1 PART I ENSEMBLE LEARNING

1.1 Ensemble Learning

Ensemble learning usually produces more accurate solutions than a single model would.

Ensemble Learning is a technique that create multiple models and then combine them them to produce improved results. Ensemble learning usually produces more accurate solutions than a single model would.

- Ensemble learning methods is applied to regression as well as classification.
- Ensemble learning for regression creates multiple repressors i.e. multiple regression models such as linear, polynomial, etc.
- Ensemble learning for classification creates multiple classifiers i.e. multiple classification models such as logistic, decision tress, KNN, SVM, etc.

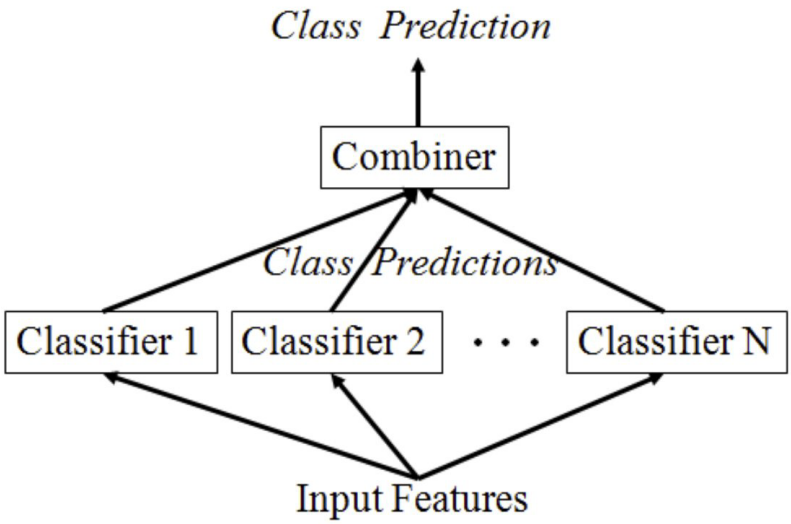


Figure 1: Ensemble learning view

1.2 Which components to combine?

- different learning algorithms
- same learning algorithm trained in different ways
- same learning algorithm trained the same way

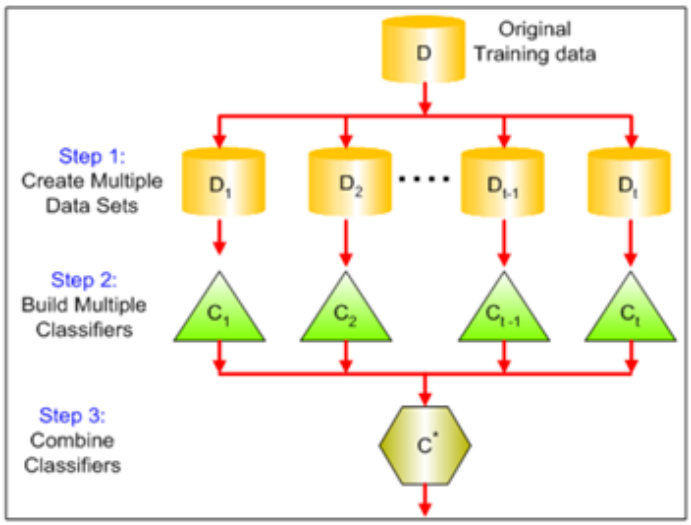
There are two steps in ensemble learning:

- Multiples machine learning models were generated using same or different machine learning algorithm. These are called "base models". The prediction perform on the basis of base models.
- Techniques/Methods in ensemble learning

1.3 Model Combination Schemes - Combining Multiple Learners

No one single algorithm is always the most accurate. Now, we are going to discuss models composed of multiple learners that complement each other so that by combining them, we attain higher accuracy.

There are also different ways the multiple base-learners are combined to generate the final output:
Figure2: General Idea - Combining Multiple Learners



Type *Markdown* and LaTeX: α^2

1.3.1 Multiexpert combination

Multiexpert combination methods have base-learners that work in parallel. These methods can in turn be divided into two:

- In the global approach, also called learner fusion, given an input, all base-learners generate an output and all these outputs are used.

Examples are voting and stacking.

- In the local approach, or learner selection, for example, in mixture of experts, there is a gating model, which looks at the input and chooses one (or very few) of the learners as responsible for generating the output.

1.3.2 Multistage combination

Multistage combination methods use a serial approach where the next base-learner is trained with or tested on only the instances where the previous base-learners are not accurate enough. The idea is that the base-learners (or the different representations they use) are sorted in increasing complexity so that a complex base-learner is not used (or its complex representation is not extracted) unless the preceding simpler base-learners are not confident.

1.3.2.1 An example is cascading.

Let us say that we have L base-learners. We denote by $\mathrm{d_j}(x)$ the prediction of base-learner M_j given the arbitrary dimensional input x. In the case of multiple representations, each M_j uses a different input representation x_j. The final prediction is calculated from the predictions of the base-learners:

$$y = f(d_1, d_2, \ldots, d_L \mid \Phi) \tag{1}$$

where $f(\cdot)$ is the combining function with Φ denoting its parameters.

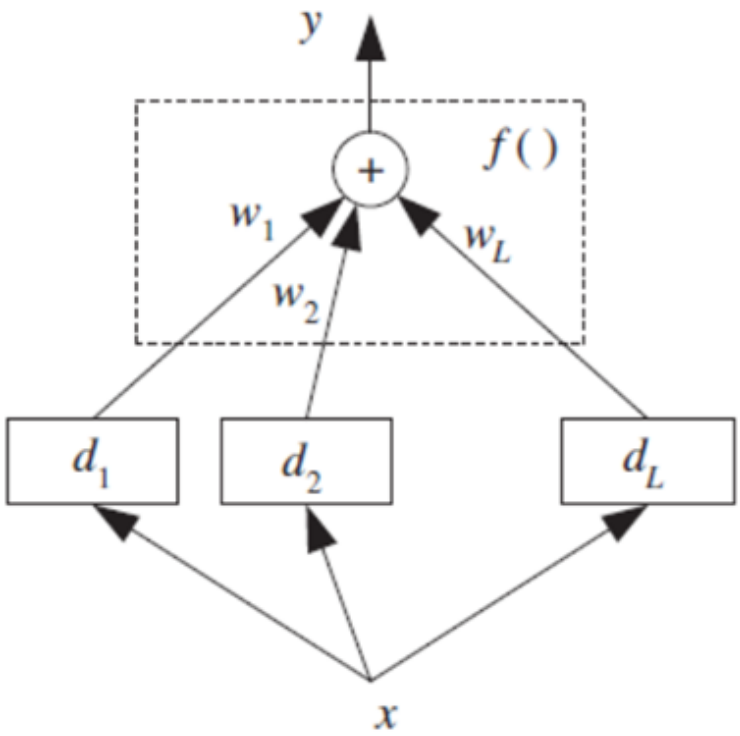


Figure 3: Base-learners are dj and their outputs are combined using $f(\cdot)$.

This is for a single output; in the case of classification, each base-learner has K outputs that are separately used to calculate y_i, and then we choose the maximum. Note that here all learners observe the same input; it may be the case that different learners observe different representations of the same input object or event

When there are K outputs, for each learner there are $d_j(x), i = 1, \ldots, K, j = 1, \ldots, L$, and, combining them, we also generate K values, $y_i i = 1, \ldots, K$ and then for example in classification, we choose the class with the maximum yi value:

Choose C_i if $y_i = \max_{k-1}^{K} y_k$

1.4 Voting

The simplest way to combine multiple classifiers is by voting, which corresponds to taking a linear combination of the learners, Refer figure 3.

$$y_i = \sum_j w_j d_{ji} \text{ where } w_j \geq 0, \sum_j w_j = 1 \tag{2}$$

This is also known as ensembles and linear opinion pools. In the simplest case, all learners are given equal weight and we have simple voting that corresponds to taking an average. Still, taking a (weighted) sum is only one of the possibilities and there are also other combination rules, as shown in table 1. If the outputs are not posterior probabilities, these rules require that outputs be normalized to the same scale

Table 1 - Classifier combination rules

Rule	Fusion function $f(\cdot)$
Sum	$y_i = \frac{1}{L} \sum_{j=1}^{L} d_{ji}$
Weighted sum	$y_i = \sum_j w_j d_{ji}, w_j \geq 0, \sum_j w_j = 1$
Median	$y_i = \text{median}_j \, d_{ji}$
Minimum	$y_i = \min_j d_{ji}$
Maximum	$y_i = \max_j d_{ji}$
Product	$y_i = \prod_j d_{ji}$

An example of the use of these rules is shown in table 2, which demonstrates the effects of different rules. Sum rule is the most intuitive and is the most widely used in practice. Median rule is more robust to outliers; minimum and maximum rules are pessimistic and optimistic, respectively. With the product rule, each learner has veto power; regardless of the other ones, if one learner has an output of 0 , the overall output goes to 0. Note that after the combination rules, y_i do not necessarily sum up to 1.

Table 2: Example of combination rules on three learners and three classes

	C_1	C_2	C_3
d_1	0.2	0.5	0.3
d_2	0.0	0.6	0.4
d_3	0.4	0.4	0.2
Sum	0.2	0.5	0.3
Median	0.2	0.5	0.4
Minimum	0.0	0.4	0.2
Maximum	0.4	0.6	0.4
Product	0.0	0.12	0.032

In weighted sum, d_{ji} is the vote of learner j for class C_i and w_j is the weight of its vote. Simple voting is a special case where all voters have equal weight, namely, $w_j = 1/L$. In classification, this is called plurality voting where the class having the maximum number of votes is the winner.

When there are two classes, this is majority voting where the winning class gets more than half of the votes. If the voters can also supply the additional information of how much they vote for each class (e.g., by the posterior probability), then after normalization, these can be used as weights in a weighted voting scheme. Equivalently, if d_{ji} are the class posterior probabilities, $\mathrm{P}\left(\mathrm{C_i} \mid \mathrm{x}, \mathrm{M_j}\right)$, then we can just sum them up $\left(w_j = 1/L\right)$ and choose the class with maximum y_i.

In the case of regression, simple or weighted averaging or median can be used to fuse the outputs of base-regressors. Median is more robust to noise than the average.

Another possible way to find w_j is to assess the accuracies of the learners (regressor or classifier) on a separate validation set and use that information to compute the weights, so that we give more weights to more accurate learners.

Voting schemes can be seen as approximations under a Bayesian framework with weights approximating prior model probabilities, and model decisions approximating model-conditional likelihoods.

$$P(C_i \mid x) = \sum_{\text{all models } \mathcal{M}_j} P\left(C_i \mid x, \mathcal{M}_j\right) P\left(\mathcal{M}_j\right) \tag{3}$$

Simple voting corresponds to a uniform prior. If we have a prior distribution preferring simpler models, this would give larger weights to them. We cannot integrate over all models; we only choose a subset for which we believe $P\left(M_j\right)$ is high, or we can have another Bayesian step and calculate $P\left(C_i \mid x, M_j\right)$, the probability of a model given the sample, and sample high probable models from this density.

Let us assume that d_j are iid with expected value $E\left[d_j\right]$ and variance $\operatorname{Var}\left(d_j\right)$, then when we take a simple

2 PART 2 - BAGGING AND RANDOM FOREST

Definition 1 ***Bootstrap aggregating****, also called* ***bagging*** *(from bootstrap aggregating), is a machine learning ensemble meta-algorithm designed to improve the stability and accuracy of machine learning algorithms used in statistical classification and regression. It also reduces variance and helps to avoid overfitting. Although it is usually applied to decision tree methods, it can be used with any type of method. Bagging is a special case of the model averaging approach.*

2.1 Bagging ensemble

Bootstrap aggregating, often abbreviated as bagging, involves having each model in the ensemble vote with equal weight. In order to promote model variance, bagging trains each model in the ensemble using a randomly drawn subset of the training set. As an example, the random forest algorithm combines random decision trees with bagging to achieve very high classification accuracy.

The simplest method of combining classifiers is known as bagging, which stands for bootstrap aggregating, the statistical description of the method. This is fine if you know what a bootstrap is, but fairly useless if you don't. A bootstrap sample is a sample taken from the original dataset with replacement, so that we may get some data several times and others not at all. The bootstrap sample is the same size as the original, and lots and lots of these samples are taken: B of them, where B is at least 50 , and could even be in the thousands. The name bootstrap is more popular in computer science than anywhere else, since there is also a bootstrap loader, which is the first program to run when a computer is turned on. It comes from the nonsensical idea of 'picking yourself up by your bootstraps,' which means lifting yourself up by your shoelaces, and is meant to imply starting from nothing.

Bootstrap sampling seems like a very strange thing to do. We've taken a perfectly good dataset, mucked it up by sampling from it, which might be good if we had made a smaller dataset (since it would be faster), but we still ended up with a dataset the same size. Worse, we've done it lots of times. Surely this is just a way to burn up computer time without gaining anything. The benefit of it is that we will get lots of learners that perform slightly differently, which is exactly what we want for an ensemble method. Another benefit is that estimates of the accuracy of the classification function can be made without complicated analytic work, by throwing computer resources at the problem (technically, bagging is a variance reducing algorithm; the meaning of this will become clearer when we talk about bias and variance). Having taken a set of bootstrap samples, the bagging method simply requires that we fit a model to each dataset, and then combine them by taking the output to be the majority vote of all the classifiers. A NumPy implementation is shown next, and then we will look at a simple example.

2.2 Bagging summary

- Bagging is an ensemble method involving training the same algorithm many times using different subsets sampled from the training data
- In bagging, it uses same algorithm (only one algorithm is used)
- However the model is not training on entire training set
- Instead each model is trained on different subset of data
- Bagging: Bootstrap Aggregation.
- Uses a technique known as the bootstrap.
- Reduces variance of individual models in the ensemble.
- For example, suppose a training dataset contains 3 parts - a,b,c.
- It create subset by method sample by replacement. For example aaa,aab,aba,acc,aca etc.
- On this subset, the models are trained.

2.2.1 Bagging Classication

- Aggregates predictions by majority voting (Final model is selected by voting).
- BaggingClassifier in scikit-learn.

2.2.2 Bagging Regression

- Aggregates predictions through averaging (Final model is selected by avergaing).
- BaggingRegressor in scikit-learn.

2.2.3 Bagging limitations

- Some instances may be sampled several times for one model,
- Other instances may not be sampled at all.

2.2.4 Out Of Bag (OOB) instances

- On average,for each model, 63% of the training instances are sampled.
- The remaining 37% constitute the OOB instances.
- Since OOB instances are not seen by the model during training.

- This can be used to estimate the performance of the model without the need of cross validation.
- This technique is known as OOB evaluation

2.2.5 Random Forest

- Another ensemble model
- Base estimator: Decision Tree
- Each estimator is trained on a different bootstrap sample having the same size as the training set
- RF introduces further randomization in the training of individual trees
- d features are sampled at each node without replacement (d < total number of features)

2.2.6 Random Forests Classication:

- Aggregates predictions by majority voting
- RandomForestClassifier in scikit-learn

2.2.7 Random Forests Regression:

- Aggregates predictions through averaging
- RandomForestRegressor in scikit-learn

2.2.8 Feature Importance

- Tree-based methods: enable measuring the importance of each feature in prediction.
- In sklearn : how much the tree nodes use a particular feature (weighted average) to reduce impurity accessed using the attribute feature *importance*

Exercise 1 Implement the following algorithm

3 Compute bootstrap samples

samplePoints = np.random.randint(0,nPoints,(nPoints,nSamples)) classifiers = [] for i in range(nSamples): sample = [] sampleTarget = [] for j in range(nPoints): sample.append(data[samplePoints[j,i sampleTarget.append(targets[samplePoints[j,i]])

4 Train classifiers

classifiers.append(self.tree.make_tree(sample,sampleTarget,features))

```
### RANDOM FORESTS
A random forest is an ensemble learning method where multiple decision
trees are constructed and then they are merged to get a more accurate
prediction.

```

```
If there is one method in machine learning that has grown in popularity
over the last few years, then it is the idea of random forests. The
concept has been around for longer than that, with several different
people inventing variations, but the name that is most strongly attached
to it is that of Breiman.

Figure 3: Example of random forest with majority voting

![]
(https://cdn.mathpix.com/snip/images/JLhghVzuzGlQQaSZmO3ac9WdOJ31u9sR2SB
3r3Rrsz0.original.fullsize.png)

The idea is largely that if one tree is good, then many trees (a forest)
should be better, provided that there is enough variety between them.
The most interesting thing about a random forest is the ways that it
creates randomness from a standard dataset. The first of the methods
that it uses is the one that we have just seen: bagging. If we wish to
create a forest then we can make the trees different by training them on
slightly different data, so we take bootstrap samples from the dataset
for each tree. However, this isn't enough randomness yet. The other
obvious place where it is possible to add randomness is to limit the
choices that the decision tree can make. At each node, a random subset
of the features is given to the tree, and it can only pick from that
subset rather than from the whole set.

**Algorithm**
Here is an outline of the random forest algorithm.
1. The random forests algorithm generates many classification trees.
Each tree is generated as follows:

    a) If the number of examples in the training set is $N$, take a
sample of $N$ examples at random - but with replacement, from the
original data. This sample will be the training set for generating the
tree.

    b) If there are $M$ input variables, a number $m$ is specified such
that at each node, $m$ variables are selected at random out of the $M$
and the best split on these $m$ is used to split the node. The value of
$m$ is held constant during the generation of the various trees in the
forest.

   c) Each tree is grown to the largest extent possible.

2. To classify a new object from an input vector, put the input vector
down each of the trees in the forest. Each tree gives a classification,
and we say the tree "votes" for that class. The forest chooses the
classification
```

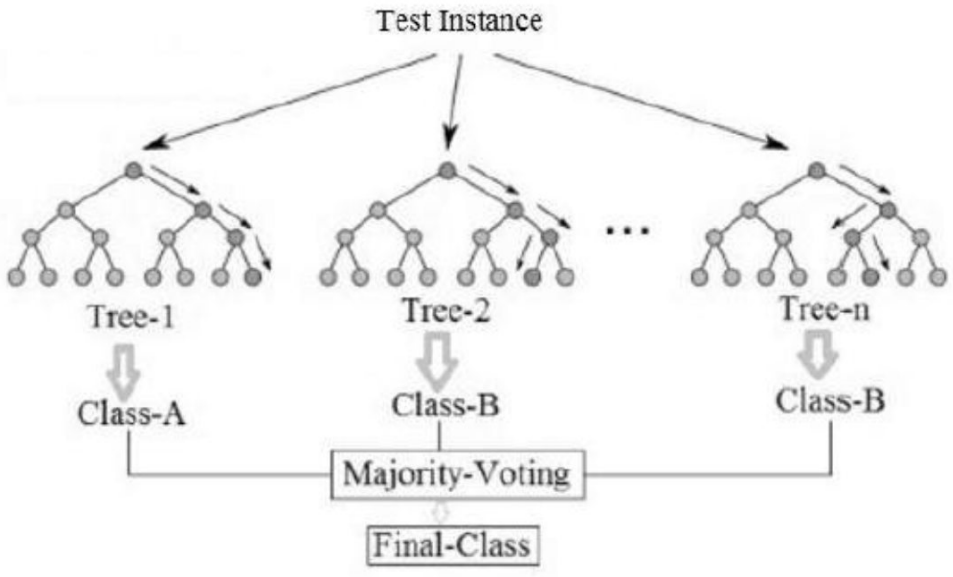
4.0.1 RANDOM FORESTS

A random forest is an ensemble learning method where multiple decision trees are constructed and then they are merged to get a more accurate prediction.

If there is one method in machine learning that has grown in popularity over the last few years, then it is the idea of random forests. The concept has been around for longer than that, with

several different people inventing variations, but the name that is most strongly attached to it is that of Breiman.

Figure 3: Example of random forest with majority voting



The idea is largely that if one tree is good, then many trees (a forest) should be better, provided that there is enough variety between them. The most interesting thing about a random forest is the ways that it creates randomness from a standard dataset. The first of the methods that it uses is the one that we have just seen: bagging. If we wish to create a forest then we can make the trees different by training them on slightly different data, so we take bootstrap samples from the dataset for each tree. However, this isn't enough randomness yet. The other obvious place where it is possible to add randomness is to limit the choices that the decision tree can make. At each node, a random subset of the features is given to the tree, and it can only pick from that subset rather than from the whole set.

Algorithm Here is an outline of the random forest algorithm.

1. The random forests algorithm generates many classification trees. Each tree is generated as follows:

   a) If the number of examples in the training set is N, take a sample of N examples at random - but with replacement, from the original data. This sample will be the training set for generating the tree.

   b) If there are M input variables, a number m is specified such that at each node, m variables are selected at random out of the M and the best split on these m is used to split the node. The value of m is held constant during the generation of the various trees in the forest.

   c) Each tree is grown to the largest extent possible.
2. To classify a new object from an input vector, put the input vector down each of the trees in the forest. Each tree gives a classification, and we say the tree "votes" for that class. The forest chooses the classification

Strengths

The following are some of the important strengths of random forests.

- It runs efficiently on large data bases.
- It can handle thousands of input variables without variable deletion.
- It gives estimates of what variables are important in the classification.
- It has an effective method for estimating missing data and maintains accuracy when a large proportion of the data are missing.
- Generated forests can be saved for future use on other data.
- Prototypes are computed that give information about the relation between the variables and the classification.
- The capabilities of the above can be extended to unlabeled data, leading to unsupervised clustering, data views and outlier detection.
- It offers an experimental method for detecting variable interactions.
- Random forest run times are quite fast, and they are able to deal with unbalanced and missing data.
- They can handle binary features, categorical features, numerical features without any need for scaling.
- *Weaknesses**
- A weakness of random forest algorithms is that when used for regression they cannot predict beyond the range in the training data, and that they may over-fit data sets that are particularly noisy.
- The sizes of the models created by random forests may be very large. It may take hundreds of megabytes of memory and may be slow to evaluate.
- Random forest models are black boxes that are very hard to interpret.

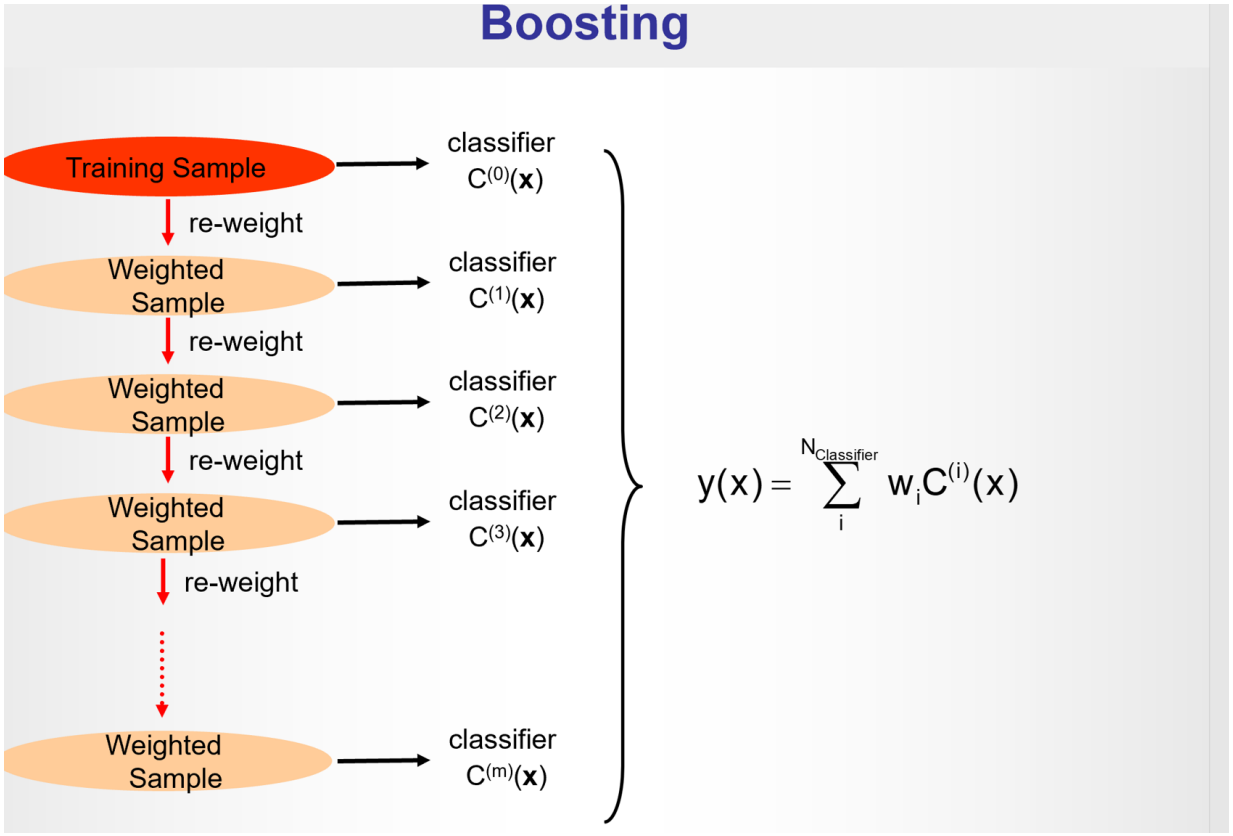
5 Boosting

- Boosting: train next learner on mistakes made by previous learner(s)

In bagging, generating complementary base-learners is left to chance and to the instability of the learning method. In boosting, we actively try to generate complementary base-learners by training the next learner on the mistakes of the previous learners. The original boosting algorithm combines three weak learners to generate a strong learner. A weak learner has error probability less than $1/2$, which makes it better than random guessing on a two-class problem, and a strong learner has arbitrarily small error probability.

Original Boosting Concept

Given a large training set X, we randomly divide it into three. We use X_1 and train d_1. We then take X_2 and feed it to d_1. We take all instances misclassified by d_1 and also as many instances on which d_1 is correct from X_2, and these together form the training set of d_2. We then take X_3 and feed it to d_1 and d_2. The instances on which d_1 and d_2 disagree form the training set of d_3. During testing, given an instance, we give it to d_1 and d_2; if they agree, that is the response, otherwise the response of d_3 is taken as the output.



Algorithm

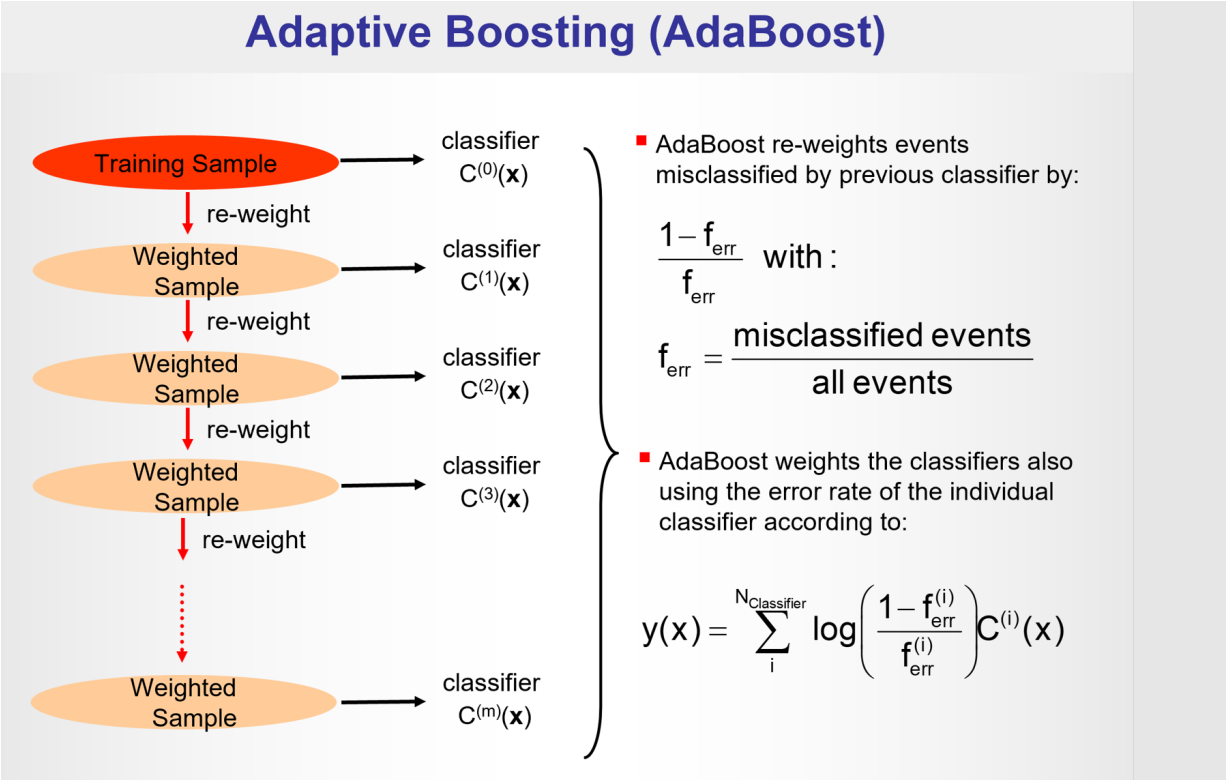
1. Split data X into $\{X_1, X_2, X_3\}$
2. Train d_1 on X_1
    - Test d_1 on X_2
3. Train d_2 on d_1 's mistakes on X_2
    - Test d_1 and d_2 on X_3
4. Train d_3 on disagreements between d_1 and d_2
    - Testing: apply d_1 and d_2; if disagree, use d_3
    - Drawback: need large X

overall system has reduced error rate, and the error rate can arbitrarily be reduced by using such systems recursively, that is, a boosting system of three models used as d_j in a higher system.

Though, it is quite successful, the disadvantage of the original boosting method is that it requires a very large training sample. The sample should be divided into three and furthermore, the second and third classifiers are only trained on a subset on which the previous ones errors. So unless one has a quite large training set, d_2 and d_3 will not have training sets of reasonable size.

5.1 AdaBoost

Freund and Schapire (1996) proposed a variant, named AdaBoost, short for adaptive boosting, that uses the same training set over and over and thus need not be large, but the classifiers should be simple so that they do not overfit. AdaBoost can also combine an arbitrary number of base learners, not three.



5.1.1 AdaBoost algorithm

Training:

For all $\{x^t, r^t\}_{t-1}^N \in X$, initialize $p_1^t = 1/N$

For all base-learners $j = 1, \ldots, L$

Randomly draw x_j from X with probabilities p_j^t

Train d_j using χ_j

For each (x^t, r^t), calculate $y_j^t - d_j(x^t)$

Calculate error rate: $\epsilon_j \leftarrow \sum_t p_j^t \cdot 1(y_j^t \neq r^t)$

If $\epsilon_j > 1/2$, then $L \leftarrow j - 1$; stop

$\beta_j \leftarrow \epsilon_j / (1 - \epsilon_j)$

For each (x^t, r^t), decrease probabilities if correct:

If $y_j^t = r^t$, then $p_{j+1}^t \leftarrow \beta_j p_j^t$ Else $p_{j+1}^t \leftarrow p_j^t$

Normalize probabilities:

$Z_j \leftarrow \sum_t p_{j+1}^t; p_{j+1}^t - p_{j+1}^t / Z_j$

Testing:

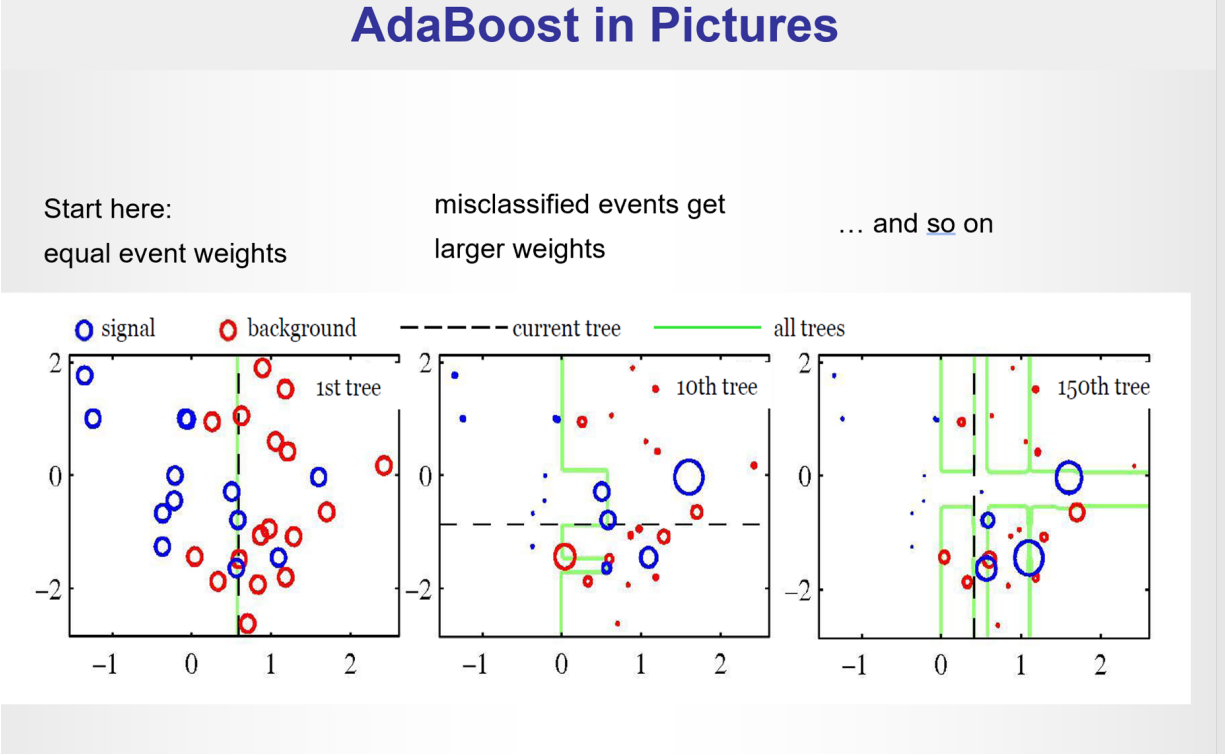
Given x, calculate $d_j(x), j = 1, \ldots, L$

Calculate class outputs, $i = 1, \ldots, K$:

$$y_i = \sum_{j-1}^{L} \left(\log \frac{1}{\beta_j} \right) d_{ji}(x)$$

The idea is to modify the probabilities of drawing the instances as a function of the error. Let us say p_j^t denotes the probability that the instance pair $\left(x^t, r^t\right)$ is drawn to train the jth base-learner. Initially, all $p_1^t = 1/N$. Then we add new base-learners as follows, starting from $j = 1 : \epsilon_j$ denotes the error rate of d_j. AdaBoost requires that learners are weak, that is, $\epsilon_j < 1/2, \forall j$; if not, we stop adding new baselearners. Note that this error rate is not on the original problem but on the dataset used at step j. We define $Bj = \epsilon_j / \left(1 - \epsilon_j\right) < 1$, and we set $p_{j+1}^t = B_j p_j^t$ if d_j correctly classifies xt; otherwise, $p_{j+1}^t = p_j^t$. Because p_{j+1}^t should be probabilities, there is a normalization where we divide p_{j+1}^t by $\sum t p_{j+1}^t$, so that they sum up to 1. This has the effect that the probability of a correctly classified instance is decreased, and the probability of a misclassified instance increases. Then a new sample of the same size is drawn from the original sample according to these modified probabilities, p_{j+1}^t with replacement, and is used to train $\mathrm{d_{j+1}}$.

This has the effect that d_{j+1} focuses more on instances misclassified by d_j; that is why the base-learners are chosen to be simple and not accurate, since otherwise the next training sample would contain only a few outlier and noisy instances repeated many times over. For example, with decision trees, decision stumps, which are trees grown only one or two levels, are used. So it is clear that these would have bias but the decrease in variance is larger and the overall error decreases. An algorithm like the linear discriminant has low variance, and we cannot gain by AdaBoosting linear discriminants.



5.2 Gradient Boosting

The notebook explores the [Gradient Boosting] algorithm (https://en.wikipedia.org/wiki/Gradient_boosting (https://en.wikipedia.org/wiki/Gradient_boosting)).

5.3 Lab exercise

Exercise 2 Gradient boosting algorithms are considered important alternatives to neural nets (to be discussed next) for solving classification and regression problems. You are supposed to read the concept of gradient boosting discussed in this notebook and the related lecture that I have posted. The main example in this notebook studies the gradient based algorithms for regression problem implemented utilizing sklearn and other libraries. In this lab you are supposed to modify the main example to solve the classification problem for the pima-indian-diabetes dataset utilizing the `diabetes.csv` dataset and study the effect of various parameters as it is done in the regression example presented here. You should modify the comments presented after each experiment.

You can do this lab with your team provided you keep safe distance

5.4 Gradient Boosting Concept

Boosting algorithms play a crucial role in dealing with **bias variance trade-off**.

- Unlike **bagging algorithms**, which only controls for **high variance in a model**.
- **boosting** controls both the aspects **(bias & variance)**, and is considered to be more effective.

A sincere **understanding of GBM** here should give you much needed **confidence to deal with such critical issues.**

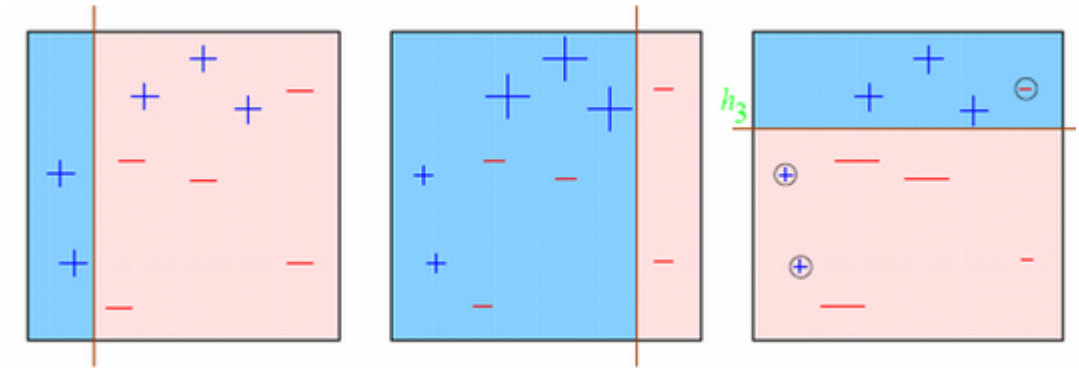
5.4.1 Table of Contents

- How Boosting Works?
- Understanding GBM Parameters
- Tuning Parameters (with Example)

5.4.2 How Boosting Works?

Boosting is a sequential technique which works on the principle of **ensemble**. It combines a set of **weak learners** and delivers improved prediction accuracy. At any instant t, the model outcomes are weighed based on the outcomes of previous instant t-1. The outcomes predicted correctly are given a lower weight and the ones miss-classified are weighted higher. This technique is followed for a classification problem while a similar technique is used for regression.

Let’s understand it visually:



(https://www.analyticsvidhya.com/wp-content/uploads/2016/02/boosting.png)

Observations:

1. **Box 1: Output of First Weak Learner (From the left)**
    - Initially all points have same weight (denoted by their size).
    - The decision boundary predicts 2 +ve and 5 -ve points correctly.
2. **Box 2: Output of Second Weak Learner**
    - The points classified correctly in box 1 are given a lower weight and vice versa.
    - The model focuses on high weight points now and classifies them correctly. But, others are misclassified now.

Similar trend can be seen in box 3 as well. This continues for many iterations. In the end, all models are given a weight depending on their accuracy and a consolidated result is generated.

Did I whet your appetite ? Good. Refer to these articles (focus on GBM right now):

- Learn Gradient Boosting Algorithm for better predictions (with codes in R) (https://www.analyticsvidhya.com/blog/2015/09/complete-guide-boosting-methods/)
- Quick Introduction to Boosting Algorithms in Machine Learning (https://www.analyticsvidhya.com/blog/2015/11/quick-introduction-boosting-algorithms-machine-learning/)
- Getting smart with Machine Learning – AdaBoost and Gradient Boost (https://www.analyticsvidhya.com/blog/2015/05/boosting-algorithms-simplified/)

Question 1: What is Weak Learner? (https://www.analyticsvidhya.com/blog/2015/11/quick-introduction-boosting-algorithms-machine-learning/)

Question 2: How boosting identify weak rules? (https://www.analyticsvidhya.com/blog/2015/11/quick-introduction-boosting-algorithms-machine-learning/)
Answer 2: To find weak rule, we apply base learning (ML) algorithms with a different distribution. Each time base learning algorithm is applied, it generates a new weak prediction rule. This is an iterative process. After many iterations, the boosting algorithm combines these weak rules into a single strong prediction rule.

Question 3: [How do we choose different distribution for each round?' (https://www.analyticsvidhya.com/blog/2015/11/quick-introduction-boosting-algorithms-machine-learning/)]
Answer 3: here are the following steps:

Step 1: The base learner takes all the distributions and assign equal weight or attention to each observation.

Step 2: If there is any prediction error caused by first base learning algorithm, then we pay higher attention to observations having prediction error. Then, we apply the next base learning algorithm.

Step 3: Iterate Step 2 till the limit of base learning algorithm is reached or higher accuracy is achieved.

Finally, it combines the outputs from weak learner and creates a strong learner which eventually improves the prediction power of the model. Boosting pays higher focus on examples which are mis-classified or have higher errors by preceding weak rules.

5.5 Gradient Boosting for Regression

You are given $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, and the task is to fit a model $F(x)$ to minimize square loss.

Suppose your friend wants to help you and gives you a model F.

You check his model and find the model is good but not perfect.

There are some mistakes: $F(x_1) = 0.8$, while $y_1 = 0.9$, and $F(x_2) = 1.4$ while $y_2 = 1.3 \ldots$.

How can you improve this model?

Rule of the game:

- "You are not allowed to remove anything from F or change any parameter in F.
- You can add an additional model (regression tree) h to F, so the new prediction will be $F(x) + h(x)$

5.5.1 Simple solution:

Or, equivalently, you wish

$$\begin{aligned} h(x_1) &= y_1 - F(x_1) \\ h(x_2) &= y_2 - F(x_2) \\ &\cdots \\ h(x_n) &= y_n - F(x_n) \end{aligned} \tag{4}$$

Can any regression tree h achieve this goal perfectly?

Maybe not....

But some regression tree might be able to do this approximately.

How?

Just fit a regression tree h to data $(x_1, y_1 - F(x_1)), (x_2, y_2 - F(x_2)), \ldots, (x_n, y_n - F(x_n))$
Congratulations, you get a better model!

5.5.2 Simple solution:

$y_i - F(x_i)$ are called residuals. These are the parts that existing model F cannot do well.

The role of h is to compensate the shortcoming of existing model F

If the new model $F + h$ is still not satisfactory, we can add another regression tree...

We are improving the predictions of training data, is the procedure also useful for test data?

Yes! Because we are building a model, and the model can be applied to test data as well.

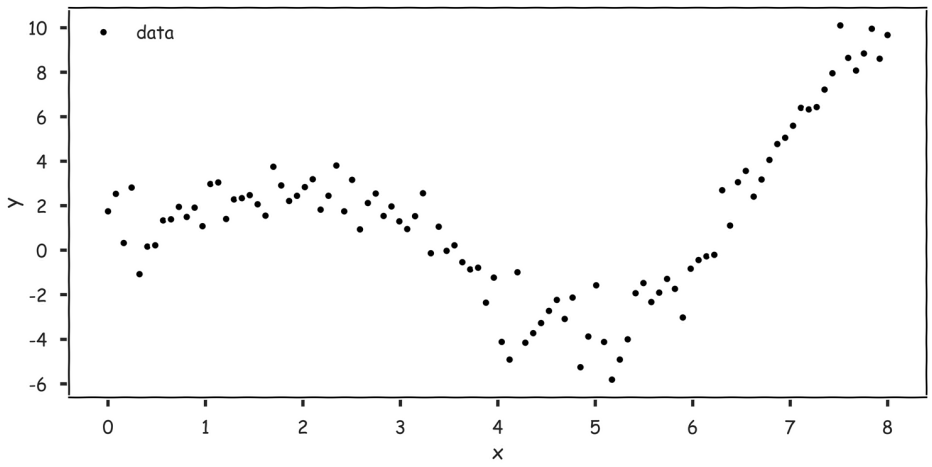
How is this related to gradient descent?

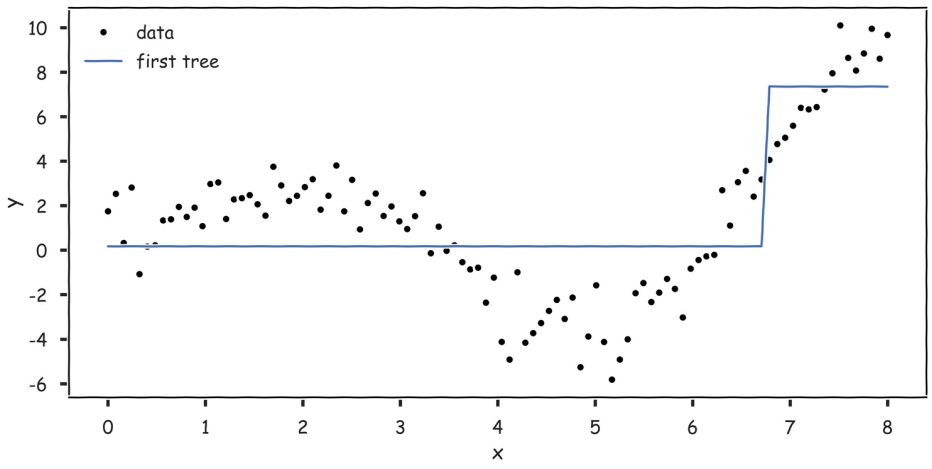
5.6 Gradient Boosting for Regression

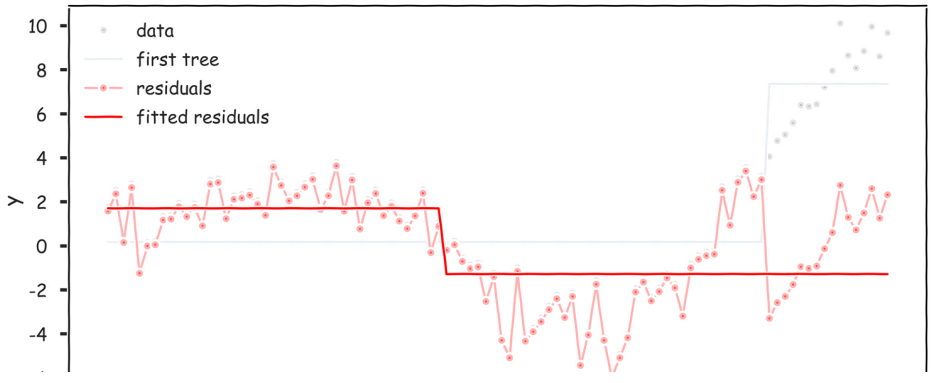
Gradient boosting is a method for iteratively building a complex regression model T by adding simple models. Each new simple model added to the ensemble compensates for the weaknesses of the current ensemble.

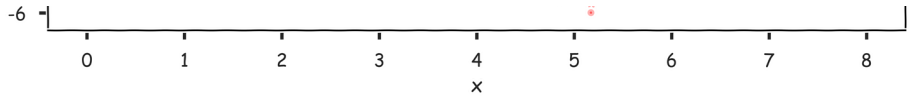
1. Fit a simple model $T^{(0)}$ on the training data
$$\{(x_1, y_1), \ldots, (x_N, y_N)\} \tag{5}$$
Set $T \leftarrow T^{(0)}$. Compute the residuals $\{r_1, \ldots, r_N\}$ for T.
2. Fit a simple model, $T^{(1)}$, to the current residuals, i.e. train using
$$\{(x_1, r_1), \ldots, (x_N, r_N)\} \tag{6}$$
3. set $T \leftarrow T + \lambda T^{(1)}$
4. Compute residuals, set $r_n \leftarrow r_n - \lambda T^i(x_n), n = 1, \ldots, N$
5. Repeat steps 2-4 until stopping condition met. where λ is a constant called the learning rate.

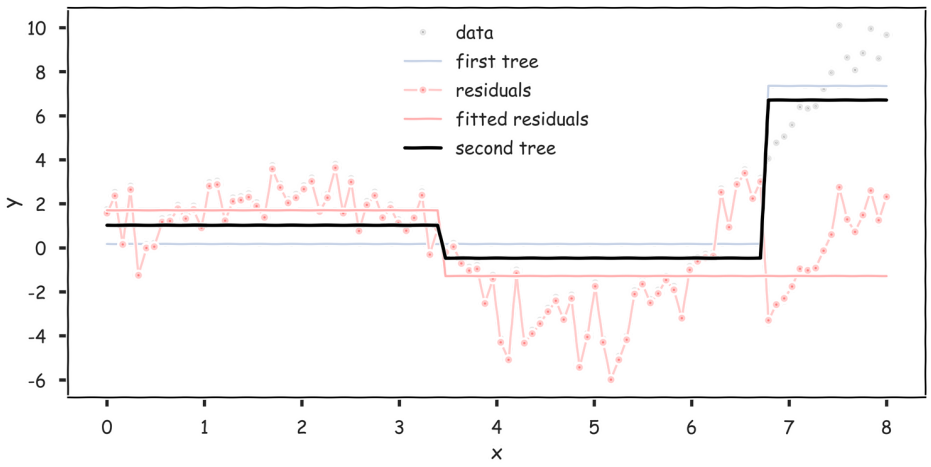
```
1 ![](data.png)
2 ![](f1.png)
3 ![](f2.png)
4 ![](f3.png)
5 ![](f4.png)
6 ![](f5.png)
7 ![](f6.png)
```

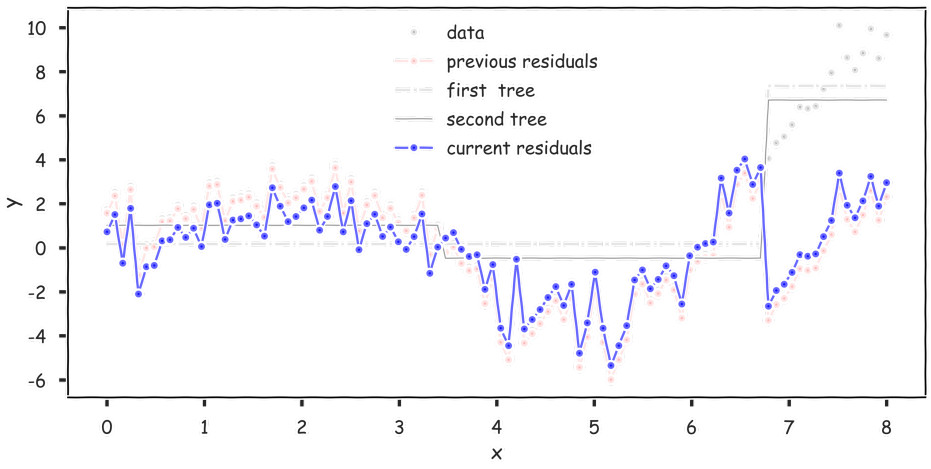

data
y
x

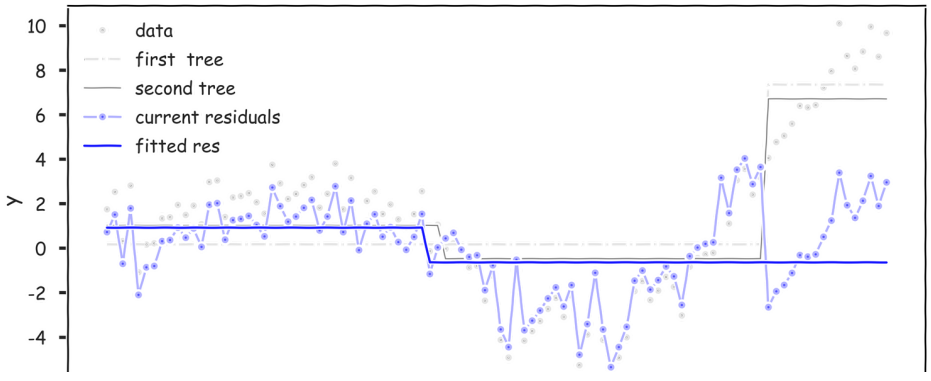

data
first tree
y
x

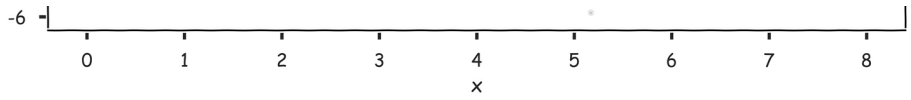

data
first tree
residuals
fitted residuals
y

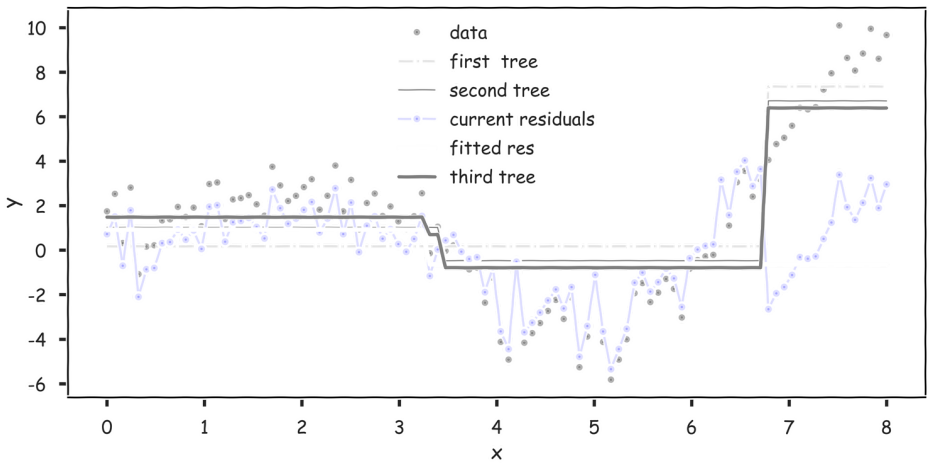

-6
0
1
2
3
4
5
6
7
8
x


data
first tree
residuals
fitted residuals
second tree
10
8
6
4
2
0
-2
-4
-6
y
0
1
2
3
4
5
6
7
8
x


data
previous residuals
first tree
second tree
current residuals
10
8
6
4
2
0
-2
-4
-6
y
0
1
2
3
4
5
6
7
8
x


data
first tree
second tree
current residuals
fitted res
10
8
6
4
2
0
-2
-4
y





5.7 Why Does Gradient Boosting Work?

Intuitively, each simple model $T^{(t)}$ we add to our ensemble model T, models the errors of T.

Thus, with each addition of $T^{(1)}$, the residual is reduced

$$r_n - \lambda T^{(i)}(x_n) \tag{7}$$

Note that gradient boosting has a tuning parameter, λ.

If we want to easily reason about how to choose λ and investigate the effect of λ on the model T, we need a bit more mathematical formalism. In particular, how can we effectively descend through this optimization via an iterative algorithm?

5.7.1 Gradient Descent

Minimize a function by moving in the opposite direction of the gradient.

$$\theta_i := \theta_i - \rho \frac{\partial J}{\partial \theta_i} \tag{8}$$

- How boosting is related to gradient descent?
- Loss function $L(y, F(x)) = (y - F(x))^2/2$
- We want to minimize $J = \sum_i L(y_i, F(x_i))$ by adjusting $F(x_1), F(x_2), \ldots, F(x_n)$.
- Notice that $F(x_1), F(x_2), \ldots, F(x_n)$ are just some numbers

- We can treat $F(x_i)$ as parameters and take derivatives
$$\frac{\partial J}{\partial F(x_i)} = \frac{\partial \sum_i L(y_i, F(x_i))}{\partial F(x_i)} = \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} = F(x_i) - y_i \tag{9}$$
So we can interpret residuals as negative gradients.
$$y_i - F(x_i) = -\frac{\partial J}{\partial F(x_i)} \tag{10}$$
- How is this related to gradient descent?
$$\begin{aligned} F(x_i) &:= F(x_i) + \mathrm{h}(x_i) \\ F(x_i) &:= F(x_i) + y_i - \mathrm{F}(x_i) \\ F(x_i) &:= F(x_i) - 1\frac{\partial J}{\partial \mathrm{F}(x_i)} \\ \theta_i &:= \theta_i - \rho\frac{\partial J}{\partial \theta_i} \end{aligned} \tag{11}$$

How is this related to gradient descent?

For regression with **square loss**,

- residual ⇔ negative gradient
- fit h to residual ⇔ fit h to negative gradient
- update F based on residual ⇔ update F based on negative gradient

So we are actually updating our model using **gradient descent**!

It turns out that the concept of gradients is more general and useful than the concept of residuals. So from now on, let's stick with gradients. The reason will be explained later.

5.8 Gradient Boosting as Gradient Descent

Often in regression, our objective is to minimize the MSE
$$\mathrm{MSE}\left(\hat{y}_1, \ldots, \hat{y}_N\right) = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - \hat{y}_i\right)^2 \tag{12}$$
Treating this as an optimization problem, we can try to directly minimize the MSE with respect to the predictions
$$\begin{aligned} \nabla \mathrm{MSE} &= \left[\frac{\partial \mathrm{MSE}}{\partial \hat{y}_1}, \ldots, \frac{\partial \mathrm{MSE}}{\partial \hat{y}_N}\right] \\ &= -2\left[y_1 - \hat{y}_1, \ldots, y_N - \hat{y}_N\right] \\ &= -2\left[r_1, \ldots, r_N\right] \end{aligned} \tag{13}$$
The update step for gradient descent would look like
$$\hat{y}_n \leftarrow \hat{y}_n + \lambda r_n, n = 1, \ldots, N \tag{14}$$

The solution is to change the update step in gradient descent. Instead of using the gradient - the residuals - we use an approximation of the gradient that depends on the predictors:
$$\hat{y} \leftarrow \hat{y}_n + \lambda \hat{r}_n(x_n), n = 1, \ldots, N \tag{15}$$
In gradient boosting, we use a simple model to approximate the residuals, $\hat{r}_n(x_n)$, in each iteration.

Motto: gradient boosting is a form of gradient descent with the MSE as the objective function.

Technical note: note that gradient boosting is descending in a space of models or functions relating x_n to y_n !

Regression with square Loss

Let us summarize the algorithm we just derived using the concept of gradients. Negative gradient:

$$-g(x_i) = -\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} = y_i - F(x_i) \tag{16}$$

start with an initial model, say, $F(x) = \frac{\sum_{i=1}^{n} y_i}{n}$ iterate until converge:

- calculate negative gradients $-g(x_i)$
- fit a regression tree h to negative gradients $-g(x_i)$
- $F := F + \rho h$, where $\rho = 1$

The benefit of formulating this algorithm using gradients is that it allows us to consider other loss functions and derive the corresponding algorithms in the same way.

5.8.1 The algorithm

5.8.1.1 Inspiration

The algorithm is inspired by the gradient descent algorithm. We consider a real function $f(x)$ and we calculate the gradient $\frac{\partial f}{\partial x}(x)$ to build a sequence:

$$x_{t+1} = x_t - \epsilon_t \frac{\partial f}{\partial x}(x_t) \tag{17}$$

The (x_t) sequence converges to the minimum of the f function. We apply this to an error function from a regression problem.

$$f(x) = \sum_{n=1}^{N} l(F(X_i), y_i) \tag{18}$$

Most often, we apply this method to a function F which depends on a parameter θ

$$f(\theta, x) = \sum_{n=1}^{N} l(F(\theta, X_i), y_i) \tag{19}$$

And it is the sequence $\theta_{t+1} = \theta_t - \epsilon_t \frac{\partial f}{\partial \theta}(\theta_t)$ which converges towards the minimum of the function f so that the function $f(\theta, x)$ approximates the points (X_i, y_i) at best. But, we could quite solve this problem in a space of functions and not a space of parameters:

$$G_{t+1} = G_t - \epsilon_t \frac{\partial f}{\partial G}(G_t) \tag{20}$$

The $\frac{\partial f}{\partial G}$ gradient is easy to calculate since it does not depend on G. We could therefore construct the regression function G as an additive sequence of functions $F_k \sim -\epsilon_t \frac{\partial f}{\partial G}(G_t)$.

$$G_t = \sum_{k=1}^{t} F_k \tag{21}$$

And we could construct the function F_k as a solution to a regression problem defined by the pairs (X_i, z_i) with:

$$\begin{aligned} z_i &= -\epsilon_t \frac{\partial f}{\partial G}(G_t(X_i), y_i) \\ f(X_i, y_i) &= l(G_t(X_i), y_i) \end{aligned} \tag{22}$$

That's the idea.

5.8.1.2 Algorithm

I resume here the wikipedia page. We are looking to build a model that minimizes the $L(y, F(x)) = \sum_{i=1}^{n} l(y_i, F(X_i))$ error. We note r the learning rate.

Step 1: we wedge a first regression model, here, simply a constant, by optimizing $F_0(x) = \arg\min_\gamma \sum_{i=1}^{n} L(y_i, \gamma)$. $F_0(x)$ is a constant.

We then write $F_m(x) = \gamma_0 \sum_{k=1}^{m} r\gamma_k h_k(x)$ where γ_0 is the constant function to be constructed during the first step.

Step 2: we then calculate the $e_{im} = l(y_i, F_m(x_i))$ errors and the opposite of the $r_{im} = -\left[\frac{\partial l(y_i, F_m(x_i))}{\partial F_m(x_i)}\right]$ gradient

Step 3: we choose the function $h_{m+1}(x)$ so that it approximates the residues r_{im} as well as possible.

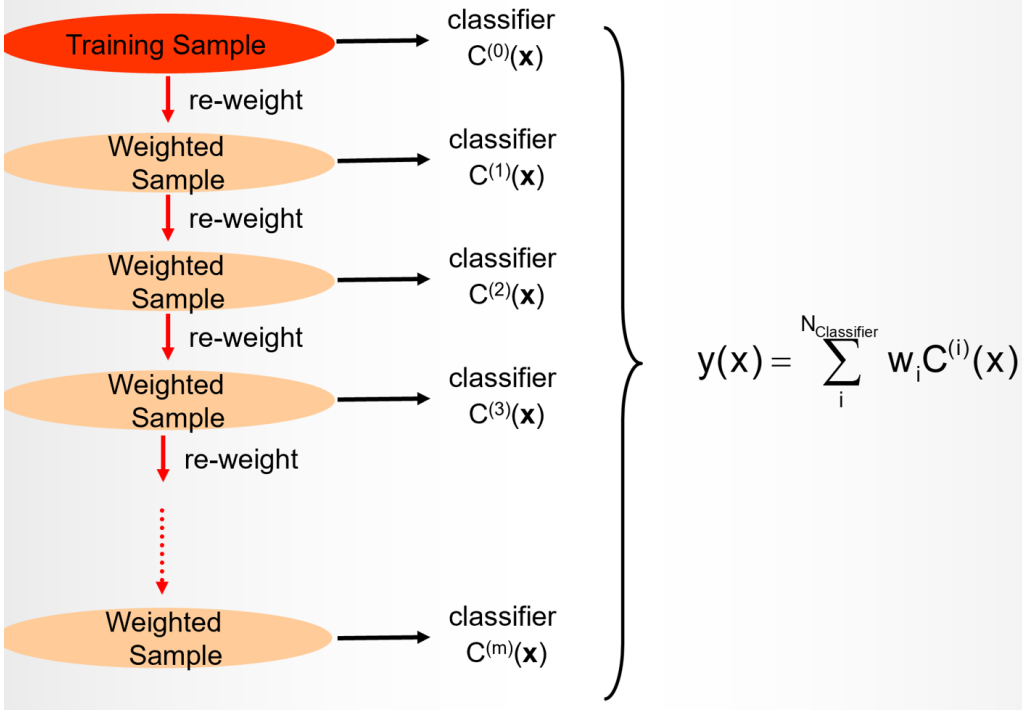
Step 4: we choose the coefficient γ_{m+1} so that it minimizes the expression $\min_\gamma \sum_{i=1}^{n} l\left(y_i, \gamma_0 + \sum_{k=1}^{m} r\gamma_k h_k(x_i) + \gamma h_{m+1}(x_i)\right)$.

We return step 2 as many times as there are iterations. When the error is a quadratic error $l(y, F(x)) = (y - F(x))^2$, the residuals become $r_{im} = -2(y_i - F_m(x_i))$. Consequently, the h function approximates at best what is missing to reach the objective. A learning rate equal to 1 means that the sum of the predictions of each h_m function oscillates around the true value, a low value gives the impression of a function that converges in small steps, a large value increases the amplitude of the oscillations at point to prevent the algorithm from converging.

We also see that the algorithm is first interested in the points where the gradient is the strongest, therefore in principle the largest errors.

```
1 ![](boosting.png)
2 ![](adaboosting.pnag)
3 ![](adaboosting_picture.png)
```

Boosting

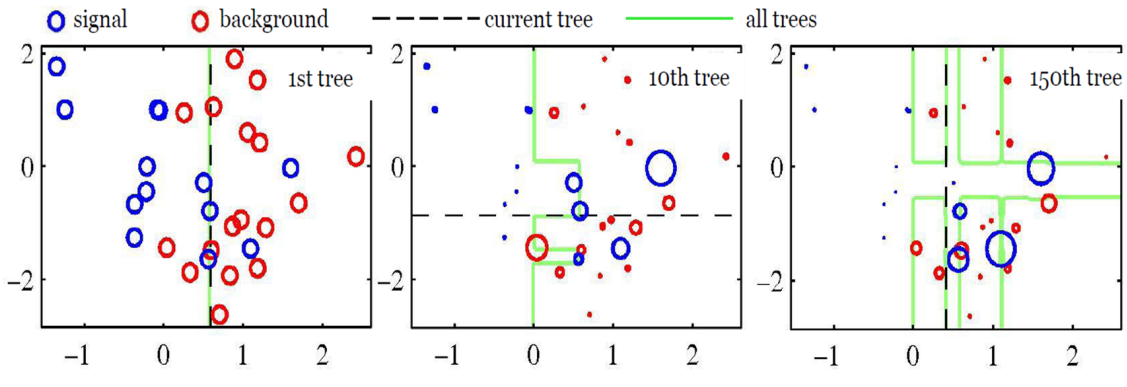


AdaBoost in Pictures

Start here:
equal event weights

misclassified events get
larger weights

… and so on



5.9 XGBoosting (Extreme Gradient Boosting)!

- Ever since its introduction in 2014, XGBoost has been lauded as the holy grail of machine learning hackathons and competitions. From predicting ad click-through rates to classifying high energy physics events, XGBoost has proved its mettle in terms of performance – and speed.
- Execution Speed: Generally, XGBoost is fast. Really fast when compared to other implementations of gradient boosting. But newly introduced LightGBM is faster than XGBoosting.
- Model Performance: XGBoost dominates structured or tabular datasets on classification and regression predictive modeling problems. The evidence is that it is the go-to algorithm for competition winners on the Kaggle competitive data science platform.

Remark 1 *Read the book I have upload to piazza with code.*

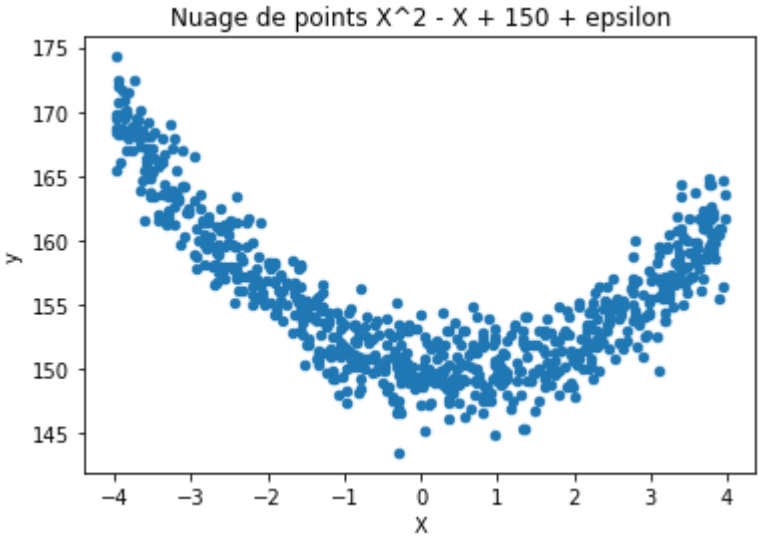
5.9.1 First example

We consider the default settings of the [GradientBoostingRegressor] class (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html#sklearn.ens (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html#sklearn.ens

In [1]:
```
1 %matplotlib inline
```

In [2]:

```
from numpy.random import randn, random
from pandas import DataFrame
from sklearn.model_selection import train_test_split
rnd = randn(1000)
X = random(1000) * 8 - 4
y = X ** 2 - X + rnd * 2 + 150 # X^2 - X + 150 + epsilon
X = X.reshape((-1, 1))
X_train, X_test, y_train, y_test = train_test_split(X, y)
df = DataFrame({'X': X_train.ravel(), 'y': y_train})
ax = df.plot(x='X', y='y', kind='scatter')
ax.set_title("Nuage de points X^2 - X + 150 + epsilon");
```



In [3]:

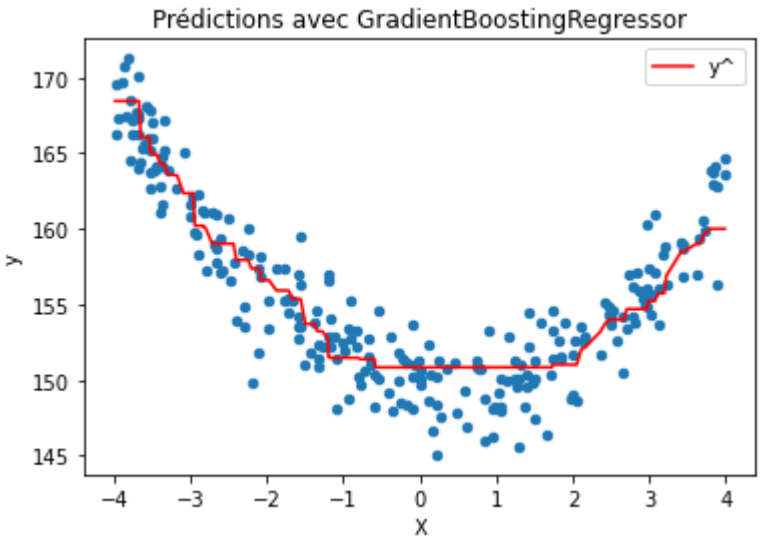
```
from sklearn.ensemble import GradientBoostingRegressor
model = GradientBoostingRegressor(max_depth=1)
model.fit(X_train, y_train)
```

Out[3]:

```
GradientBoostingRegressor(max_depth=1)
```

In [4]:

```
import numpy
ind = numpy.argsort(X_test, axis=0)
y_ = model.predict(X_test)
df = DataFrame({'X': X_test[ind].ravel(),
                'y': y_test[ind].ravel(),
                'y^': y_[ind].ravel()})
ax = df.plot(x='X', y='y', kind='scatter')
df.plot(x='X', y='y^', kind='line', ax=ax, color="r")
ax.set_title("Prédictions avec GradientBoostingRegressor");
```



Nothing unexpected so far. Let's try something else. We look with a single iteration.

In [5]:

```
model = GradientBoostingRegressor(max_depth=1, n_estimators=1, learning_
model.fit(X_train, y_train)
```

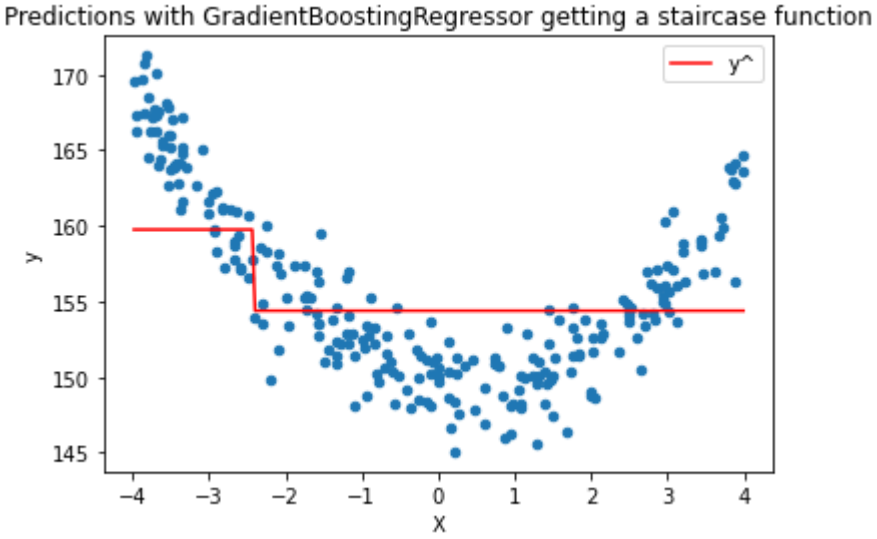
Out[5]:

```
GradientBoostingRegressor(learning_rate=0.5, max_depth=1, n_estimators=1)
```

model = GradientBoostingRegressor (max *depth = 1, n* estimators = 1, learning *rate = 0.5)*
model.fit (X train, y_train)

In [6]:

```
y_ = model.predict(X_test)
df = DataFrame({'X': X_test[ind].ravel(),
                'y': y_test[ind].ravel(),
                'y^': y_[ind].ravel()})
ax = df.plot(x='X', y='y', kind='scatter')
df.plot(x='X', y='y^', kind='line', ax=ax, color="r")
ax.set_title("Predictions with GradientBoostingRegressor getting a stair
```



Let us try to show the evolution of the predicted curve as a function of the number of steps and return to 100 estimators.

In [7]:

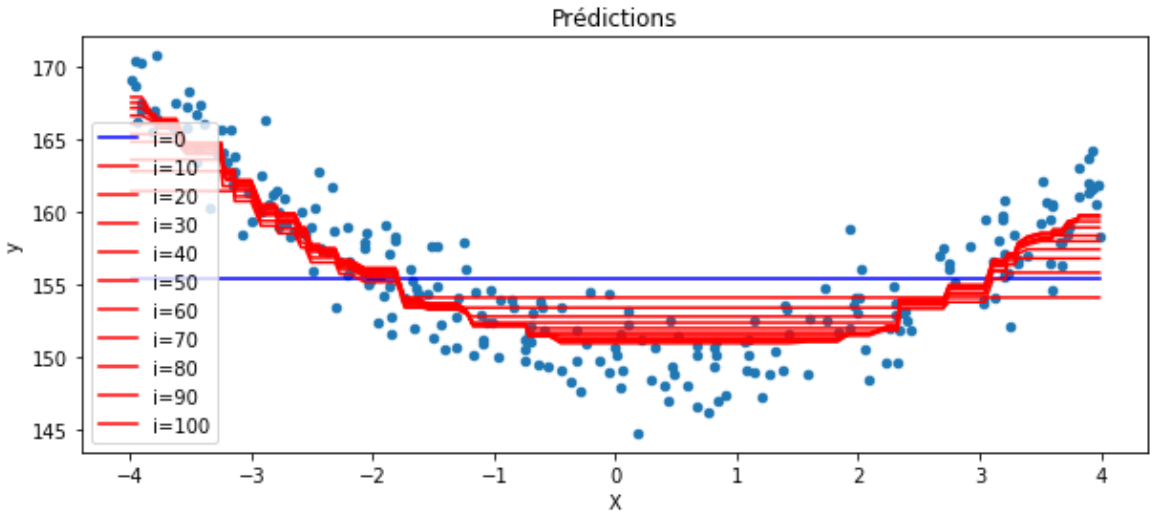
```
model = GradientBoostingRegressor(max_depth=1)
model.fit(X_train, y_train)
```

Out[7]:

```
GradientBoostingRegressor(max_depth=1)
```

In [13]:

```
for i in range(0, model.estimators_.shape[0] + 1, 10):
    if i == 0:
        df = DataFrame({'X': X_test[ind].ravel(),
                        'y': y_test[ind].ravel()})
        ax = df.plot(x='X', y='y', kind='scatter', figsize=(10, 4))
        y_ = model.init_.predict(X_test)
        color = 'b'
    else:
        y_ = sum([model.init_.predict(X_test)] +
                 [model.estimators_[k, 0].predict(X_test) * model.learni
                  for k in range(0, i)])
        color = 'r'
    df = DataFrame({'X': X_test[ind].ravel(),
                    'y^': y_[ind].ravel()})
    df.plot(x='X', y='y^', kind='line', ax=ax, color=color, label='i=%d
ax.set_title("Prédictions");
```



5.9.2 Learning rate and iterations

And if we chose a *learning_rate*, smaller or larger ...

In [8]:

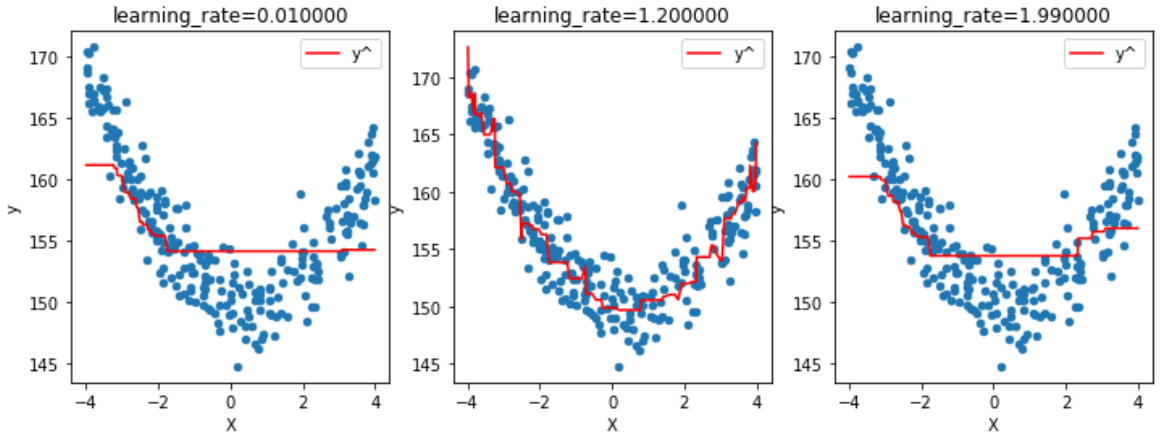
```
model01 = GradientBoostingRegressor(max_depth=1, learning_rate=0.01)
model01.fit(X_train, y_train)
modela = GradientBoostingRegressor(max_depth=1, learning_rate=1.2)
modela.fit(X_train, y_train)
modelb = GradientBoostingRegressor(max_depth=1, learning_rate=1.99)
modelb.fit(X_train, y_train)
```

Out[8]:

```
GradientBoostingRegressor(learning_rate=1.99, max_depth=1)
```

In [15]:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(1, 3, figsize=(12, 4))
ind = numpy.argsort(X_test, axis=0)

for i, mod in enumerate([model01, modela, modelb]):
    df = DataFrame({'X': X_test[ind].ravel(),
                    'y': y_test[ind].ravel(),
                    'y^': mod.predict(X_test)[ind].ravel()})
    df.plot(x='X', y='y', kind='scatter', ax=ax[i])
    df.plot(x='X', y='y^', kind='line', ax=ax[i], color="r")
    ax[i].set_title("learning_rate=%f" % mod.learning_rate);
```



Too low a value of *learning_rate* seems to retain the model of converging, a large value produces unpredictable effects. To understand why, we need to understand the details the algorithm.

5.9.3 Quantile regression

Ordinary least square regression is one of the most widely used statistical methods. However, it is a parametric model and relies on assumptions that are often not met. Quantile regression makes no assumptions about the distribution of the residuals. It also lets you explore different aspects of the relationship between the dependent variable and the independent variables.

In this case, the quadratic error is replaced by an error in absolute value. The residuals in this case are equal to -1 or 1.

In OLS, we minimize the sum of squared errors:

$$\sum u_i^2 = \sum (Y_i - B_1 - B_2 X)^2 \tag{23}$$

In QRM, we minimize the absolute sum of errors:

$$\sum |u_i| = \sum |Y_i - B_1 - B_2 X_i| \tag{24}$$

In [16]:

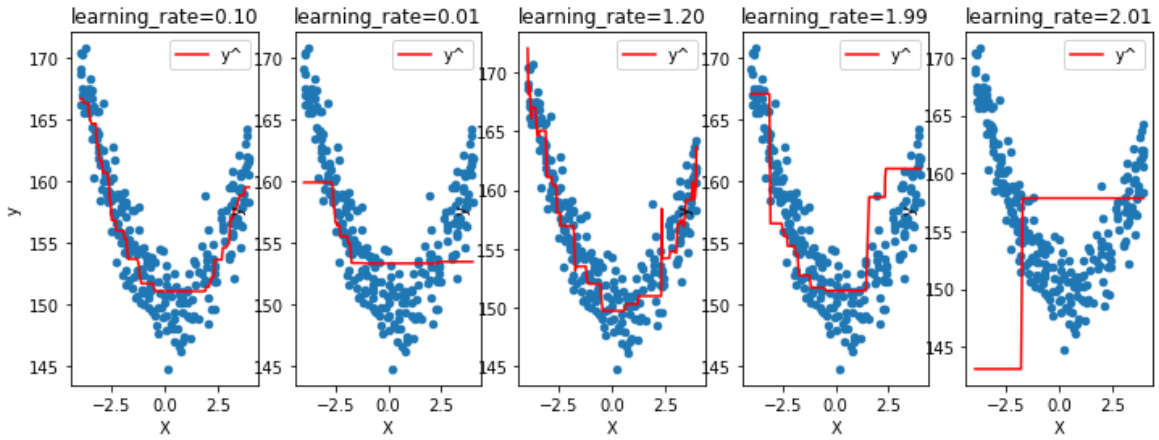
```
alpha = 0.5
model = GradientBoostingRegressor(alpha=alpha, loss='quantile', max_dept
model.fit(X_train, y_train)
model01 = GradientBoostingRegressor(alpha=alpha, loss='quantile', max_de
model01.fit(X_train, y_train)
modela = GradientBoostingRegressor(alpha=alpha, loss='quantile', max_dep
modela.fit(X_train, y_train)
modelb = GradientBoostingRegressor(alpha=alpha, loss='quantile', max_dep
modelb.fit(X_train, y_train)
modelc = GradientBoostingRegressor(alpha=alpha, loss='quantile', max_dep
modelc.fit(X_train, y_train)
```

Out[16]:

```
GradientBoostingRegressor(alpha=0.5, criterion='friedman_mse', init=None,
                          learning_rate=2.01, loss='quantile', max_depth=1,
                          max_features=None, max_leaf_nodes=None,
                          min_impurity_decrease=0.0, min_impurity_split=Non
e,
                          min_samples_leaf=1, min_samples_split=2,
                          min_weight_fraction_leaf=0.0, n_estimators=100,
                          n_iter_no_change=None, presort='auto',
                          random_state=None, subsample=1.0, tol=0.0001,
                          validation_fraction=0.1, verbose=0, warm_start=Fa
lse)
```

In [17]:

```
fig, ax = plt.subplots(1, 5, figsize=(12, 4))
ind = numpy.argsort(X_test, axis=0)

for i, mod in enumerate([model, model01, modela, modelb, modelc]):
    df = DataFrame({'X': X_test[ind].ravel(),
                    'y': y_test[ind].ravel(),
                    'y^': mod.predict(X_test)[ind].ravel()})
    df.plot(x='X', y='y', kind='scatter', ax=ax[i])
    df.plot(x='X', y='y^', kind='line', ax=ax[i], color="r")
    ax[i].set_title("learning_rate=%1.2f" % mod.learning_rate);
```



Concretely, the parameter *max_depth = 1* corresponds to a simple $f(x) = \mathbb{1}_{x>s}$ function and the final model is a weighted sum of indicator functions.

5.9.4 Learning rate and overfitting - comparison of GradientBoostingRegressor with RandomForestRegressor

In [18]:

```
from sklearn.ensemble import RandomForestRegressor
from tqdm import tqdm

def experiment(models, tries=25):
    scores = []
    for _ in tqdm(range(tries)):
        rnd = randn(1000)
        X = random(1000) * 8 - 4
        y = X ** 2 - X + rnd * 2 + 150 # X^2 - X + 150 + epsilon
        X = X.reshape((-1, 1))
        X_train, X_test, y_train, y_test = train_test_split(X, y)
        scs = []
        for model in models:
            model.fit(X_train, y_train)
            sc = model.score(X_test, y_test)
            scs.append(sc)
        scores.append(scs)
    return scores

scores = experiment([
    GradientBoostingRegressor(max_depth=1, n_estimators=100),
    RandomForestRegressor(max_depth=1, n_estimators=100)
])
scores[:3]
```

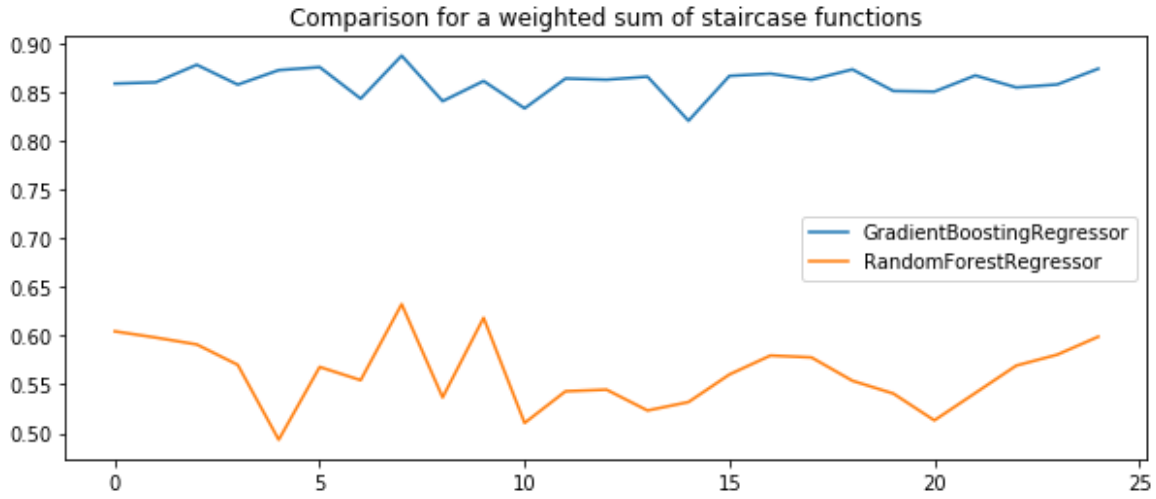
```
100%|██████████████████████████████████████| 25/25 [00:01<00:00, 12.82it/s]
```

Out[18]:

```
[[0.8590582491261854, 0.6046029627822649],
 [0.8603406002102773, 0.5982241653720843],
 [0.8783298318411722, 0.5911475320625542]]
```

In [19]:

```
fig, ax = plt.subplots(1, 1, figsize=(10, 4))
ax.plot([_[0] for _ in scores], label="GradientBoostingRegressor")
ax.plot([_[1] for _ in scores], label="RandomForestRegressor")
ax.set_title("Comparison for a weighted sum of staircase functions")
ax.legend();
```



This result is expected because the random forest is an average of a regression model all learned under the same conditions while the boosting gradient is interested in the error after the sum of the first regressors. Let us see with decision trees and no longer staircase functions.

In [20]:

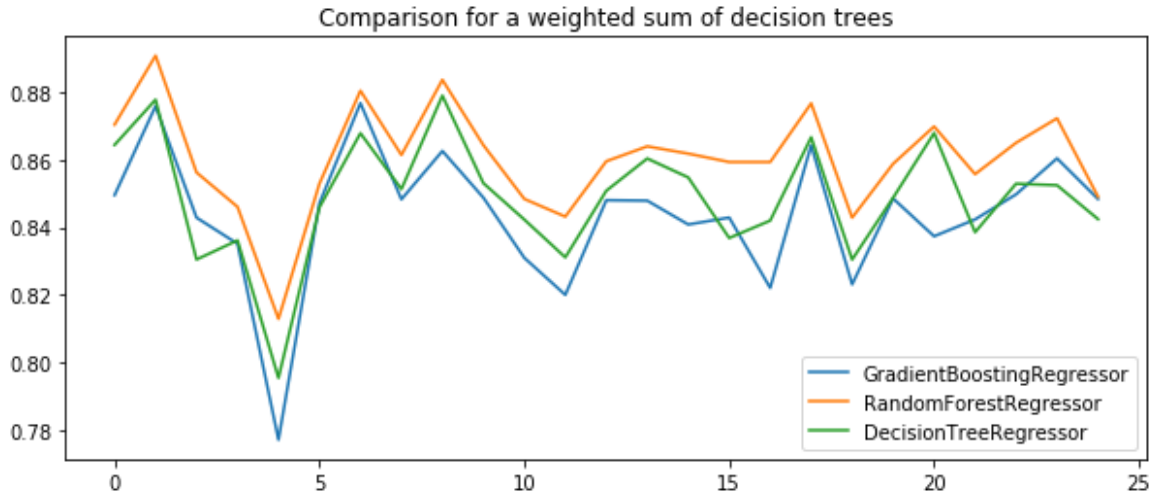
```
from sklearn.tree import DecisionTreeRegressor

scores = experiment([
    GradientBoostingRegressor(max_depth=5, n_estimators=100),
    RandomForestRegressor(max_depth=5, n_estimators=100),
    DecisionTreeRegressor(max_depth=5)
])
```

```
100%|██████████████████████████████████████████| 25/25 [00:02<00:00,  9.11it/s]
```

Comparison for a weighted sum of decision trees

In [21]:

```
fig, ax = plt.subplots(1, 1, figsize=(10, 4))
ax.plot([_[0] for _ in scores], label="GradientBoostingRegressor")
ax.plot([_[1] for _ in scores], label="RandomForestRegressor")
ax.plot([_[2] for _ in scores], label="DecisionTreeRegressor")
ax.set_title("Comparison for a weighted sum of decision trees")
ax.legend();
```



The *GradientBoostingRegressor* model is clearly worse when the underlying model - the decision tree - performs well. We see that the random forest is better than a decision tree alone. This means that it generalizes better and that the decision tree is over learning. Likewise, the *GradientBoostingRegressor* is more exposed to over-learning.

In [22]:

```
scores = experiment([
    RandomForestRegressor(max_depth=5, n_estimators=100),
    GradientBoostingRegressor(max_depth=5, n_estimators=100, learning_ra
    GradientBoostingRegressor(max_depth=5, n_estimators=100, learning_ra
    GradientBoostingRegressor(max_depth=5, n_estimators=100, learning_ra
])
scores[:2]
```

```
100%|██████████████████████████████████████████| 25/25 [00:04<00:00,  5.85it/s]
```

Out[22]:

```
[[0.8458291746325406,
  0.8297658281123015,
  0.8245549900356747,
  0.8037276474212294],
 [0.8471845382178536,
  0.8396317372192924,
  0.8247642763106952,
  0.8067100616077794]]
```

In [23]:

```
fig, ax = plt.subplots(1, 1, figsize=(10, 4))
ax.plot([_[0] for _ in scores], label="RandomForestRegressor")
ax.plot([_[1] for _ in scores], label="GBR(5, lr=0.05)")
ax.plot([_[2] for _ in scores], label="GBR(5, lr=0.1)")
ax.plot([_[3] for _ in scores], label="GBR(5, lr=0.2)")
ax.set_title("Comparison for different learning_rate")
ax.legend();
```

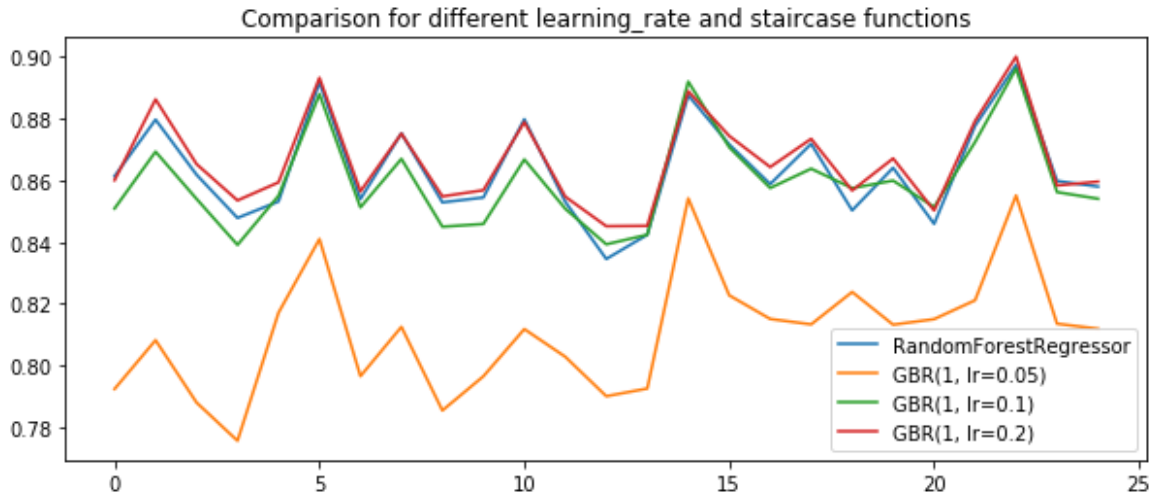
Decreasing *learning_rate* is clearly a way to avoid over-learning, but the previous graphs have shown that more iterations are needed when the learning rate is small.

In [24]:
```
scores = experiment([
    RandomForestRegressor(max_depth=5, n_estimators=100),
    GradientBoostingRegressor(max_depth=1, n_estimators=100, learning_ra
    GradientBoostingRegressor(max_depth=1, n_estimators=100, learning_ra
    GradientBoostingRegressor(max_depth=1, n_estimators=100, learning_ra
])
scores[:2]
```

```
100%|██████████████████████████████████████████| 25/25 [00:03<00:00,  6.94i
t/s]
```

Out[24]:
```
[[0.8613504364851733,
  0.7924670577838948,
  0.8509048926900717,
  0.8600608204862137],
 [0.8796775001636237,
  0.8082652938319602,
  0.8693017679582076,
  0.886249765719035]]
```

In [25]:
```
fig, ax = plt.subplots(1, 1, figsize=(10, 4))
ax.plot([_[0] for _ in scores], label="RandomForestRegressor")
ax.plot([_[1] for _ in scores], label="GBR(1, lr=0.05)")
ax.plot([_[2] for _ in scores], label="GBR(1, lr=0.1)")
ax.plot([_[3] for _ in scores], label="GBR(1, lr=0.2)")
ax.set_title("Comparison for different learning_rate and staircase funct
ax.legend();
```



The simpler the underlying model, the higher the *learning_rate* because the simple models do not over-learn.

5.9.5 Gradient Boosting with other Libraries

A weighted sum of linear regression remains a linear regression. It is impossible to test this scenario with *scikit-learn* since only decision trees are implemented. But there are other libraries that implement gradient boosting.

5.9.5.1 XGBoost

https://xgboost.readthedocs.io/en/latest/build.html#
(https://xgboost.readthedocs.io/en/latest/build.html)

Learning Task Parameters

Specify the learning task and the corresponding learning objective. The objective options are below:

- objective [default=reg:squarederror]
  - reg:squarederror: regression with squared loss.
  - reg:squaredlogerror: regression with squared log loss $1/2[log(pred + 1) - log(label + 1)]^2$. All input labels are required to be greater than -1. Also, see metric rmsle for possible issue with this objective.
  - reg:logistic: logistic regression
  - binary:logistic: logistic regression for binary classification, output probability
  - binary:logitraw: logistic regression for binary classification, output score before logistic transformation
  - binary:hinge: hinge loss for binary classification. This makes predictions of 0 or 1, rather than producing probabilities.
  - count:poisson –poisson regression for count data, output mean of poisson distribution
    - max_delta_step is set to 0.7 by default in poisson regression (used to safeguard optimization)
  - survival:cox: Cox regression for right censored survival time data (negative values are considered right censored). Note that predictions are returned on the hazard ratio scale (i.e., as HR = exp(marginal_prediction) in the proportional hazard function h(t) = h0(t) * HR).
  - multi:softmax: set XGBoost to do multiclass classification using the softmax objective, you also need to set num_class(number of classes)
  - multi:softprob: same as softmax, but output a vector of ndata * nclass, which can be further reshaped to ndata * nclass matrix. The result contains predicted probability of each data point belonging to each class.
  - rank:pairwise: Use LambdaMART to perform pairwise ranking where the pairwise loss is minimized
  - rank:ndcg: Use LambdaMART to perform list-wise ranking where Normalized Discounted Cumulative Gain (NDCG) is maximized
  - rank:map: Use LambdaMART to perform list-wise ranking where Mean Average Precision (MAP) is maximized
  - reg:gamma: gamma regression with log-link. Output is a mean of gamma distribution. It might be useful, e.g., for modeling insurance claims severity, or for any outcome that might be gamma-distributed.
  - reg:tweedie: Tweedie regression with log-link. It might be useful, e.g., for modeling total loss in insurance, or for any outcome that might be Tweedie-distributed. ```

5.9.5.2 Example of use xgboost

In [26]:
```
import xgboost as xgb
# read in data
dtrain = xgb.DMatrix('demo/data/agaricus.txt.train')
dtest = xgb.DMatrix('demo/data/agaricus.txt.test')
# specify parameters via map
param = {'max_depth':2, 'eta':1, 'objective':'binary:logistic' }
num_round = 2
bst = xgb.train(param, dtrain, num_round)
# make prediction
preds = bst.predict(dtest)
preds
```

```
[18:31:35] 6513x127 matrix with 143286 entries loaded from demo/data/agaric
us.txt.train
[18:31:36] 1611x127 matrix with 35442 entries loaded from demo/data/agaricu
s.txt.test
```

Out[26]:
```
array([0.28583017, 0.9239239 , 0.28583017, ..., 0.9239239 , 0.05169873,
       0.9239239 ], dtype=float32)
```

In [43]:
```
from xgboost import XGBRegressor
```

In [71]:
```
scores = experiment([
    RandomForestRegressor(max_depth=5, n_estimators=100),
    XGBRegressor(max_depth=1, n_estimators=100, learning_rate=0.05),
    XGBRegressor(max_depth=1, n_estimators=100, learning_rate=0.1),
    XGBRegressor(max_depth=1, n_estimators=100, learning_rate=0.2),
])
scores[:2]
```

```
100%|██████████████████████████████████████████| 25/25 [00:05<00:00,  4.23i
t/s]
```

Out[71]:
```
[[0.851890179048378,
  0.7427874268081183,
  0.8377267784582689,
  0.8506930345711928],
 [0.8967819950099597,
  0.8138593313967923,
  0.8888928443893042,
  0.8969635816569166]]
```

"Comparison for different learning_rate \ net of staircase functions" "with XGBoost"

In [72]:

```
fig, ax = plt.subplots(1, 1, figsize=(10, 4))
ax.plot([_[0] for _ in scores], label="RandomForestRegressor(5)")
ax.plot([_[1] for _ in scores], label="XGB(1, lr=0.05)")
ax.plot([_[2] for _ in scores], label="XGB(1, lr=0.1)")
ax.plot([_[3] for _ in scores], label="XGB(1, lr=0.2)")
ax.set_title("Comparison for different learning_rate \ net of staircase
ax.legend();
```

The results are much the same.

In [31]:
```
fig, ax = plt.subplots(1, 1, figsize=(10, 4))
ax.plot([_[0] for _ in scores], label="RandomForestRegressor(5)")
ax.plot([_[1] for _ in scores], label="XGB(5, lr=0.05)")
ax.plot([_[2] for _ in scores], label="XGB(5, lr=0.1)")
ax.plot([_[3] for _ in scores], label="XGB(5, lr=0.2)")on pour différent
             "avec XGBoost")
ax.legend();
```

5.9.5.3 LightGBM

In [45]:
```
from lightgbm import LGBMRegressor
```

In [46]:
```
scores = experiment([
    RandomForestRegressor(max_depth=5, n_estimators=100),
    LGBMRegressor(max_depth=1, n_estimators=100, learning_rate=0.05),
    LGBMRegressor(max_depth=1, n_estimators=100, learning_rate=0.1),
    LGBMRegressor(max_depth=1, n_estimators=100, learning_rate=0.2),
])
scores[:2]
```

```
100%|██████████████████████████████████████████| 25/25 [00:02<00:00,  9.84i
t/s]
```

Out[46]:
```
[[0.8617120857549567,
  0.8035740757873253,
  0.8571470291709791,
  0.8674293101683874],
 [0.8623447353615801,
  0.8037886426493026,
  0.857715580345726,
  0.8660086422339337]]
```

In [47]:

```
fig, ax = plt.subplots(1, 1, figsize=(10, 4))
ax.plot([_[0] for _ in scores], label="RandomForestRegressor(5)")
ax.plot([_[1] for _ in scores], label="LGB(1, lr=0.05)")
ax.plot([_[2] for _ in scores], label="LGB(1, lr=0.1)")
ax.plot([_[3] for _ in scores], label="LGB(1, lr=0.2)")
ax.set_title("Comparison for different learning_rate \ net of staircase
"with LightGBM")
ax.legend();
```

In [48]:

```
scores = experiment([
    RandomForestRegressor(max_depth=5, n_estimators=100),
    LGBMRegressor(max_depth=5, n_estimators=100, learning_rate=0.05),
    LGBMRegressor(max_depth=5, n_estimators=100, learning_rate=0.1),
    LGBMRegressor(max_depth=5, n_estimators=100, learning_rate=0.2),
])
scores[:2]
```

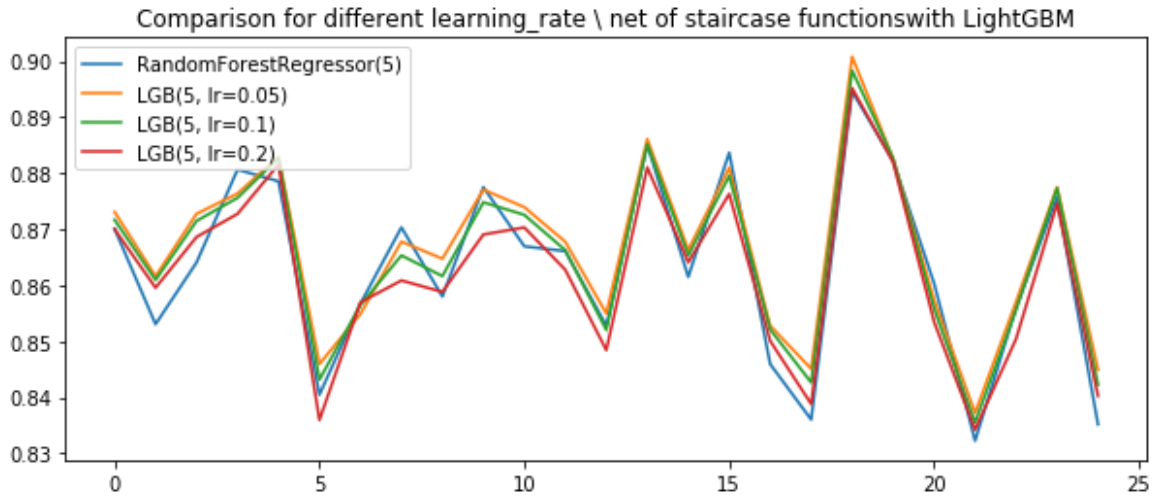
```
100%|██████████████████████████████████████| 25/25 [00:03<00:00,  7.21i
t/s]
```

Out[48]:

```
[[0.8700771290568682,
  0.8730668496398959,
  0.871666608003002,
  0.8699889507026795],
 [0.853030812928907,
  0.8615820684244622,
  0.860937708218124,
  0.8595363420288401]]
```

In [49]:

```
fig, ax = plt.subplots(1, 1, figsize=(10, 4))
ax.plot([_[0] for _ in scores], label="RandomForestRegressor(5)")
ax.plot([_[1] for _ in scores], label="LGB(5, lr=0.05)")
ax.plot([_[2] for _ in scores], label="LGB(5, lr=0.1)")
ax.plot([_[3] for _ in scores], label="LGB(5, lr=0.2)")
ax.set_title("Comparison for different learning_rate \ net of staircase
"with LightGBM")
ax.legend();
```



LightGBM (https://lightgbm.readthedocs.io/en/latest/) seems less sensitive to *learning_rate* than XGBoost (https://xgboost.readthedocs.io/en/latest/index.html).

5.9.5.4 CatBoost

CatBoost (https://catboost.ai/) is one of the most recent. It is supposed to be more effective for categories which is not the case here.

In [51]:

```
from catboost import CatBoostRegressor
```

In [53]:

```
scores = experiment([
    RandomForestRegressor(max_depth=5, n_estimators=100),
    CatBoostRegressor(max_depth=1, n_estimators=100, learning_rate=0.05,
    CatBoostRegressor(max_depth=1, n_estimators=100, learning_rate=0.1,
    CatBoostRegressor(max_depth=1, n_estimators=100, learning_rate=0.2,
])
scores[:2]
```

```
100%|██████████████████████████████████████████| 25/25 [00:08<00:00,  2.86it/s]
```

Out[53]:

```
[[0.8542087998876264,
  0.8229064790081374,
  0.8558364881359926,
  0.8568076892829759],
 [0.8625394382599387,
  0.8267904972293666,
  0.8662663740170987,
  0.867240240064011]]
```

In [54]:

```
fig, ax = plt.subplots(1, 1, figsize=(10, 4))
ax.plot([_[0] for _ in scores], label="RandomForestRegressor(5)")
ax.plot([_[1] for _ in scores], label="CAT(1, lr=0.05)")
ax.plot([_[2] for _ in scores], label="CAT(1, lr=0.1)")
ax.plot([_[3] for _ in scores], label="CAT(1, lr=0.2)")
ax.set_title("Comparison for different learning_rate \ net of staircase
"with CatBoost")
ax.legend();
```

In [57]:

```
scores = experiment([
    RandomForestRegressor(max_depth=5, n_estimators=100),
    CatBoostRegressor(max_depth=5, n_estimators=100, learning_rate=0.05,
    CatBoostRegressor(max_depth=5, n_estimators=100, learning_rate=0.1,
    CatBoostRegressor(max_depth=5, n_estimators=100, learning_rate=0.2,
])
scores[:2]
```

```
100%|██████████████████████████████████████████| 25/25 [00:12<00:00,  2.04it/s]
```

Out[57]:

```
[[0.8791293261694145,
  0.8890669411833441,
  0.8880987600255773,
  0.88556102167787],
 [0.8579866184445712,
  0.8614921507888764,
  0.8598602613685565,
  0.8551331531808335]]
```

In [58]:

```
fig, ax = plt.subplots(1, 1, figsize=(10, 4))
ax.plot([_[0] for _ in scores], label="RandomForestRegressor(5)")
ax.plot([_[1] for _ in scores], label="CAT(5, lr=0.05)")
ax.plot([_[2] for _ in scores], label="CAT(5, lr=0.1)")
ax.plot([_[3] for _ in scores], label="CAT(5, lr=0.2)")
ax.set_title("Comparison for different learning_rate \ net of staircase
"with CatBoost")
ax.legend();
```