NYC DATA SCIENCE ACADEMY

Python Machine Learning

Boosting and Gradient Boosting

NYC Data Science Academy

Outline

- Boosting-Through the Example of LSBoost
- Gradient boosting

Boosting

- Boosting is a very general sequential ensemble technique which aggregates many weak learners to produce a strong learner
- It differs from the parallel ensembling in that it produces a strong learner in a sequential way: Iteratively, the kth weak learner makes use of the previous k-1 weak learners' outcome to make its own educated guess
- Both parallel ensembling (like bagging, random forests) and boosting involve the team works of individual weak learners, but in drastically different ways:
  - Parallel ensembling is like a committee of weak learners
  - Boosting is like relay track race

Many Facets of Boosting

- There are a zoo of boosting algorithms in the machine learning literature, led by **adaboost**, …., **brownboost**, **logitboost**, …, **gradient boost**, **xgboost**, etc
- While the math of the general boosting algorithm can be daunting, we focus on motivating its key idea using **LSBoost** with linear regression
- This motivates us to consider a more general **gradient boost** framework, a special subclass of boosting algorithm, which contains **xgboost**, the recently introduced popular boosting algorithm, as a special case

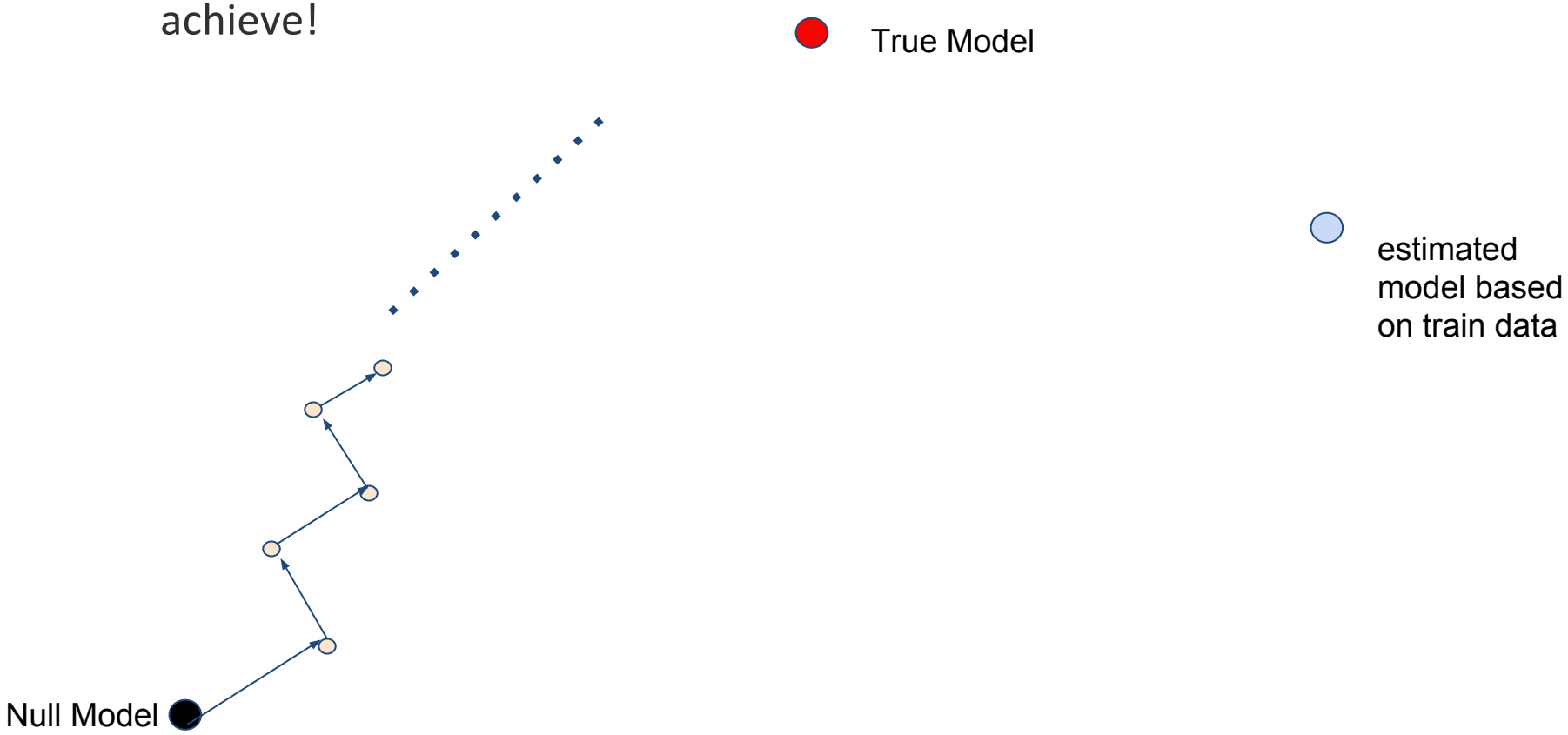
- The original **adaboost**, which starts the boosting methodology, is a binary classifier
- Even though boosting can be applied to different types of weak learners, the most popular choice uses trees as weak learners
- Gradient boosting, and its special case xgboost, can be formulated for both regression tasks and classification tasks

LSBoost--A Reflection on Linear Regression

- The general concept of **boosting** has been rooted in the theory of linear regression
- To make the link manifest, we introduce **LSBoost**, least square boost, as a simple illustration
- In the context of multiple linear regression, the **normal equation**, which is the closed form formula of the estimated linear model coefficients, is the best unbiased linear estimator based on the training set
- When the test sets are introduced, the linear model estimated by the normal equation can produce overfitting, due to high model variance
- Penalized Ridge/Lasso regression introduces biases in the coefficient estimation by reducing the model variance
- In other words, the true model generating the full data, might not be the one pinned down by the normal equation using a train set, but is located somewhere else in the space of linear coefficients!

A Step by Step Approach

- Instead of running the penalized regression, is there an iterative way to approach the true model step by step? This is what **LSBoost** tries to achieve!



One Feature at a Time

- The weak learners in this context are the linear models using only single features
- To simplify our discussion, we **normalize the features and the target**
- Starting from the null model with all features' coefficients being zero, we ask which weak learner (i.e. a single feature) is most promising to improve the null model
- Apparently it is the feature which has the highest correlation to the target!
- Another way to phrase it is to run multiple simple linear regressions simultaneously on individual features and pick the one with the largest (in absolute value) regression coefficient

The Danger of Overshooting and Learning Rate

- Naively the chosen feature scaled up by the regression coefficient is the best shot (among all features) of using a single feature to build a linear model
- Nevertheless, this can overshot the true model. To remedy, a small positive number ϵ is multiplied in front and $\epsilon \cdot \beta_j^{step=1} X_j$ is the best shot at the first stage (j is the index corresponding to the best feature)
- Such an ϵ is called the **step size/learning rate** in the literature
- With the first step finished, what should the second weak learner do?
- Instead of running simple linear regressions across all features on the original target y, $y - \epsilon \cdot \beta_j^{step=1} X_j$ should be used instead
- This is nothing but the residual after finishing the first step

The Inductive Step and Its Limit

- Inductively, suppose that we are done with the kth step and choose k (potentially duplicated) features as the weak learners:
- We end up with the k-th step residual $y - \sum_{i \leq k} \epsilon \cdot \beta_{j_i}^{step=i} X_{j_i}$, which we build the next one-feature simple linear regression model upon
- The partial sum of all the weak learners is a linear model which performs stronger than the individual constituents
- This procedure never stops, but it can be proved theoretically that the limiting linear model is the one estimated by the normal equation
- Hopefully when we early-stop at a finite k, the resulting linear model provides a better bias-variance trade-off than the naive multiple linear regression model

The Choice of the Step Size

- The choices of the learning-rate/step-size affect the speed to get to the desired minimal test error model in the search path
- If ϵ is too large, it has a quick convergence to the multiple linear regression model, which might have missed the best model in the path. This is known as **overshooting**, which we like to avoid
- On the other hand, a very small ϵ reduces the danger of **overshooting** the optimal location. But it increases the number of iterations needed to get there and may waste a lot of computation power
- When the boosting is applied on a nonlinear weak learner, a very small ϵ can lead to solutions trapped at a local minimum

What Do We Learn Through this Example?

- **LSBoost** is by no means a turbo-charger which boosts the weak learners into a high performance model
- It is noteworthy to point out that it demonstrates important characteristics resembling many other boosting algorithms:
  - The k+1th weak learner bases its learning on the result of all the previous k weak learners (true for all boosting algos)
  - In particular, the k+1th weak learner learns the patterns of the residual--the difference of the target from the partial sum of the previous k weak learners' predictions, scaled by a small learning rate (true for the **gradient boosting regressor** we will talk about next)
- The typical boosting algorithm involves an infinite number of basis functions, but in **LSBoost** there is only a finite number of possible features

Boosting and Gradient Boosting

- Boosting--Through the Example of LSBoost
- Gradient Boosting

Gradient Boosting

- **Gradient boosting** is a special framework of **boosting** algorithms which updates a sequence of weak learners into a strong learner
- The framework can be formulated for both regression and classification tasks
- The concept of **gradient boosting** is applicable to many types of weak learners, including linear regression, logistic regression, generalized linear models and the most popular decision tree based models
- sklearn's **gradient boosting** allows only tree based **gbm** models
- **xgboost**, the extreme **gradient boosting** machine, is a recent high performance expansion within the **gbm** family which adds a lot of whistles and bells for faster speed and improved accuracy

Key Properties of Gradient Boosting

- For regression problem, sklearn's gradient boosting regressor uses the quadratic **MSE** as its default loss function
- For classification problem, sklearn's gradient boosting classifier uses the **log loss/cross entropy loss** as its default loss function
- Like the other tree based models, it is able to handle multi-class classification directly without resorting to **one vs the rest** ensembling
- Like the above toy **LSBoost** example, the **gradient boosting** algorithm find a sequence of weak learners (often using decision trees) recursively
- Because it is designed to avoid overshooting in the iterative step, the trees used as the weak learners can be rather short. This enhances its speed performance dramatically compared to the computation speed of random forests

Key Hyperparameters of Gradient Boosting

- It is crucial to notice that **GradientBoostClassifier** does **NOT** make use of **DecisionTreeClassifier** as its building block
- Instead both gradient boosting regressor/classifiers formulate their tasks into different decision tree regression problems
- Besides the standard hyperparameters of a decision tree, two key tuning hyperparameters are:
  - n_estimators--the number of weak learners (trees) to use
  - learning_rate--the analogue of ϵ which discounts the result of the single tree in each iterative step
- As a consequence of bypassing **DecisionTreeClassifier**, gradient boosting trees do not use **gini** index nor **Information entropy** as its tree-splitting criterion (unlike **random forests**). Instead it uses **mse** (for regression) and the so-called **Friedman_mse** (for classification) by default

The HighLight of the Algorithm

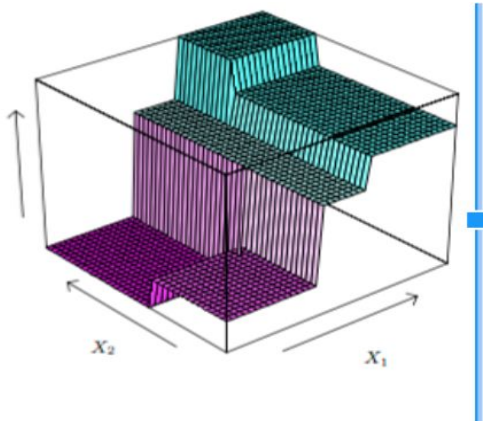
- There are a few questions we would like to address regarding **gradient boosting**:
  - How does the iterative tree building algorithm work?
  - How does it unify the regression and classification tasks?
  - Where does the term 'gradient' in **gradient boosting** come from?

The Iterative Procedure for Gradient Boosting Regression

- The recursive procedure is similar to the **LSBoost** regression algorithm we have presented
- Suppose that y is the target we would like to approximate with. Schematically we would like to approximate y as a sum of a sequence of trees in an additive way

$$y \sim \sum_{i \leq n\_estimators} \epsilon \cdot \phi_{T_i}$$

, where ϵ is the learning_rate, a small positive number, ϕ_{T_i} is the tree approximation function associated with the ith tree T_i



Iterative Procedure Continued

- At the beginning, we start with the trivial approximation 0. We train a short (in terms of tree depth) decision tree to approximate the target y (under **MSE/RSS**)
- The procedure is identical to fitting a **DecisionTreeRegressor**
- Instead of using the result directly, we discount it by a small learning_rate
- The philosophical idea is to let many trees work cooperatively than to let a single tree finish the job (why?)
- Suppose that the first k trees have been chosen to approximate the target y. We may use **MSE** to gauge the quality of its fit
- **MSE** is the quantitative measure of the residual between the true target y and our kth step approximation

$$y - \sum_{i \leq k} \epsilon \cdot \phi_{T_i}$$

The Termination

- The key insight of **gradient boosting regressor** is to use the k+1th tree to fit the kth residual $y - \sum_{i \leq k} \epsilon \cdot \phi_{T_i}$ rather than fit the original target y (like a **random forest** would do)
- We end up with a discounted fit, $\epsilon \cdot \phi_{T_{k+1}}$, from the k+1th tree T_{k+1}
- We emphasize that the discounted fit on the residual

$$y - \sum_{i \leq k} \epsilon \cdot \phi_{T_i} \sim \epsilon \cdot \phi_{T_{k+1}}$$

is essentially the same as fitting the original target y by all k+1 trees

$$y \sim \sum_{i \leq k} \epsilon \cdot \phi_{T_i} + \epsilon \cdot \phi_{T_{k+1}} = \sum_{i \leq k+1} \epsilon \phi_{T_i}$$

- **MSE/RSS** gauges the size of the k+1th residual
  - If it is small enough, we can stop early and terminate the iterative procedure
  - If it is still large, continue to the next step. Go on until k hits the upper bound n_estimators

Some Remark on the Boosting Regressor

- We notice that the above recursive step is parallel to the **LSBoost** presented earlier. The major difference here is to replace a single feature simple linear regression by a **decision tree regressor**
- Because different trees fitting on different residuals (compared to **random forest regressor** where all trees fit on the same target y), they tend to be less correlated
- **Random forest** is a horizontal ensemble technique where all member trees collectively work on the **SAME** task
- **Boosting** is a vertical ensemble technique where the original task is split into many individual subtasks and the individual trees work on the separated subtasks (divide and conquer!)

Introducing Parallelism to Boosting Trees

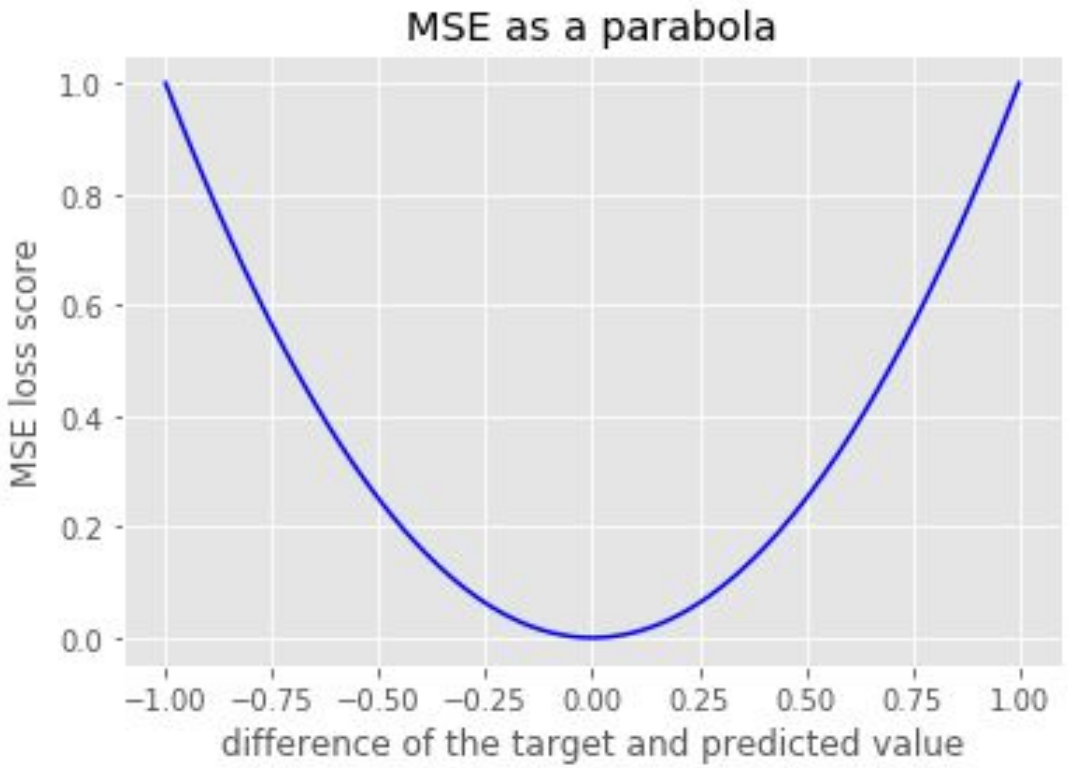
- Because all the trees within a **random forest** are built independently, it is rather straightforward to parallelize the task, i.e. running it on a distributed multiple CPUs platform
- We can train a batch of trees on each computing node and take the mean of their results
- But the same idea does not work for a tree boosting algorithm
- Tree-boosting is a sequential tree building algorithm where the latter trees are built on the results of the previous fitted trees
- This cannot be parallelized like **random forests** do
- Instead the parallelism needs to come from fitting different branches of the same tree

What About the Gradient Boosting Classifiers?

- Like the other tree classifiers, **gradient boosting classifier** tries to approximate the probability density functions of all the classes simultaneously
- So naturally the backbone of the classification task is a multi-target regression task
- The difference from a usual boosting regressor is that it CANNOT use **MSE** as its loss function (why not?)
- For binary classification, boosting classifier uses **log loss**
- For multiclass classification, it uses a generalization of **log loss** called **cross-entropy loss** (also widely used in neural network/deep learning)
- **Cross-entropy loss** reduces to **log loss** when the number of classes drop to 2

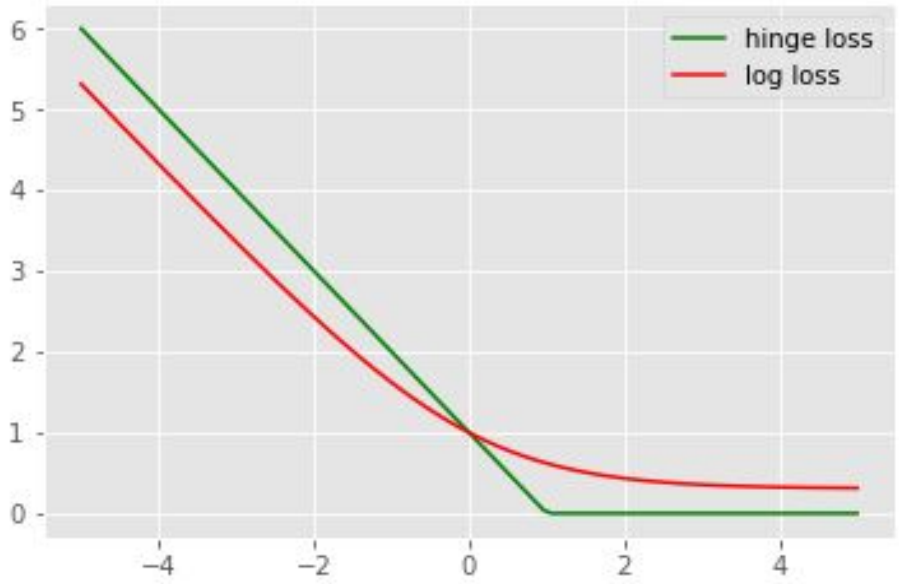
MSE Loss Function as an Upward Parabola

- **MSE/RSS** loss penalizes bigger discrepancy between true target and the prediction aggressively



Hinged Loss vs Log Loss

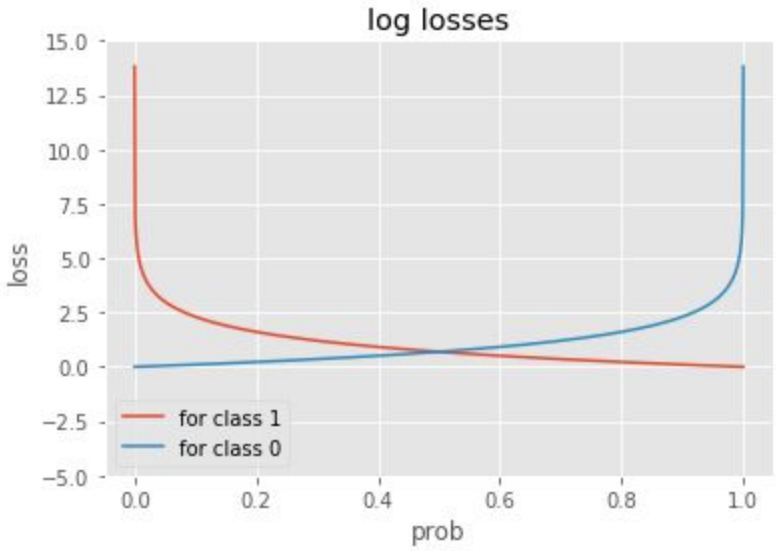
- The following plot compares the **hinged loss** used in **SVMs** vs **log loss** in logistic regression. **Log loss** is defined in the domain of 'log-odds' instead of probabilities, in order to be compared with **hinged loss** side by side

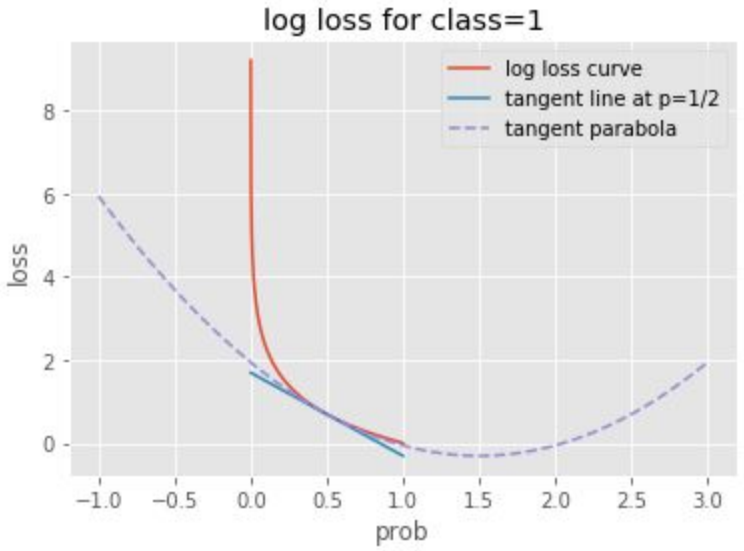


- N :ant) asymptotic to each other on the left

Local Tangent Parabola

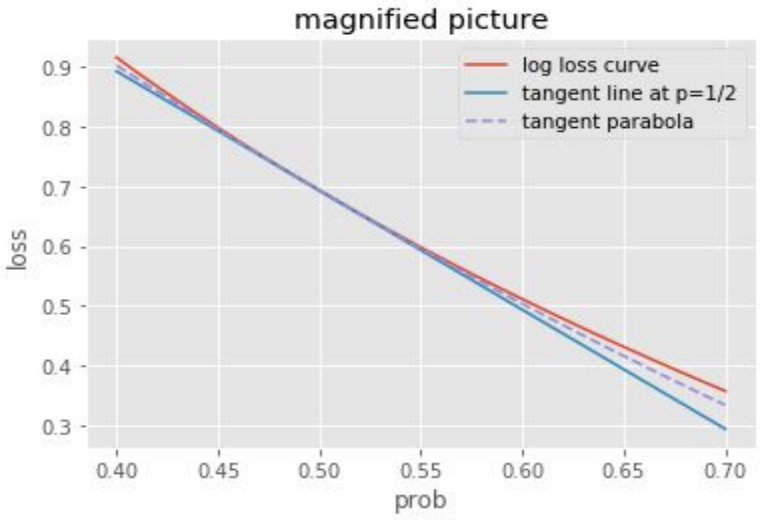
- **Per Sample Log loss** is a highly nonlinear function
- The **LHS** shows the per-sample **log loss** function plot for both classes
- For example taking p = 0.5, there is a unique tangent parabola to the **log loss** curve at p=0.5 (tangent slope = -2) with a leading term ½ p**2
- Different samples within different rectangular regions of the tree have their own tangent parabolas

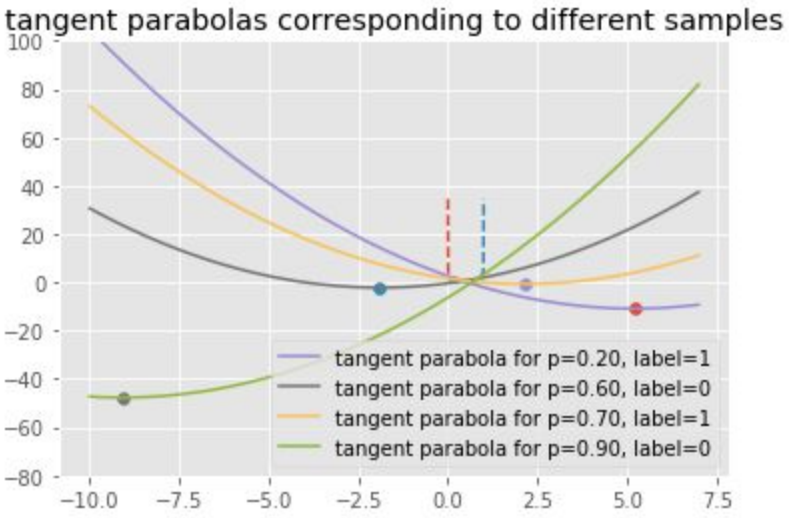




Multiple Tangent Parabolas

- Each sample has its own tangent parabola, which is effectively an **RSS**
- The colored dots represent the minimums of these parabolas, which are the optimal prediction value for that particular data point
- Notice the 'optimal' value is not always in the [0,1] interval
- Because all the samples within the same tree nodes share the same prediction value, different samples from the same node need to compromise





Friedman-MSE

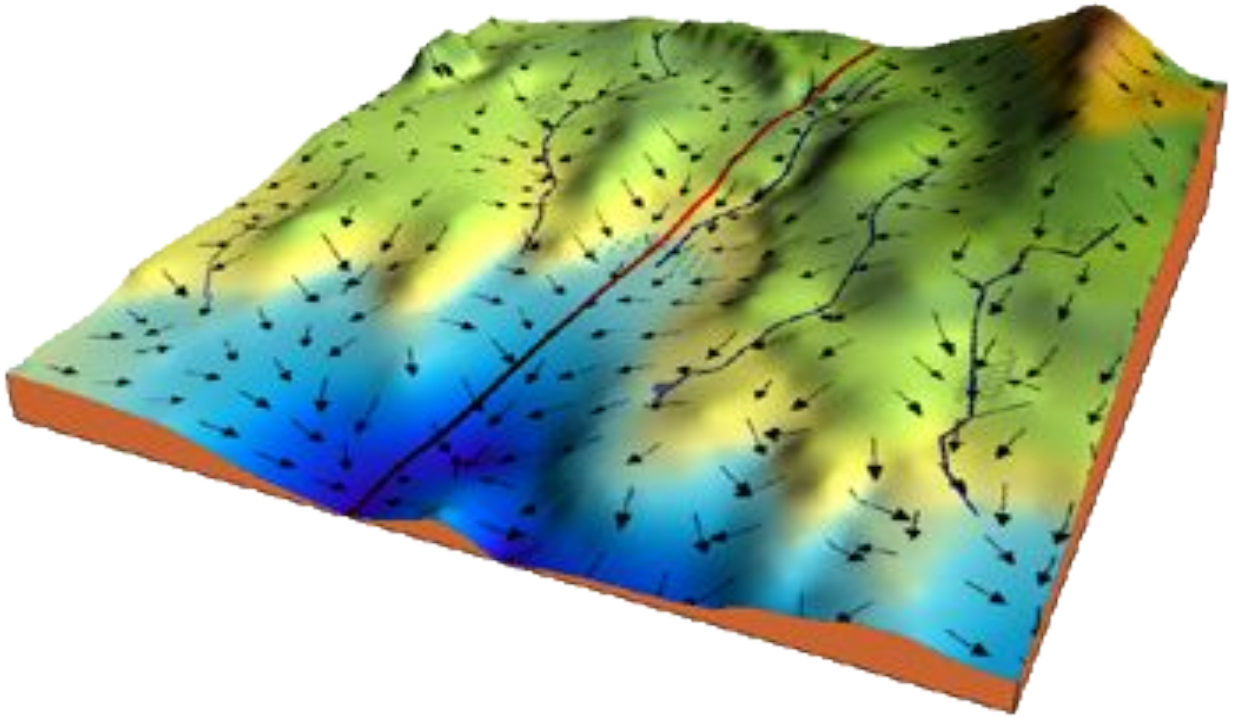
- The amount of horizontal-shifting from the tangent point to the optimal point (the unique minimum of the parabola) is equal to the slope of the tangent line at the tangent point!
- The amount of vertical-shifting from the tangent point to the optimal point is ½ of squared tangent slopes!
- It turns out that the mean value of all these sample-specific optimal values is the global optimum for the particular tree node. In other words, the mean of all tangent slopes associated with the samples in the box should be the optimal predicted value in each rectangular region
- Suppose that within a rectangular box there are q samples, with their tangent slopes denoted by $\alpha_i, 1 \leq i \leq q$, then the predicted shifting of the target value is by $\frac{\sum_{i \leq q} \alpha_i}{q}$, and the optimal vertical shift of the tree node is denoted by $\frac{\sum_{i \leq q} \alpha_i^2}{2}$. This is known as **Friedman-MSE** for optimizing the tree splitting (replacing **Gini/Information Entropy** for classification trees)

Gradient Descent vs Gradient Boosting

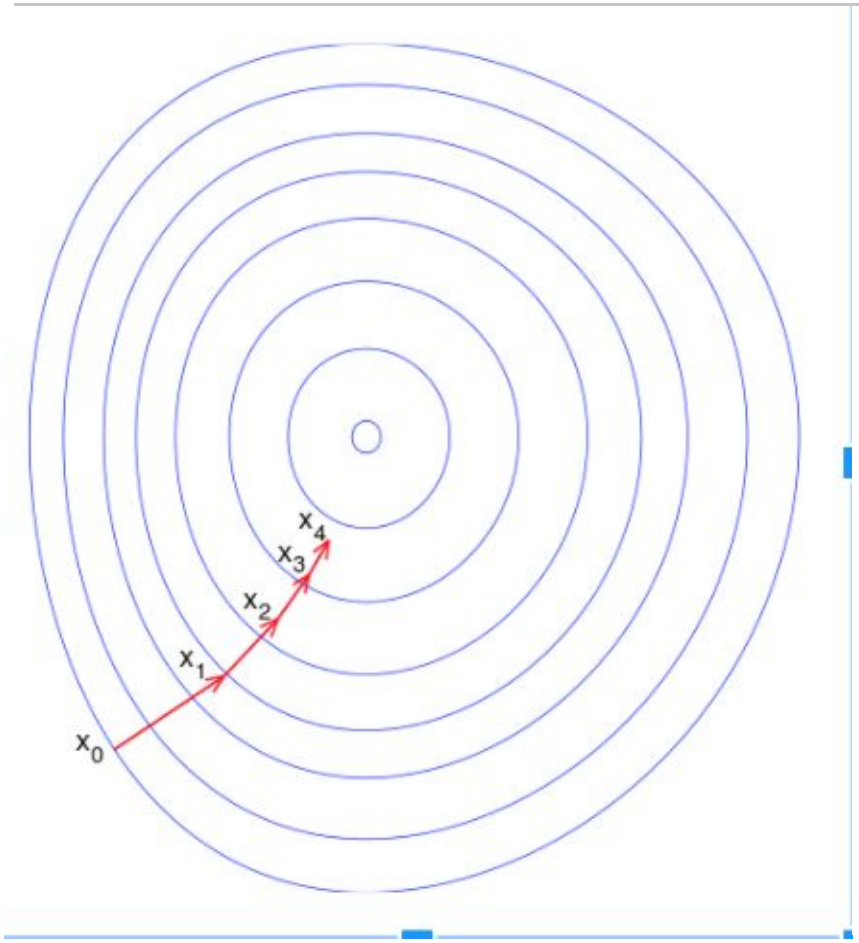
- Finally we are ready to tighten up the relationship between our previous discussion and **gradient descent** learning algorithm
- When the loss function is highly nonlinear, the aforementioned approach uses local tangent parabolas at different positions to replace the original nonlinear **log loss** function
- When the original loss function is the **RSS**/**MSE**, the tangent parabola of the **RSS** parabola is itself
- This reflects the fact that when y is the true target, and z is the predicted value, the tangent slope of per sample **RSS** = $\frac{(y-z)^2}{2}$ at z=y0 is simply y-y0, the error between the true and predicted target values
- This suggests that iteratively the kth update of the tree-boosting regression procedure should forecast the mean value of the errors

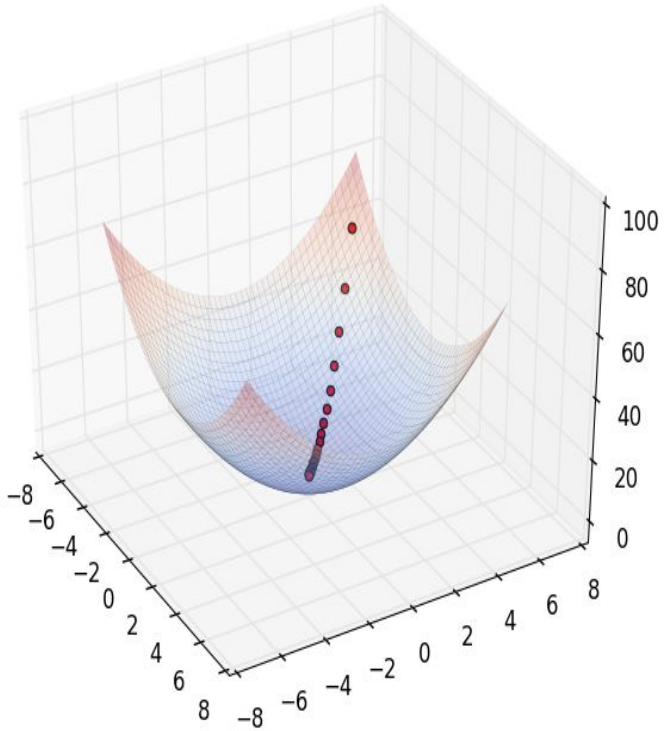
Gradient Descent vs Gradient Boost

- The **Negative Gradient Field** of the geographic domain which points to the steepest descent directions



Visualization on Gradient Descent





- The contour plot is on the left and the 3D plot is on the right

Boosting as Gradient Descent

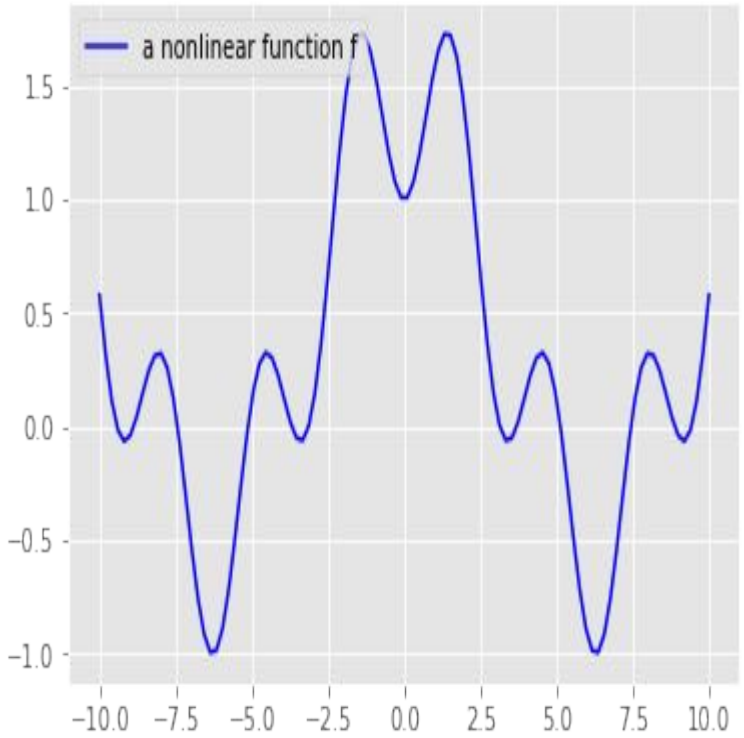
- On one hand, we have an iterative boosting procedure which updates the regression function (under a specific loss function like **RSS** or **log loss**) step by step using tangent parabolas
- On the other hand, **gradient descent** also updates the approximated solution step by step using the local gradient information
- It turns out the parallelism between these two iterative methodologies is not accidental. There is a way to view the aforementioned **boosting** algorithm as a **gradient descent** algorithm in some suitable 'function space'
- The details of the formulation and computation is beyond the scope of our discussion. Nevertheless, we will highlight the key idea in the following slides

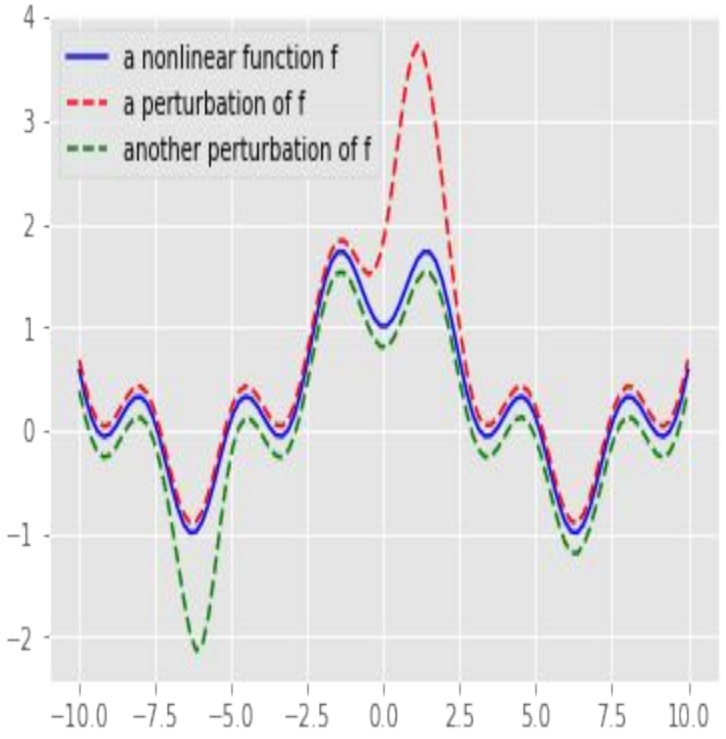
The Gradient Direction

- Suppose that we approximate the target function g by a known function f
- The quality of the fit is measured by the sum/mean of the per-training sample loss
- We would like to improve the approximation by updating f into $f + \epsilon h_1$
- The question is to find the best 'direction' (function) h_1 to add
- In the infinite dimensional space of functions, there are an infinite number of 'directions' to choose from; not all of them are born equal
- We need to choose the 'optimal' or 'near-optimal' direction (function) to update our original approximation f

Different Perturbations of a 1D Function

- **LHS** is the graph of a nonlinear function f, the red and green dash lines on the right are different perturbations of f





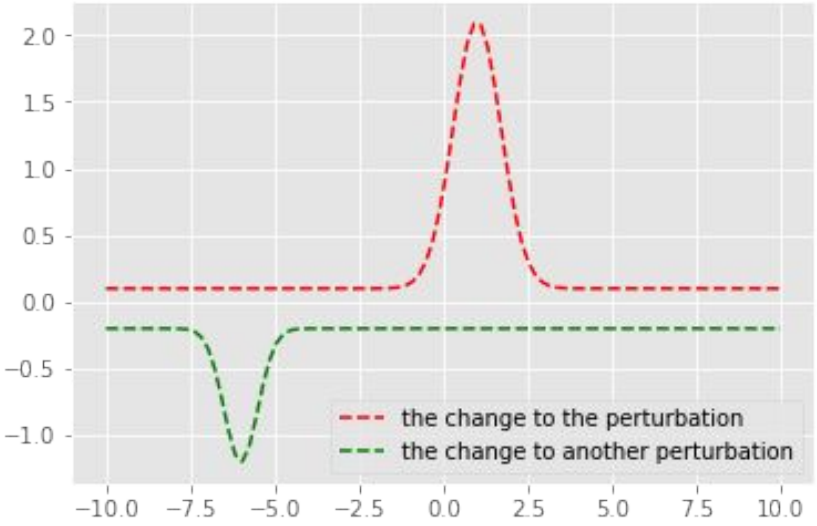
- There are an infinite dimension of perturbations of f

Training Samples and Optimal Perturbations

- The loss value is the sum/mean of all losses across all training samples
- This is known as **empirical loss** in the machine learning literature
- Two factors are crucial in deciding the 'optimal direction' to add to f
- The naive approach would perturb the original approximation **infinitesimally** along all the infinite number of directions to search for the one which minimizes the **empirical loss** (or equivalently maximizing the **empirical loss drop**). Thus it depends on the training samples
- Technically this is known as taking the '**functional derivative**'
- Often the 'optimal' direction might not be within the class of admissible functions which are allowed to be used in the **boosting** algorithm. One needs to project the naive optimal one to the subspace of admissible functions (say, the space of tree-functions for tree-boosting)

Gauging the Effectiveness of Perturbations

- Some perturbations increase the empirical loss values, some types of perturbations decrease the empirical loss values
- The following plot illustrates the functions which generate the aforementioned perturbations
- We would like to find the best direction which drops the empirical loss value the most, under suitable normalization conditions



Stochastic Gradient Descent and Gradient Boosting

- **Gradient boosting machines** implement bagging and random feature selections similar to what is done in a **random forest**
- If this facility is turned-on, it is parallel to **stochastic gradient descent** algorithm applied to the 'space of functions'
- Heuristically the **gradient boosting** establishes a sequential approximation to the given target function f, i.e. approximating the target function (or a probability function) through an iterative procedure generating a sequence of functions
- Just like the usual **gradient descent** establishes a sequential algorithm to approach the minimizer of the loss function by a sequence of points in the space of hyperparameters, the **gradient boosting** with **randomness** can be thought as a **stochastic gradient descent** establishing a sequence of functions to approximate the loss minimizer--the target function itself

Hands-on Session

- Please go to the **"Boosting in Scikit Learn"** in the lecture code.