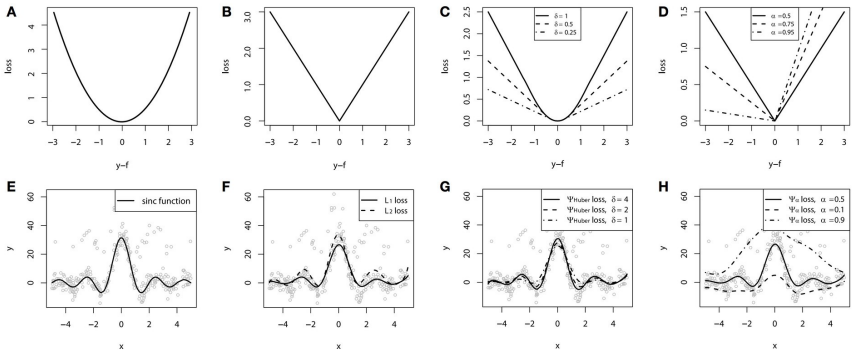


The overall results of GBM with quantile loss function are the same as the results with quadratic loss function offset by ~ 0.135. But if we were to use the 90%-quantile, we would not have enough data due to

the fact that classes would become unbalanced. We need to remember this when we deal with non-standard problems.

For regression tasks, many loss functions have been developed, some of them with extra properties. For example, they can be robust like in the Huber loss function. For a small number of outliers, the loss function works as *L2*, but after a defined threshold, the function changes to *L1*. This allows for decreasing the effect of outliers and focusing on the overall picture.

We can illustrate this with the following example. Data is generated from the function $y = sin(x) / x$ with added noise, a mixture from normal and Bernoulli distributions. We show the functions on graphs A-D and the relevant GBM on F-H (graph E represents the initial function):



Original size.

In this example, we used splines as the base algorithm. See, it does not always have to be trees for boosting?

We can clearly see the difference between the functions *L2*, *L1*, and Huber loss. If we choose optimal parameters for the Huber loss, we can get the best possible approximation among all our options. The difference can be seen as well in the 10%, 50%, and 90%-quantiles.

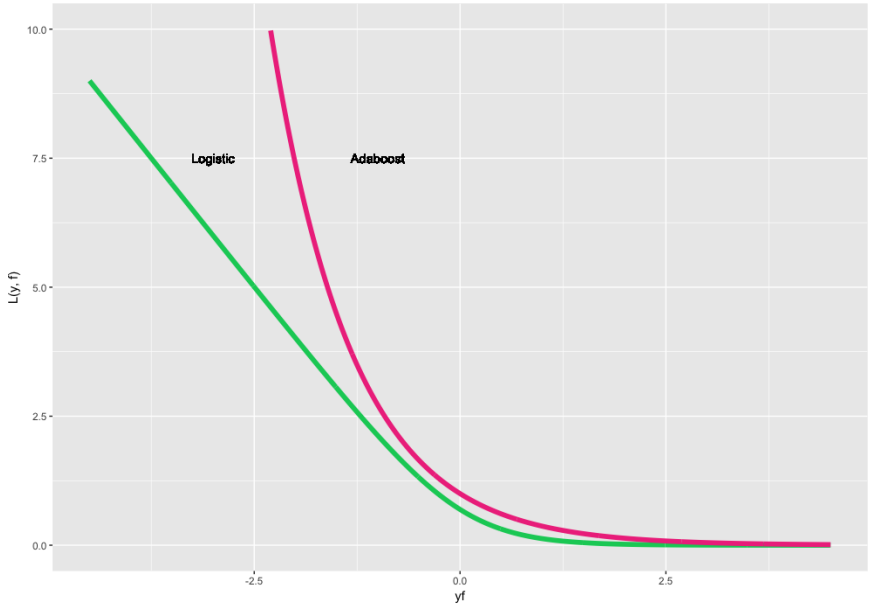
Unfortunately, Huber loss function is supported only by very few popular libraries/packages; h2o supports it, but XGBoost does not. It is relevant to other things that are more exotic like conditional expectiles, but it may still be interesting knowledge.

Classification loss functions

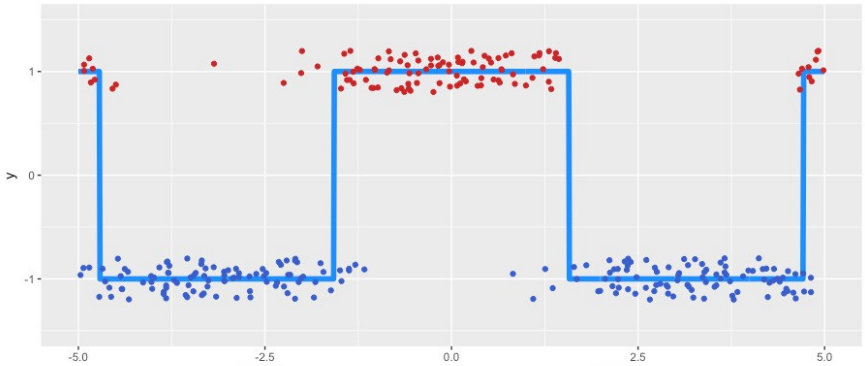
Now, let's look at the binary classification problem. We saw that GBM can even optimize non-differentiable loss functions. Technically, it is possible to solve this problem with a regression *L2* loss, but it wouldn't be correct.

The distribution of the target variable requires us to use log-likehood, so we need to have different loss functions for targets multiplied by their predictions. The most common choices would be the following:

- $L(y,f) = log(1 + exp(-2yf))$ a.k.a. Logistic loss or Bernoulli loss. This has an interesting property that penalizes even correctly predicted classes, which helps not only helps to optimize loss but also to move the classes apart further, even if all classes are predicted correctly.
- $L(y,f) = exp(-yf)$ a.k.a. AdaBoost loss. The classic AdaBoost is equivalent to GBM with this loss function. Conceptually, this function is very similar to logistic loss, but it has a bigger exponential penalization if the prediction is wrong.



Let's generate some new toy data for our classification problem. As a basis, we will take our noisy cosine, and we will use the sign function for classes of the target variable. Our toy data looks like the following (jitter-noise is added for clarity):

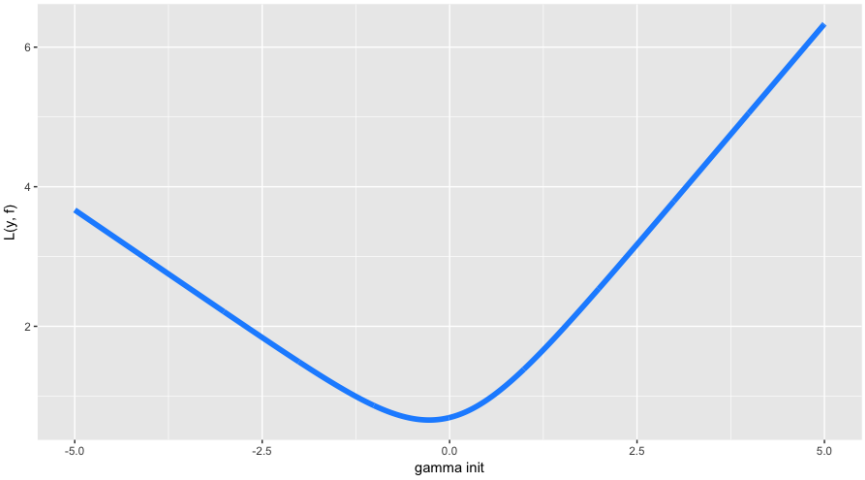


¯_(ツ)_/¯

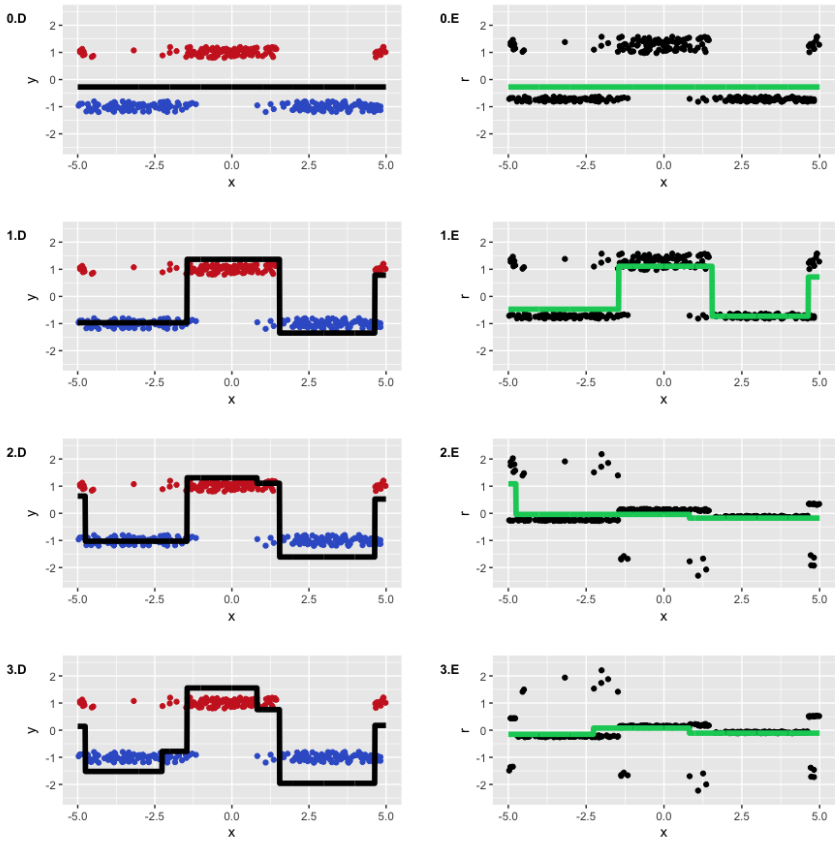
We will use logistic loss to look for what we actually boost. So, again, we put together what we will use for GBM:

- Toy data $\left\{(x_i, y_i)\right\}_{i=1,\dots,300}, y_i \in \{-1, 1\}$ ✓
- Number of iterations $M = 3$ ✓;
- Logistic loss as the loss function, its gradient is computed the following way:
$r_i = \frac{2 \cdot y_i}{1 + exp(2 \cdot y_i \cdot \hat{f}(x_i))}, \quad \text{for } i = 1, \dots, 300$ ✓;
- Decision trees as base algorithms $h(x)$ ✓;
- Hyperparameters of the decision trees: tree's depth is equal to 2 ✓;

This time, the initialization of the algorithm is a little bit harder. First, our classes are imbalanced (63% versus 37%). Second, there is no known analytical formula for the initialization of our loss function, so we have to look for it via search:



Our optimal initial approximation is around -0.273. You could have guessed that it was negative because it is more profitable to predict everything as the most popular class, but there is no formula for the exact value. Now let's finally start GBM, and look what actually happens under the hood:

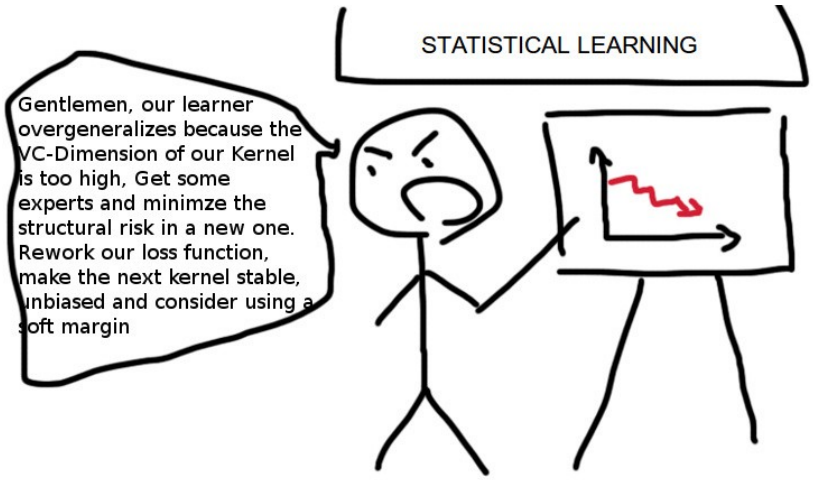


The algorithm successfully restored the separation between our classes. You can see how the "lower" areas are separating because the trees are more confident in the correct prediction of the negative class and how the two steps of mixed classes are forming. It is clear that we have a lot of correctly classified observations and some amount of observations with large errors that appeared due to the noise in the data.

Weights

Sometimes, there is a situation where we want a more specific loss function for our problem. For example, in financial time series, we may want to give bigger weight to large movements in the time series; for churn prediction, it is more useful to predict the churn of clients with

high LTV (or lifetime value: how much money a client will bring in the future).



The statistical warrior would invent their own loss function, write out the gradient for it (for more effective training, include the Hessian), and carefully check whether this function satisfies the required properties. However, there is a high probability of making a mistake somewhere, running up against computational difficulties, and spending an inordinate amount of time on research.

In lieu of this, a very simple instrument was invented (which is rarely remembered in practice): weighing observations and assigning weight functions. The simplest example of such weighting is the setting of weights for class balance. In general, if we know that some subset of data, both in the input variables and in the target variable, has greater importance for our model, then we just assign them a larger weight *w(x,y)*. The main goal is to fulfill the general requirements for weights:

$$w_i \in \mathbb{R},$$
$$w_i \geq 0 \quad \text{for } i = 1, \ldots, n,$$
$$\sum_{i=1}^{n} w_i > 0$$

Weights can significantly reduce the time spent adjusting the loss function for the task we are solving and also encourages experiments with the target models' properties. Assigning these weights is entirely a function of creativity. We simply add scalar weights:

$$L_w(y,f) = w \cdot L(y,f),$$

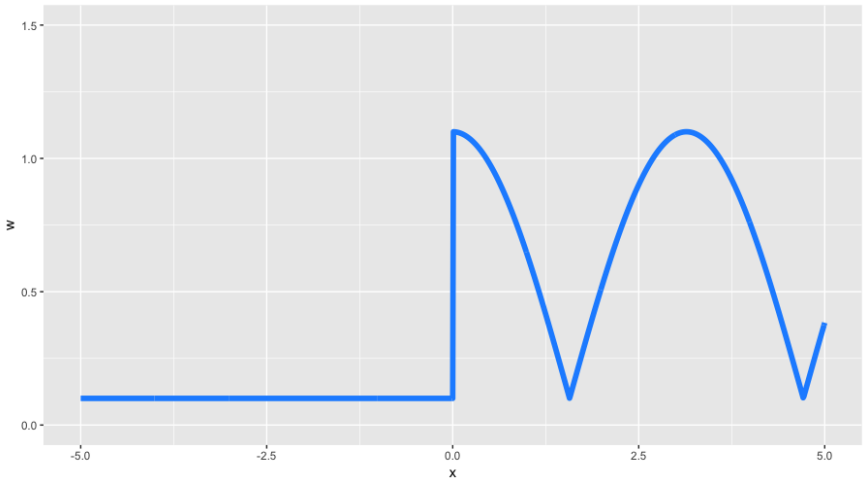
$$r_{it} = -w_i \cdot \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x)=\hat{f}(x)},$$

$$\text{for } i = 1, \ldots, n$$

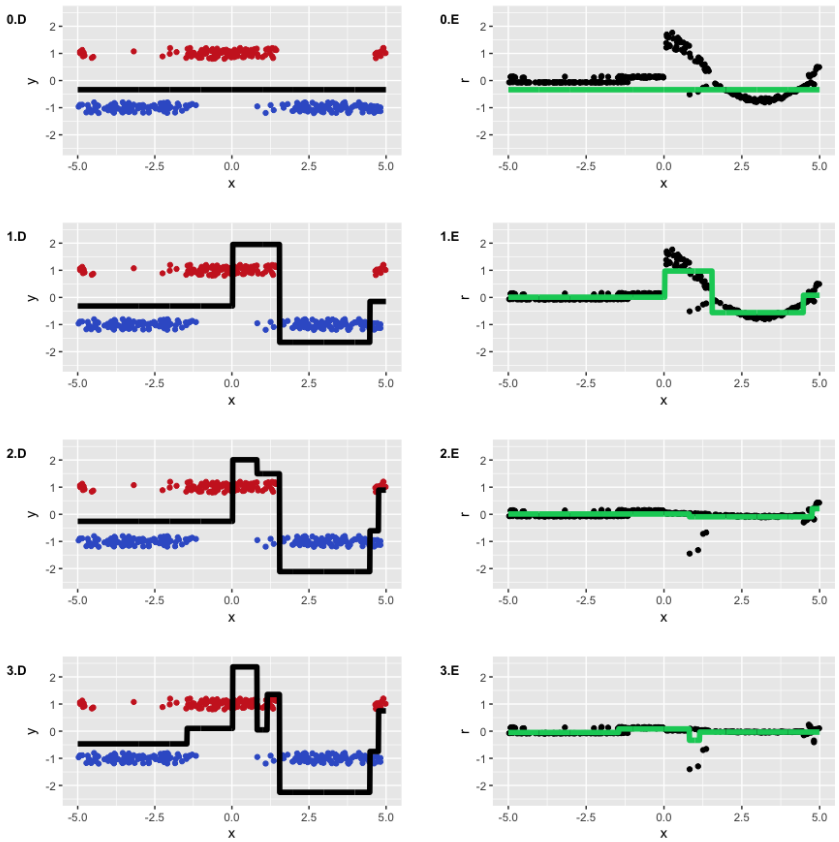
It is clear that, for arbitrary weights, we do not know the statistical properties of our model. Often, linking the weights to the values y can be too complicated. For example, the usage of weights proportional to $|y|$ in *L1* loss function is not equivalent to *L2* loss because the gradient will not take into account the values of the predictions themselves.

We mention all of this so that we can understand our possibilities better. Let's create some very exotic weights for our toy data. We will define a strongly asymmetric weight function as follows:

$$w(x) = \begin{cases} 0.1, & \text{if } x \leq 0 \\ 0.1 + |cos(x)|, & \text{if } x > 0 \end{cases}$$



With these weights, we expect to get two properties: less detailing for negative values of *X* and the form of the function, similar to the initial cosine. We take the other GBM's tunings from our previous example with classification including the line search for optimal coefficients. Let's look what we've got:



We achieved the result that we expected. First, we can see how strongly the pseudo-residuals differ; on the initial iteration, they look almost like the original cosine. Second, the left part of the function's graph was often ignored in favor of the right one, which had larger weights. Third, the function that we got on the third iteration received enough attention and started looking similar to the original cosine (also started to slightly overfit).

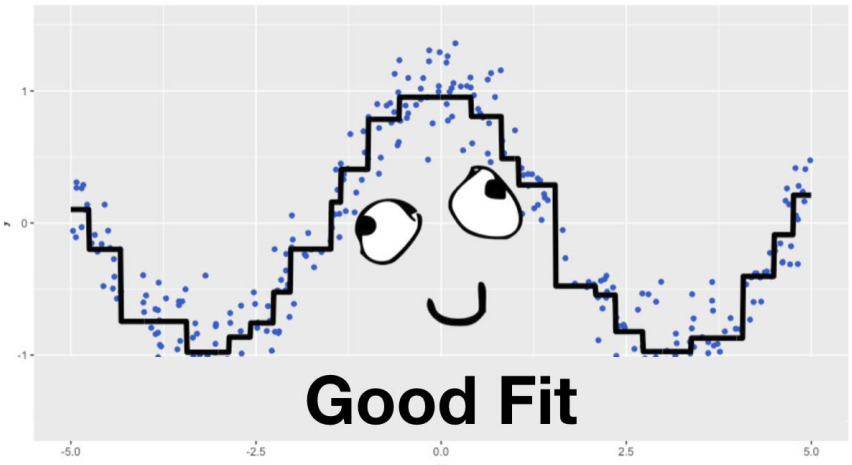
Weights are a powerful but risky tool that we can use to control the properties of our model. If you want to optimize your loss function, it is worth trying to solve a more simple problem first but add weights to the observations at your discretion.

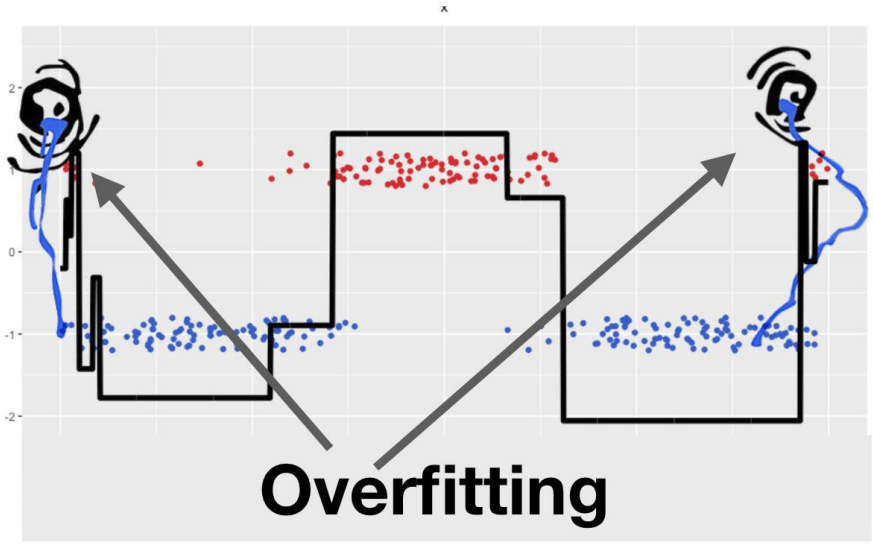
Conclusion

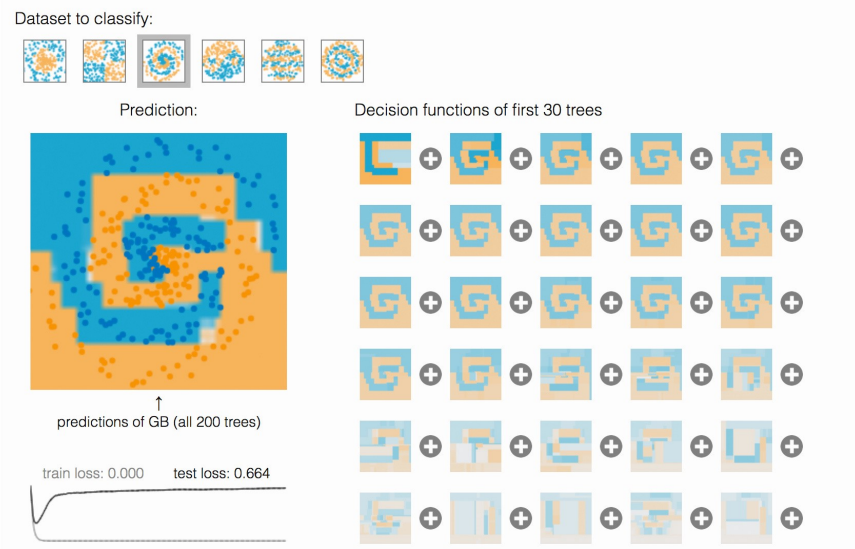
Today, we learned the theory behind gradient boosting. GBM is not just some specific algorithm but a common methodology for building ensembles of models. In addition, this methodology is sufficiently flexible and expandable—it is possible to train a large number of models, taking into consideration different loss-functions with a variety of weighting functions.

Practice and ML competitions show that, in standard problems (except for image, audio, and very sparse data), GBM is often the most effective algorithm (not to mention stacking and high-level ensembles, where GBM is almost always a part of them). Also, there are many adaptations of GBM for Reinforcement Learning (Minecraft, ICML 2016). By the way, the Viola-Jones algorithm, which is still used in computer vision, is based on AdaBoost.

In this article, we intentionally omitted questions concerning GBM's regularization, stochasticity, and hyper-parameters. It was not accidental that we used a small number of iterations $M = 3$ throughout. If we used 30 trees instead of 3 and trained the GBM as described, the result would not be that predictable:







http://arogozhnikov.github.io/2016/07/05/gradient_boosting_playground.html

4. Assignment #10

Your task is to beat at least 2 benchmarks in this Kaggle Inclass competition. Here you won't be provided with detailed instructions. We only give you a brief description of how the second benchmark was achieved using XGBoost.

5. Useful resources

- Original article about GBM by Jerome Friedman
- "Gradient boosting machines, a tutorial", paper by Alexey Natekin, and Alois Knoll
- Chapter in Elements of Statistical Learning from Hastie, Tibshirani, Friedman (page 337)
- Wiki article about Gradient Boosting
- Frontiers tutorial article about GBM
- Video-lecture by Hastie about GBM at h2o.ai conference

. . .

Author: Alexey Natekin, OpenDataScience founder, Machine Learning Evangelist. Translated and edited by Olga Daykhovskaya, Anastasia Manokhina, Egor Polusmak, and Yuanyuan Pao.