# An intuitive explanation of gradient boosting

**Richard Johansson**

## 1 Introduction

This document gives an introduction to the basic ideas of gradient boosting, the learning algorithm used in scikit-learn's `GradientBoostingRegressor` and `GradientBoostingClassifier`, or in the XGBoost software library. We first show how gradient boosting works in a special case: regression with squared error loss, when the loss gradients correspond to residuals. We finally state the general gradient boosting algorithm.

Before reading this, you may want to refresh your knowledge of decision trees. Decision trees are less useful on their own than in ensemble models. For instance, *random forests* are popular tree-based ensembles for classification and regression. Boosting, and in particular AdaBoost and gradient boosting (the topic of this text), are also important types of ensembles that often use decision trees as the base model. For classification, an ensemble of trees will typically produce their output by voting or averaging the outputs of the base classifiers. For regression, the average is used.

## 2 Boosting decision trees for a regression problem

In *boosting*, we build an ensemble of classifiers or regressors incrementally. In each step, we add a new sub-model that tries to compensate for the errors made by the previous sub-models. To avoid overfitting, boosting is typically done using fairly simple sub-models: the classical choice is small decision trees ("decision stumps").

Let's see a simple idea how boosting can be done for a regression task. The first sub-model is a "dummy regressor": just a constant, such as the mean of the output values. We then compute the *residuals* (differences between desired values and predicted values), and train a second sub-model, which is a small decision tree that tries to model these residuals. This process can be repeated: compute the latest residuals, and train another sub-model to try to fix the errors.[1] Figure 1 shows an example of how this boosting procedure gradually improves the fit, as we add more and more sub-models.

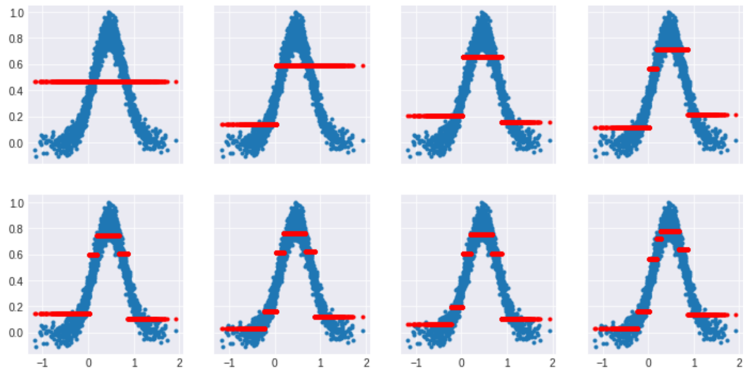

Figure 1: Example of boosting for a regression problem.

If we try to formalize this procedure, we get something like Algorithm 1.

[1]For an example where these steps are carried out in detail, see `http://www.cse.chalmers.se/~richajo/dit866/backup_2019/lectures/l8/Step-by-step%20toy%20boosting%20walkthrough.html`.

**Algorithm 1** Simple boosting algorithm for regression.

```
let $h_0$ be a "dummy" constant model
let $F_0$ be an ensemble just consisting of $h_0$
for $m = 1, \ldots, M$
    for each pair $(\boldsymbol{x}_i, y_i)$ in the training set
        compute the residual $R(y_i, F_{m-1}(\boldsymbol{x}_i)) = y_i - F_{m-1}(\boldsymbol{x}_i)$
    train a regression sub-model $h_m$ on the residuals
    add $h_m$ to the ensemble: $F_m(\boldsymbol{x}) = F_{m-1}(\boldsymbol{x}) + h_m(\boldsymbol{x})$
return the ensemble $F_M$
```

## 3 Gradient boosting

The algorithm that we just described can be generalized, so that we optimize the boosted ensemble with respect to some *loss function*: currently, our algorithm is trying to minimize the squared error, which makes sense, since it's a regression problem. But what if we'd like to optimize some other loss, such as the log loss or cross-entropy for a classification problem, or another regression loss (e.g. absolute error)? This is where *gradient boosting* comes in.

The boosting procedure describes in Algorithm 1 can be seen as a form of *gradient descent*. As you recall, gradient descent optimizes a loss function by applying the following update rule repeatedly:

$$\boldsymbol{x} = \boldsymbol{x} - \eta \nabla_{\text{Loss}}(\boldsymbol{x})$$

where $\nabla_{\text{Loss}}(\boldsymbol{x})$ is the gradient of the loss function that we're trying to optimize, and $\eta$ is a step length (learning rate). What does this have to do with the boosting algorithm? The key observation is that the (negative) gradient of the squared error loss function, evaluated at the output value $\hat{y}$, is the same as the residual (multiplied by 2):

$$\text{Loss}(y_i, \hat{y}) = (y_i - \hat{y})^2 \qquad -\nabla_{\text{Loss}}(\hat{y}) = 2 \cdot (y_i - \hat{y})$$

So the boosting algorithm can be seen as a form of gradient descent that optimizes the squared error loss function, because in each step, it adds a sub-model that tries to mimic the negative gradient of this loss. (The learning rate $\eta$ would be $\frac{1}{2}$ in this case.) We have previously seen gradient descent in the context of linear models and neural networks: the difference here is that we now update the model by adding new sub-models, while previously we updated the model by changing the weights. Figure 2 shows this intuition pictorially: the gradient boosting algorithm adds sub-models to the ensemble incrementally to minimize the loss function (in our case, the squared error loss).

So what if we'd like to optimize some other loss function? In that case, we apply the same recipe: for each training instance, we compute the negative gradient of the loss; then we train a sub-model that tries to imitate this gradient. The difference compared to Algorithm 1 is that these negative gradients are no longer interpretable as residuals: instead, they are now called *pseudo-residuals*.

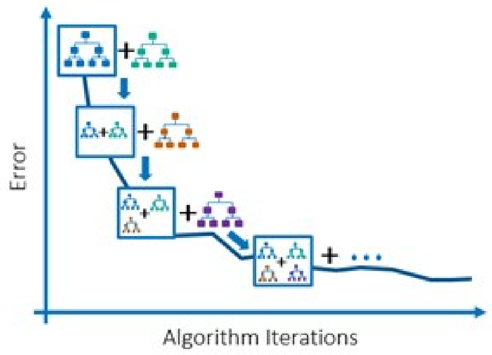

Figure 2: Gradient boosting adds sub-models incrementally to minimize a loss function.

**Algorithm 2** Gradient boosting.

```
let $F_0$ be a "dummy" constant model
for $m = 1, \ldots, M$
  for each pair $(\boldsymbol{x}_i, y_i)$ in the training set
    compute the pseudo-residual $R(y_i, F_{m-1}(\boldsymbol{x}_i))$ = negative gradient of the loss
  train a regression sub-model $h_m$ on the pseudo-residuals
  add $h_m$ to the ensemble: $F_m(\boldsymbol{x}) = F_{m-1}(\boldsymbol{x}) + \eta \cdot h_m(\boldsymbol{x})$
return the ensemble $F_M$
```

Algorithm 2 shows the pseudocode of the general gradient boosting algorithm. A few things to keep in mind here:

- Note that gradient boosting always uses *regression* trees, even if we are solving a classification problem. For classification, we convert the output scores into probabilities by applying the sigmoid or softmax.
- The learning rate $\eta$ can be controlled to reduce the risk of overfitting. (In scikit-learn, the default value is 0.1.) In practice, it is often adjusted automatically.