Chapter 8.

Learning Methods

We review some of the most commonly used learning methods, including k-nearest neighbor (KNN), decision trees and random forests, support vector machines, neural nets and cluster analysis. The ensemble methods, also viewed as part of regularization in broad sense, can be very useful in combining weaker methods into a strong method. We shall consider both regresison and classification problems, with data $(y_i, \mathbf{x}_i), i = 1, ..., p$, where y_i are output and $\mathbf{x}_i$ are p-dimensional inputs. The regression model throughout this chapter is

$$y_i = f(\mathbf{x}_i) + \epsilon_i, ...i = 1, ..., n.$$

Unless otherwise stated, the loss function is the square loss. For classification problems, $y_i = \pm 1$, representing the two classes: class 1 and class -1. In this section, we will use the terms learners, predictors and estimators interchangeably.

8.1. K-nearest neighbor (KNN).

K-nearest neighbor is one of the simplest methods. For any $\mathbf{x}$ in the p-dimensional real space, let $\mathcal{K}(\mathbf{x})$ denote the indices of the data points that are the K closest to $\mathbf{x}$. In a regression problem, f is estimated by

$$\hat{f}(\mathbf{x}) = \frac{1}{K} \sum_{i \in \mathcal{K}(\mathbf{x})} y_i,$$

which is just the average of the responses of the K data points that are the closest to $\mathbf{x}$. The norm $\| \cdot \|$ is usually the Euclidean norm. Notice that $\hat{f}(\mathbf{x})$ is a step function of $\mathbf{x}$. In other words, the entire input space is partitioned into $\binom{n}{K}$ sets, each corresponding to one cluster of K data points. Arrange the totally $\binom{n}{K}$ such clusters as clusters $1, 2, ..., \binom{n}{K}$. The set corresponding to the j-th cluster are all the points that are closest to cluster j than to any other cluster of data points. Some of the sets could be empty. Notice that the smaller that K, the larger the model, and the larger the variance and the smaller the bias. For prediction purpose, the best K are generally determined by cross-validation.

For classification, simply classify a new observation with input $\mathbf{x}$ as from class 1 if $\hat{f}(\mathbf{x}) > 0$ and into class -1 otherwise. This is a majority vote scheme: classify the observation into the class which is the majority class among all the K nearest data points.

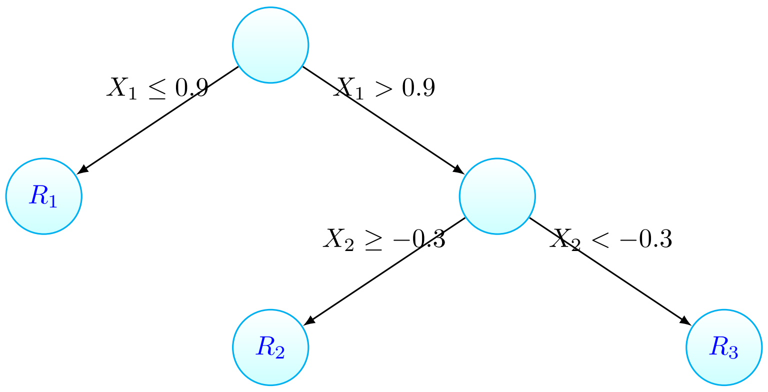
8.2. Classification and regression trees (CART).

Decision trees are popular nonlinear methods that have proper interpretation. In fact, they may even have better interpretation than linear regression models. Decision trees can also be viewed as step function regression. The predicted outputs are constants over specially constructed partitions of the input value space. In the following, we introduce the classification and regression trees (CART).

Under squared loss, if $\hat{f}$ is a constant over a region R, then $\hat{f}(\mathbf{x})$ is the average of all data points $\mathbf{x}_i$ in R. Regression trees are greedy algorithms to grow a tree that evenually leads to the partition of the input space. At every step, the algorithm determines the way to split a given region, according to the values of one variable. The critical issue is the choice of which variable and which cut point to split the data. Set $R^-_{j,s} = \{i : \mathbf{x}_i \in R, x_{ij} \le s\}$ and $R^+_{j,s} = \{i : \mathbf{x}_i \in R, x_{ij} > s\}$. Then R can be split into two regions $R^+_{j,s}$ and $R^-_{j,s}$ according whether the j-th variable is greater than s or not. The reduction of the residual sum of squares (RSS) is

$$\sum_{i \in R} (y_i - \bar{y})^2 - \Big(\sum_{i \in R^+_{j,s}} (y_i - \bar{y}^+)^2 + \sum_{i \in R^-_{j,s}} (y_i - \bar{y}^-)^2 \Big)$$

where $\bar{y}^+$ and $\bar{y}^-$ are the average of y_i for $i \in R^+_{j,s}$ and $i \in R^-_{j,s}$, respectively. Then, it is natural that one chooses the variable j and the cutpoint s so that the reduction of RSS is the largest, since the smaller the RSS, the better the fit. In this way, the method of regression trees defines a recursive binary split algorithm to partition the space. Specifically, first split the entire input space into two regions. then, for each of the two regions, perform the split by the same rule. Continue the split till stop. Every split must be choosing the best variable and best cutpoint to achieve the largest reduction of RSS. In the end, each partitioned area represents a terminal node, also called leaves, while every split corresponds to an internal node. The entire process resembles the growth of trees. One joins a node with the following and the line is viewed as branches. Traditionally, the tree-like structure of data-split is drawn upside-down. The size of the tree can be conveniently measured by the number of terminal nodes. A simple tree is shown in the following with three terminal nodes (leaves) and two internal nodes. The leaves are $R_1 = \{X_1 \le 0.9\}$; $R_2 = \{X_1 > 0.9, X_2 \ge -0.3\}$ and $R_3 = \{X_1 > 0.9, X_2 < -0.3\}$.



There is a major problem of when to stop the split. One way is to pre-specify a threshold for the amount reduction of RSS or for the number of data points in the region. A drawback with these types of stop rules is that they are usually near-sighted as they are based on the current step of data-split. A commonly adopted strategy is to grow a very large tree, which tends to overfit the data, and then perform a pruning of the tree. The *cost-complexity pruning* is the most popular with computational efficiency. For a large tree, denoted as T_0, we consider any of its subtrees T that are part of T_0 by collapsing any number of internal nodes. Consider the common regularization formula of "Loss + Penalty"

$$\sum_{m=1}^{|T|} \sum_{i \in R_m} (y_i - \bar{y}_m)^2 + \alpha |T|$$

where $|T|$ denotes the number of the terminal nodes of the subtree T, and R_m and the terminal nodes and $\bar{y}_m$ is the mean outputs of the data points in R_m, and α is tuning parameter. Denote the minimized subtree as T_α. If $\alpha = 0$, no penalty the optimal tree is the original T_0. If $\alpha = \infty$, the tree has no split at all. The predictor is just $\bar{y}$. The larger the α, the more penalty for model complexity. Just like Lasso, there exists efficient computation algorithm to compute the entire path of T_α for all α. Then we can use the cross-validation method to find the best α to minimize the test error.

For classification trees, one can follow the same line of procedure as that of regression trees, but using, instead of RSS, the error measurements that are more proper for classications. For a region R, let $\hat{p}_k$ be the percentage of observations in this region that belong to class k. We introduce an impurity measuare that corresponds to the RSS in regression. The typical impurity measures are

(a) The classification error rate (for this region R) is

$$E = 1 - \max_k \hat{p}_k.$$

(b) The Gini index is

$$G = \sum_{k=1}^{K} \hat{p}_k(1-\hat{p}_k)$$

(c) The cross-entropy is

$$D = -\sum_{k=1}^{K} \hat{p}_k \log(\hat{p}_k).$$

If a region R is nearly pure, most of the observations are from one class, then the Gini-index and cross-entropy would take smaller values than classfication error rate. Gini-index and cross-entropy are more sentive to node purity. To evaluate the quality of a particluar split, the Gini-index and cross-entropy are more popularly used as error measurement crietria than classification error rate. Any of these three approaches might be used when pruning the tree. The classification error rate is preferable if prediction accuracy of the final pruned tree is the goal.

Large regression trees are known to be of high variance, meaning that given different training data, the estimates can be quite different. As a result, prediction based on a single large tree can be unstable and inaccurate. A different way to boost the prediciton accuracy of trees is to grow many small trees and combine them together. This will be introduced in Section 8.5.

8.3. Support vector machines. (SVM)

Support vector machines are commonly applied to classification problems, though it also has application to regression. We consider here the classification problem. Consider a fixed p-dimensional vector $\mathbf{b} = (b_1, ..., b_p)$. All p-vector $\mathbf{x} = (x_1, ..., x_p)$ can be written as $a\mathbf{b}+\mathbf{z}$ where $\mathbf{z}$ is perpendicular to $\mathbf{b}$. Consider linear function

$$g(\mathbf{x}) =< \mathbf{b}, \mathbf{x} >= b_1x_1 + ... + b_px_p.$$

Then $g(\mathbf{x}) = 0$ defines a $p-1$ dimensional hyperplane, which is perpendicular to the vector $\mathbf{b} = (b_1, ..., b_p)^T$ and through origin. Denote this hyperplane as $\mathcal{A}_0$. And all the points of x satisfying $g(x) = c$ make a $p-1$ dimensional hyperplane that can be expressed as $c\mathbf{b}/\|\mathbf{b}\|^2 + \mathcal{A}_0$, denoted as $\mathcal{A}_c$. Thus a point $\mathbf{x}$ satifying $g(\mathbf{x}) - c > 0$ are on one side of $\mathcal{A}_c$ along the direction of $\mathbf{b}$, and those satisfying $g(\mathbf{x}) - c < 0$ are on the other side of $\mathcal{A}_c$. Moreover, $(g(\mathbf{x}) - c)/\|\mathbf{b}\|$ is the signed distance between $\mathbf{x}$ and $\mathcal{A}_c$.

Condier first the simple case that the data $\mathcal{D} = \{(y_i, \mathbf{x}_i), i = 1, ..., n\}$, where $y_i = \pm 1$, are separated by a hyperplane

$$f(\mathbf{x}) = \beta_0 + \beta_1x_1 + ... + \beta_px_p,$$

meaning that, for all data points i in class 1, $f(\mathbf{x}_i) > 0$, and they are all on one side of the hyperplane; and for all i in the other class -1, $f(\mathbf{x}_i) < 0$, and they are all on the other the other side of the hyperplane. It can be equivalently expressed as

$$y_i f(\mathbf{x}_i) > 0, \qquad \text{for all } i = 1, .., n.$$

A classification rule would simple classify a subject with input $\mathbf{x}$ into class 1 if $f(\mathbf{x}) > 0$ and into class -1 if $f(\mathbf{x}_i) < 0$.

A *maximal margin classifier* seeks to find the hyperplane such that the minimal distance of all points to the separating hyperplane is the smallest. This is the following optimization problem:

$$\text{maximize}_{\beta_0,\beta_1,...,\beta_p} M$$

$$\text{subject to } \sum_{j=1}^{p} \beta_j^2 = 1,$$

$$\text{and } y_i(\beta_0 + \beta_1 x_{i1} + ... + \beta_p x_{ip}) \geq M \text{ for all } i$$

where M is the half of the width of the strip that separates the data points in two classes. This strip is defined by $|f(\mathbf{x})| \leq M$. The data points in this strip or on the border of the strip is called support vectors.

In general, the two classes are usually not separable by any hyperplane, and, even if they are, the max-margin may not be desirable because of its high variance, and thus possible over-fit. The generalization of the maximal margin classifier to the non-separable case is known as the *support vector classifier,* where a soft-margin is used in place of the max margin. It results in greater robustness to individual observations, and better classification of most of the training observations. Soft-margin classifer (support vector classifier) allow some violation of the margin: some can be on the wrong side of the margin (in the river) or even wrong side of the hyperplane. The solution is the given by the following optimization procedure:

$$\text{maximize}_{\beta_0,\beta_1,...,\beta_p} M$$

$$\text{subject to} \qquad \sum_{j=1}^{p} \beta_j^2 = 1, \qquad \text{and}$$

$$y_i(\beta_0 + \beta_1 x_{i1} + ... + \beta_p x_{ip}) \geq M(1-\epsilon_i), \;\; \epsilon_i \geq 0 \text{ for all } i$$

$$\text{and } \sum_{i=1}^{n} \epsilon_i \leq C,$$

where C is a nonnegative tuning parameter, and ϵ_i are the so-called *slack variables.* The classification rule is: classify an observation with input $\mathbf{x}$ into class $+1$ if $f(\mathbf{x}) > 0$; else into -1 class.

To understand the slack variables ϵ_i, we note that $\epsilon_i = 0 \iff$ the i-th observation is on the correct side of the margin; $\epsilon_i > 0 \iff$ the i-th observation is on the wrong side of the margin; and $\epsilon_i > 1 \iff$ the i-th observation is on the wrong side of the hyperplane. The tuning parameter C is a *budget* for the amount that the margin can be violated by the n observations. $C = 0 \iff$ no budget and, as a result, $\epsilon_i = 0$ for all i. The classifier is a maximal margin classifier, which exists only if the two classes are separable by hyperplanes. Larger C implies more tolerance of margin violation. Note that more than C observations can be on the wrong side of the soft-margin classifier hyperplane. As C increases, the margin widens and more violations of the margin. Observations that lie directly on the margin, or on the wrong side of the margin for their class, are known as *support vectors.* Only the support vectors affect the support vector classifier. Those strictly on the correct side of the margin do not, just as outliers do not change the median. Larger C implies more violations, and more support vectors, and smaller variance and more robust classfier.

In summary, the linear support vector classifier can be represented as

$$f(\mathbf{x}) = \beta_0 + \sum_{i=1}^{n} \alpha_i < \mathbf{x}_i, \mathbf{x} >$$

where $\alpha_i \neq 0$ only for all support vectors. Moreover, α_i can also be computed based on $< \mathbf{x}_j, \mathbf{x}_k >$. Only the inner product of the feature space is relevant in computing the linaer support vector classfier. The above support vector classifier has a linear boundary, which may not be the "ground truth" in practice. To cope with more general cases, one can consider to *enlarge the feature space.* A straightfoward method is to include the power functions of the inputs. A better approach is the use of the *kernel trick*, which gives rise to the suppot vector machines. The support vector machine actually enlarges the original feature space to a space of kernel functions:

$$\mathbf{x}_i \to K(\cdot, \mathbf{x}_i).$$

The kernel functions are bivariate functions satisfying the property of nonnegative definiteness: $\sum_{i,j} a_i a_j K(\mathbf{x}_i, \mathbf{x}_j) \geq 0$. The original feature space is the p-dimensional input space. The enlarged feature space is the space of kernel functions, which is in fact of infinite dimension. In actual fitting of the support vector machine, we only need to compute the $K(\mathbf{x}_i, \mathbf{x}_j)$ for all $\mathbf{x}_i, \mathbf{x}_j$ in training data. And the support vector machine classifier can be written as

$$f(\mathbf{x}) = \beta_0 + \sum_{i=1}^{n} \alpha_i K(\mathbf{x}, \mathbf{x}_i)$$

The commonly used kernel functions are

(a) linear kernel $K(\mathbf{x}_i, \mathbf{x}_j) =< \mathbf{x}_i, \mathbf{x}_j >= \mathbf{x}_i^T \mathbf{x}_j$.
(b) polynomial kernel of degree d: $K(\mathbf{x}_i, \mathbf{x}_j) = (1+ < \mathbf{x}_i, \mathbf{x}_j >)^d$.
(c) Gaussian radial kernel: $K(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma \|\mathbf{x}_j - \mathbf{x}_j\|^2), \quad \gamma > 0$.

The key point here is that only the inner product of the feature space is relevant in computing the linear support vector classfier.

Mathematically, the SVM is a result of "hinge loss + ridge penalty":

$$\sum_{i=1}^{n} \max[0, 1 - y_i f(\mathbf{x}_i)] + \lambda \sum_{j=1}^{p} \beta_j^2.$$

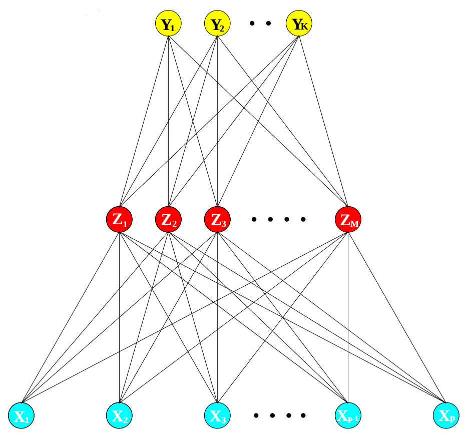
where $f(\mathbf{x}) = \beta_0 + \beta_1 x_1 + ... + \beta x_p$. The hinge loss function : $(1-x)_+$. In comparison, the classifier from the logistic regression with l_2 penalty is

$$\sum_{i=1}^{n} \log(1 + e^{-y_i f(\mathbf{x}_i)}) + \lambda \sum_{j=1}^{p} \beta_j^2$$

with the logistic loss function: $\log(1 + e^{-x})$.

8.4. Neural network

Neural network, particular deep learning, has achieved spectalur success in the past decades and is now one of the most successful learning methods. With deep architecture, the neural network is a particular kind of machine learning that achieves great power and flexibility by learning to represent the world as a nested hierarchy of concepts, with each concept defined in relation to simpler concepts, and more abstract representations computed in terms of less abstract ones. The main idea is to process, in a layerwise structure, linear combinations of the outputs of the previous layers by a nonlinear functions. These processors are called neural nets. In the basic feed-forward neural network, one can understand the network as informatoin flow forwardly from input $\mathbf{x}$ to output $\hat{y}$. The following picture shows a schematic of the neural network with three layers: input layer (bottom), hidden layer (middle) and output layer (top).



An optimization algorithm called backpropagation greatly reduces the computational complexity. We note that backpropagation only refers to a computation/optimization procedure on training data, and in fact, only the derivatives are passed backward. It does not mean the entire learning algorithm is backward propagation.

For the data with inputs-outputs: $(\mathbf{x}_i, y_i) : i = 1, ..., n$, we temporarily change the notation to $\mathbf{x}_i = (1, x_{i1}, ..., x_{ip})$ by including a constant term. Here we consider the "vanilla" neural nets, with only one hidden layer. Therefore, there are totally three layers, the input layer, the hidden layer and the output layer. Let σ be an activation function. The hidden layer is denoted as $\mathbf{h}_i$, a vector of $m+1$ dimension, implying m units in the hidden layer: $\mathbf{h}_i = (1, h_{i1}, ..., h_{im})^T$, where

$$h_{ij} = \sigma(\mathbf{w}_j^T \mathbf{x}_i); \quad j = 1, ..., m.$$

$\mathbf{w}_j$ is the j-th row of $\mathbf{W}$, which is $m \times (p+1)$ matrix. $\hat{y}_i = g(\beta^T \mathbf{h}_i)$, with activation function g. The parameters are $\theta = (\beta, \mathbf{W})$ of $m + 1 + m(p+1)$ dimension.

Consider the least squares loss:

$$\sum_{i=1}^n (y_i - \hat{y}_i)^2 = \sum_{i=1}^n (y_i - g(\beta^T \mathbf{h}_i))^2 = \sum_{i=1}^n R_i(\theta).$$

The gradient based minimization is to differentiate the loss function, as function of θ, step-by-step, *backwards.* The errors from an upper layer is propagated into the next lower layer as in the following precedure.

(a) Step 1: For top-layer (output layer):

$$\begin{aligned} \frac{\partial R_i(\theta)}{\partial \beta} &= -2(y_i - \hat{y}_i)\dot{g}(\beta^T \mathbf{h}_i)\mathbf{h}_i \\ &= \delta_i \mathbf{h}_i, \qquad \text{say.} \end{aligned}$$

(b) Step 2: For hidden layer:

$$\begin{aligned} \frac{\partial R_i(\theta)}{\partial \mathbf{w}_j} &= -2(y_i - \hat{y}_i)\dot{g}(\beta^T \mathbf{h}_i)\beta_j \frac{\partial h_{ij}}{\partial \mathbf{w}_j} \\ &= -2(y_i - \hat{y}_i)\dot{g}(\beta^T \mathbf{h}_i)\beta_j \dot{\sigma}(\mathbf{w}_j^T \mathbf{x}_i)\mathbf{x}_i \\ &= s_{ji}\mathbf{x}_i, \qquad \text{say.} \end{aligned}$$

Here δ_i and s_{ji} are the "errors" from the current model at the output and the hidden layer units. The errors satify

$$s_{ji} = \dot{\sigma}(\mathbf{w}_j^T \mathbf{x}_i)\delta_i \beta_j$$

These are the back-propagation equations, which can be used to fast update the parameters: (with learning rate γ)

$$\beta^{new} = \beta - \gamma \sum_{i=1}^n \frac{\partial R_i(\theta)}{\partial \beta} = \beta_j - \gamma \sum_{i=1}^n \delta_i \mathbf{h}_i$$

$$\mathbf{w}_j^{new} = \mathbf{w}_j - \gamma \sum_{i=1}^n \frac{\partial R_i(\theta)}{\partial \mathbf{w}_j} = \mathbf{w}_j - \gamma \sum_{i=1}^n s_{ji}\mathbf{x}_i.$$

The back-propagation can be understood as a two pass algorithm. The forward pass is: given the current parameters (weights), to compute $\hat{y}_i = g(\beta^T \mathbf{h}_i) = g(\beta^T \sigma(\mathbf{W}\mathbf{x}_i))$. The backward pass is: compute the error δ_i, then use the back-propagation equation to back-propagate it into the errors s_{ji} of the next lower layer, (and continue on if more layers are in the network). All the errors are then used to update the parameters with a proper choice of the learning rate. The advantages of the back-propagation are simplicity of computation; the local nature: the local nature: each unit passes and recieves information only to and from those units with connection; the batch learning: The parameter updates do not have to take place over all training samples, i.e., the summation could be on a subset (batch) of the training samples. It can even be just one single sample. This is widely used stochastic gradient descent (SGD).

8.5. Ensemble methods.

Ensemble methods generally have two tasks: 1. build a bank of base learners; 2. combining these base learners to form a final learner. These base learners are usually weak learners while the combined one is expected to be a strong learner.

Bagging. Bagging stands for boostrap aggregating. By perturbing the data, one generate many predictors based on the same model. Bagging perturbs the data by boostrap. Recall that $\hat{f}$ is the predictor based on original data $\mathcal{D} = \{(y_1, x_1), ..., (y_n, x_n)\})$. a boostrap sample, denoted as $(y_1^*, x_1^*), ..., (y_n^*, x_n^*)$, are a random sample from the original data $\mathcal{D}$, by sampling with replacement. Based on this boostrap sample, denoted as $\mathcal{D}_1^*$, one can construct a predictor $\hat{f}_1^*$. Repeat this process, one can construct many predictors, say, $\hat{f}_2^*, ..., \hat{f}_B^*$, where B can be as large as possible. Then the bagged predictor is

$$\hat{f}_{\text{bagging}}(x) = \frac{1}{B}\sum_{j=1}^{B} \hat{f}_j^*(x).$$

In the case of classification with two classes, 1 or -1, each $\hat{f}_j^*$ takes values 1 or -1. Then the bagged classifier is a majority-vote classifier: it classifies an new/old observation into that class which receives most "vote" from $\hat{f}_j^*$, $j = 1, ..., B$.

Associated with bagging, there is an easily available estimate of test error. Let

$$\hat{f}^*(x_i) = \frac{\sum_{j=1}^{B} \hat{f}_j^*(x_i) I((y_i, x_i) \notin \mathcal{D}_j^*)}{\sum_{j=1}^{B} I((y_i, x_i) \notin \mathcal{D}_j^*)}$$

be the average of the boostrap predictors based on the boostrap samples that do not contain the i-th data point (y_i, x_i). Then, the OOB (out-of-bag) mean squared error is

$$\sum_{i=1}^{n} (y_i - \hat{f}^*(x_i))^2,$$

which serves as an approriate estimator of the test mean squared error. In the case of classification, $\hat{f}^*(x_i)$ is the class of the majority vote of the OOB bootstrap classifiers. And the OOB classfication error is

$$(1/n)\sum_{i=1}^{n} (y_i \neq \hat{f}^*(x_i))$$

which is the estimator of the test classification error.

Boosting. The basic idea of boosting is sequentially add some base learners to the current learner in a way to keep improving predictive accuracy of the current learner, and eventually arrive at a final composite learner. At each step, the added base learner aims at predicting the residuals, the differences between the responses and the predicted responses from the current model, or predicting negative gradients of the loss function of the current fit. There exist various boosting methods. The basic boosting method using residuals can be described as follows. With a learning method, call it base learner, to produce $\hat{f}$ based on $\mathcal{D} = \{(y_i, x_i), i = 1, .., n\}$. Start with an initial predictor $\hat{f} = 0$. Set $r_i = y_i$. Then iterate as follows.

(a) Fit the data $(x_i, r_i), i = 1, .., n$, to produce a learner $\hat{g}$.

(b) Update $\hat{f}$ by $\hat{f} + \lambda\hat{g}$.

(c) Update r_i by $r_i - \lambda \hat{g}(x_i)$.

Continue this iteration till a stop, and output $\hat{f}$, which is the sum of $\lambda\hat{g}$ at each step. Here λ is the learning rate, which is usually chosen to be small. For a learning method that minimizes a loss function L, the boosting algorithm can be described as the following. Start with $\hat{f}_0(\mathbf{x})$ which is a constant γ such that $\sum_{i=1}^n L(y_i, \gamma)$ is minimized. For $k = 1, 2, ..., K$:

$$\hat{f}_k(\mathbf{x}) = \hat{f}_{k-1}(\mathbf{x}) + \hat{\lambda}\hat{g}(\mathbf{x})$$

where

$$(\hat{\lambda}, \hat{g}(\mathbf{x})) = \text{argmin}_{g\in\mathcal{G},\lambda} \sum_{i=1}^n L(y_i, \hat{f}_{k-1}(\mathbf{x}) + \lambda g(\mathbf{x}))$$

and $\mathcal{G}$ refers to the collection of learners for this learning method. For example, for the tree boosting, $\mathcal{G}$ is the collection of trees with certain size. The final output is $\hat{f}_K$. This is the *forward stagewise boosting algorithm.* Very often, the $\hat{\lambda}$ is not chosen to be the optimal one but a learning rate that is controled to be small.

The above minimization to obtain g might be computationally difficult. The gradient boosting is a short-cut to reduce the computational complexity. Set

$$g_{ik} = -\frac{\partial}{\partial z} L(y_i, z)|_{z=\hat{f}_{k-1}(\mathbf{x}_i)}$$

A base learner, denoted as $\hat{h}_k$, is fit to the data: $(g_{ik}, \mathbf{x}_i), i = 1, ..., n$. The iteration becomes

$$\hat{f}_k(\mathbf{x}) = \hat{f}_{k-1}(\mathbf{x}) + \lambda\hat{h}_k(\mathbf{x})$$

where λ is the learning rate. For square loss, the gradient boosting and residual boosting are the same.

Three techniques play important roles in boosting: 1. (shrinkage) the use of small learning rate; 2. Early stop to avoid overfit; 3. random subsampling to construct the base learners. The use of small learning rate may increase computational workload but the iteration can be more stable. Without early stopping, too many base learners putting together can cause overfit. The stop of the boosting iteration is mostly determined by checking the test error through validation or cross validation. The random subsampling is attempted to lower the correlation between the constructed base learners.

Tree boosting and random forest The basic form of boosting can be applied to the boosting trees, either based on residuals or based on gradients. Consider square loss, the the tree boosting algorithm is:

(a) $\hat{f}_0(\mathbf{x}) = \bar{y}$

(b) For $k = 1, ..., K$:

   i. set $r_i = y_i - \hat{f}_{k-1}(x_i), i = 1, ..., n$.

   ii. fit the data $(r_i, \mathbf{x}_i), i = 1, ..., n$, with a tree with J terminal nodes, denoted as $R_{k,j}, j = 1, ..., J$.

   iii. set $h_k(\mathbf{x}) = \sum_{j=1}^J \gamma_{k,j} I(\mathbf{x} \in R_{k,j})$ where $\gamma_{k,j}$ is the average of the residuals of data with inputs in $R_{k,j}$. i.e,

$$\gamma_{k,j} = \frac{1}{|R_{k,j}|} \sum_{\mathbf{x}_i \in R_{k,j}} r_i$$

   iv. update $\hat{f}_k(\mathbf{x}) = \hat{f}_{k-1}(\mathbf{x}) + \lambda h_k(\mathbf{x})$.

(c) output $\hat{f}_K$.

Boosted trees can greatly improve over the prediction of each single tree. In fact, each single tree in the process of boosting can be of small size, such as *stumps.* Stumps are trees with only two leaves.

Random forest is a method of tree boosting by bagging, with an additional element of creating less correlated trees. The specific procedure is as follows.

(a) For $k = 1, ..., B$:
   i. Draw boostrap sample $\mathcal{D}_k^* = \{(\mathbf{x}_1^*, y_1)..., (\mathbf{x}_n^*, y_n^*)\}$
   ii. Grow a tree T_k^* based on $\mathcal{D}_k^*$ in the following fashion. Before every split, select m variables at random from the p variables, and pick the best variable and cutpoint, among these m variables to perform the split.

(b) Output $\hat{f}(\mathbf{x}) = (1/B)\sum_{k=1}^B T_k^*(\mathbf{x})$.

For classification problem, each tree T_k^* provide a class prediction for input $\mathbf{x}$, and the random forest outputs the majority-vote of them as the predicted class for $\mathbf{x}$. The purpose of the random subset selection in step (ii) is to create less correlated boostrap trees, so that the bagging would have effect on variance reduction. A common choise of m is $\sqrt{p}$ or smaller, and it can be as small as 1.

Stacking. Stacking is also called stacked generalization. Suppose we have M models with predictors $\hat{f}_1, ..., \hat{f}_J$ based on the training data $(y_i, \mathbf{x}_i), i = 1, ..., n$. A simple idea is to consider linear combination of the predictors to hopefully form a predictor of better accuracy. Let $w = (w_1, ..., w_J)^T$ be the weights. One might consider

$$\sum_{i=1}^n (y_i - \sum_{j=1}^J w_j \hat{f}_j(\mathbf{x}_i))^2,$$

and wish to minize this training error to find the optimal weights. In terms of computation, the minimizer is easily solved and has an expression like the least squares estimator. However, such a minimization can easily lead to large weights on large models and small weights on small models. In other words, the model complexity is not considered in the model combination. Stacking deals with the problem by the leave-one-out-cross-validation (LOOCV). Let $\hat{f}_j^{(-i)}$ be the estimator of f based on model j and the data excluding the i-th data point $(y_i, \mathbf{x}_i)$. Let

$$\hat{w} = \text{argmin}_w \sum_{i=1}^n (y_i - \sum_{j=1}^J w_j \hat{f}_j^{(-i)}(\mathbf{x}_i))^2.$$

The resulting stacked predictor is $\sum_{j=1}^n \hat{w}_j \hat{f}_j$. Alternatively, one can also restrict $\hat{w}_j$ to nonnegative in the above minimization. It can be proved that, at the population level, the weighted model averaging with optimal weights produce a model as good as or better than any of the individual model, in the sense that it has smaller error than any single model.

There exist many other ensemble methods. The richest is the Bayesian model averaging or combination. In broad sense, the regularized regression such as Lasso or ridge regression where there are tuning parameters can also be viewed as ensemble methods, called *Buckets of models.* There is a collection of models, each model indexed by a tuning parameter. Buckets of models method tries to select the best of them, by using, for example, cross validation. Another method called *bumping* also finds a best single model from a bucket of models by boostrap sampling. let the original prediction based on the training data be $\hat{f}$. Draw B sets of boostrap samples, and fit each boostrap sample to generate boostrap prediction $\hat{f}_j^*$, with $j = 1, .., B$. Choose the the prediction among $\hat{f}, \hat{f}_1^*, ..., \hat{f}_B^*$ with the smallest training error over the entire training data, which is $\sum_{i=1}^n (y_i - \hat{f}_j^*(\mathbf{x}_i))^2$ for squared loss and prediction $\hat{f}_j^*$. Bumping picks the best prediction in terms of training error, while bagging uses the average boostrap predictions.