UNIT 3

Generalized Additive Models

A Generalised Additive Model (GAM) is an extension of the multiple linear model, which recall is

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_p x_p + \epsilon.$$

In order to allow for non-linear effects a GAM replaces each linear component
βjxj with a smooth non-linear function
fj(xj)

$$y = \beta_0 + f_1(x_1) + f_2(x_2) + \ldots + f_p(x_p) + \epsilon.$$

This is called an additive model because we estimate each fj(xj) for j=1,2,3,....,p and then add together all of these individual contributions.
Additive models provide a useful extension of linear models, making them more flexible while still retaining much of their interpretability. The familiar tools for modeling and inference in linear models are also available for additive models. The backfitting procedure for fitting these models is simple and modular, allowing one to choose a fitting method appropriate for each input variable. As a result they have become widely used in the statistical community. However additive models can have limitations for large data-mining applications. The backfitting algorithm fits all predictors, which is not feasible or desirable when a large number are available

Fitting Methods:

Backfitting Algorithm

Algorithm 9.1 *The Backfitting Algorithm for Additive Models.*

1. Initialize: $\hat{\alpha} = \frac{1}{N}\sum_1^N y_i$, $\hat{f}_j \equiv 0, \forall i, j$.

2. Cycle: $j = 1, 2, \ldots, p, \ldots, 1, 2, \ldots, p, \ldots,$

$$\hat{f}_j \leftarrow \mathcal{S}_j\left[\{y_i - \hat{\alpha} - \sum_{k \neq j} \hat{f}_k(x_{ik})\}_1^N\right],$$

$$\hat{f}_j \leftarrow \hat{f}_j - \frac{1}{N}\sum_{i=1}^N \hat{f}_j(x_{ij}).$$

until the functions $\hat{f}_j$ change less than a prespecified threshold.

This same algorithm can accommodate other fitting methods in exactly the same way, by specifying appropriate smoothing operators Sj :

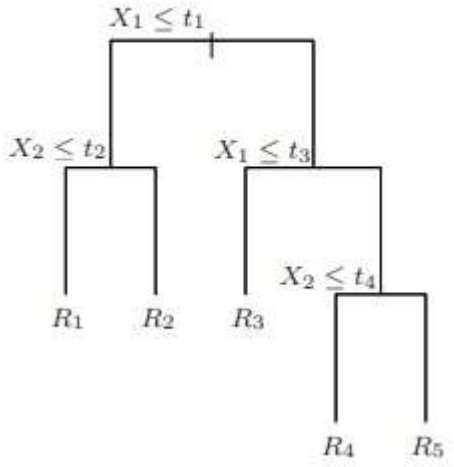
- other univariate regression smoothers such as local polynomial regression and kernel methods;
- linear regression operators yielding polynomial fits, piecewise constant fits, parametric spline fits, series and Fourier fits;
- more complicated operators such as surface smoothers for second or higher-order interactions or periodic smoothers for seasonal effects.

Regression and Classification Trees

Tree-based methods partition the feature space into a set of rectangles, and then fit a simple model (like a constant) in each one. They are conceptually simple yet powerful. We first describe a popular method for tree-based regression and classification called CART, and later contrast it with C4.5, a major competitor.

Tree Model

The full dataset sits at the top of the tree. Observations satisfying the condition at each junction are assigned to the left branch, and the others to the right branch.



All regression techniques contain a single output (response) variable and one or more input (predictor) variables. The output variable is numerical. The general regression tree building methodology allows input variables to be a mixture of continuous and categorical variables. A decision tree is generated when each decision node in the tree contains a test on some input variable's value. The terminal nodes of the tree contain the predicted output variable values.

A Regression tree may be considered as a variant of decision trees, designed to approximate real-valued functions, instead of being used for classification methods.

Building a regression tree

A regression tree is built through a process known as binary recursive partitioning, which is an iterative process that splits the data into partitions or branches, and then continues splitting each partition into smaller groups as the method moves up each branch.

Initially, all records in the Training Set (pre-classified records that are used to determine the structure of the tree) are grouped into the same partition. The algorithm then begins allocating the data into the first two partitions or branches, using every possible binary split on every field. The algorithm selects the split that minimizes the sum of the squared deviations from the mean in the two separate partitions. This splitting rule is then applied to each of the new branches. This process continues until each node reaches a user-specified minimum node size

and becomes a terminal node. (If the sum of squared deviations from the mean in a node is zero, then that node is considered a terminal node even if it has not reached the minimum size.)

Pruning the Tree

Since the tree is grown from the Training Set, a fully developed tree typically suffers from over-fitting (i.e., it is explaining random elements of the Training Set that are not likely to be features of the larger population). This over-fitting results in poor performance on real life data. Therefore, the tree must be pruned using the Validation Set. Analytic Solver Data Mining calculates the cost complexity factor at each step during the growth of the tree and decides the number of decision nodes in the pruned tree. The cost complexity factor is the multiplicative factor that is applied to the size of the tree (measured by the number of terminal nodes).

The tree is pruned to minimize the sum of:

1) the output variable variance in the validation data, taken one terminal node at a time; and

2) the product of the cost complexity factor and the number of terminal nodes. If the cost complexity factor is specified as zero, then pruning is simply finding the tree that performs best on validation data in terms of total terminal node variance.

Larger values of the cost complexity factor result in smaller trees. Pruning is performed on a last-in first-out basis, meaning the last grown node is the first to be subject to elimination.

The Regression Tree Algorithm can be used to find one model that results in good predictions for the new data

Classification Trees

If the target is a classification outcome taking values 1, 2,... ,K, the only changes needed in the tree algorithm pertain to the criteria for splitting nodes and pruning the tree. For regression we used the squared-error node impurity measure , but this is not suitable for classification.

Classification tree methods (i.e., decision tree methods) are recommended when the data mining task contains classifications or predictions of outcomes, and the goal is to generate rules that can be easily explained and translated into SQL or a natural query language.

A Classification tree labels, records, and assigns variables to discrete classes. A Classification tree can also provide a measure of confidence that the classification is correct.

A Classification tree is built through a process known as binary recursive partitioning. This is an iterative process of splitting the data into partitions, and then splitting it up further on each of the branches.

Building a Classification Tree

The process starts with a Training Set consisting of pre-classified records (target field or dependent variable with a known class or label such as purchaser or non-purchaser). The goal is to build a tree that distinguishes among the classes. For simplicity, assume that there are only two target classes, and that each split is a binary partition. The partition (splitting) criterion generalizes to multiple classes, and any multi-way partitioning can be achieved through repeated binary splits. To choose the best splitter at a node, the algorithm considers each input field in turn. In essence, each field is sorted. Every possible split is tried and considered, and the best split is the one that produces the largest decrease in diversity of the classification label within each partition (i.e., the increase in homogeneity). This is repeated for all fields, and the winner is chosen as the best splitter for that node. The process is continued at subsequent nodes until a full tree is generated.

XLMiner uses the Gini index as the splitting criterion, which is a commonly used measure of inequality. The index fluctuates between a value of 0 and 1. A Gini index of 0 indicates that all records in the node belong to the same category. A Gini index of 1 indicates that each record in the node belongs to a different category

Pruning the Tree

Pruning is the process of removing leaves and branches to improve the performance of the decision tree when moving from the Training Set (where the classification is known) to real-world applications (where the classification is unknown). The tree-building algorithm makes the best split at the root node where there are the largest number of records, and considerable information. Each subsequent split has a smaller and less representative population with which to work. Towards the end, idiosyncrasies of training records at a particular node display patterns that are peculiar only to those records. These patterns can become meaningless for prediction if you try to extend rules based on them to larger populations.

For example, if the classification tree is trying to predict height and it comes to a node containing one tall person X and several other shorter people, the algorithm decreases diversity at that node by a new rule imposing people named X are tall, and thus classify the Training Data. In the real world, this rule is obviously inappropriate. Pruning methods solve this problem -- they let the tree grow to maximum size, then remove smaller branches that fail to generalize. (Note: Do not include irrelevant fields such as name, as this is simply used an illustration.)

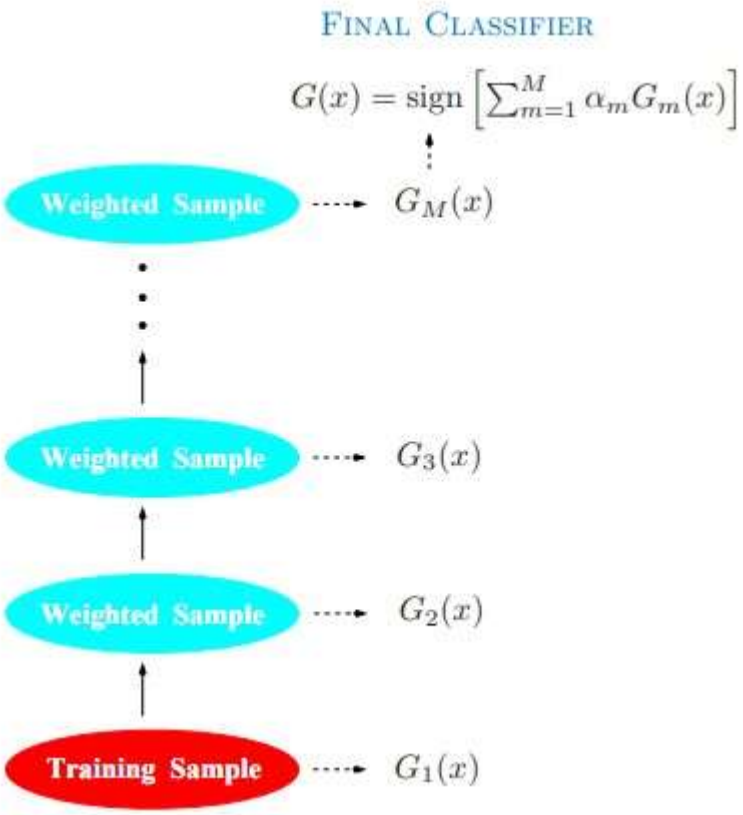
Since the tree is grown from the Training Set, when it has reaches full structure it usually suffers from over-fitting (i.e., it is explaining random elements of the Training Data that are not likely to be features of the larger population of data). This results in poor performance on data. Therefore, trees must be pruned using the Validation Set.

Boosting Methods

Boosting is one of the most powerful learning ideas introduced in the last twenty years. It was originally designed for classification problems, but as will be seen in this chapter, it can profitably be extended to regression as well. The motivation for boosting was a procedure that combines the outputs of many “weak” classifiers to produce a powerful “committee.” From this perspective boosting bears a resemblance to bagging and other committee-based approaches . However we shall see that the connection is at best superficial and that boosting is fundamentally different.

Exponential loss and AdaBoost

A weak classifier is one whose error rate is only slightly better than random guessing. The purpose of boosting is to sequentially apply the weak classification algorithm to repeatedly modified versions of the data, thereby producing a sequence of weak classifiers Gm(x),m = 1, 2,... ,M.



Adaboost (Adaptive Boosting)

Adaboost combines multiple weak learners into a single strong learner. This method does not follow Bootstrapping. However, it will create different decision trees with a single split (one depth), called decision stumps. The number of decision stumps it will make will depend on the number of features in the dataset. Suppose there are M features then, Adaboost will create M decision stumps.

1. We will assign an equal sample weight to each observation.

2. We will create M decision stumps, for M number of features.

3. Out of all M decision stumps, I first have to select one best decision tree model. For selecting it, we will either calculate the Entropy or Gini coefficient. The model with lesser entropy will be selected (means model that is less disordered).

4. Now, after the first decision stump is built, an algorithm would evaluate this decision and check how many observations the model has misclassified.

5. Suppose out of N observations, The first decision stump has misclassified T number of observations.

6. For this, we will calculate the total error (TE), which is equal to T/N.

7. Now we will calculate the performance of the first decision stump.
   Performance of stump = 1/2*loge((1-TE)/TE)

8. Now we will update the weights assigned before. To do this, we will first update the weights of those observations, which we have misclassified. The weights of wrongly classified observations will be increased and the weights of correctly classified weights will be reduced.

9. By using this formula: old weight * e performance of stump

10. Now respectively for each observation, we will add and subtract the updated weights to get the final weights.

11. But these weights are not normalized that is their sum is not equal to one. To do this, we will sum them and divide each final weight with that sum.

12. After this, we have to make our second decision stump. For this, we will make a class intervals for the normalized weights.

13. After that, we want to make a second weak model. But to do that, we need a sample dataset on which the second weak model can be run. For making it, we will run N number of iterations. On each iteration, it will calculate a random number ranging between 0-1 and this random will be compared with class intervals we created and on which class interval it lies, that row will be selected for sample data set. So new sample data set would also be of N observation.

14. This whole process will continue for M decision stumps. The final sequential tree would be considered as the final tree.

Algorithm 10.1 *AdaBoost.M1.*

1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to M:
   (a) Fit a classifier $G_m(x)$ to the training data using weights w_i.
   (b) Compute
   $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
   (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
   (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.

However, as boosting iterations proceed the error rate steadily decreases, reaching 5.8% after 400 iterations. Thus, boosting this simple very weak classifier reduces its prediction error rate by almost a factor of four. It also outperforms a single large classification tree (error rate

24.7%). Since its introduction, much has been written to explain the success of AdaBoost in producing accurate classifiers. Most of this work has centered on using classification trees as the “base learner” G(x), where improvements are often most dramatic.

The AdaBoost.M1 algorithm was originally motivated from a very different perspective than presented in the previous section. Its equivalence to forward stagewise additive modeling based on exponential loss was only discovered five years after its inception. By studying the properties of the exponential loss criterion, one can gain insight into the procedure and discover ways it might be improved. The principal attraction of exponential loss in the context of additive modeling is computational; it leads to the simple modular reweighting AdaBoost algorithm. However, it is of interest to inquire about its statistical properties. What does it estimate and how well is it being estimated? The first question is answered by seeking its population minimizer. It is easy to show

$$f^*(x) = \arg\min_{f(x)} \mathrm{E}_{Y|x}(e^{-Yf(x)}) = \frac{1}{2}\log\frac{\Pr(Y=1|x)}{\Pr(Y=-1|x)},$$

$$\Pr(Y=1|x) = \frac{1}{1+e^{-2f^*(x)}}.$$

Example: Spam Data

. The spam data are introduced in Chapter 1, and used as an example for many of the procedures in Chapter 9 (Sections 9.1.2, 9.2.5, 9.3.1 and 9.4.1). Applying gradient boosting to these data resulted in a test error rate of 4.5%, using the same test set as was used in Section 9.1.2. By comparison, an additive logistic regression achieved 5.5%, a CART tree fully grown and 10.9 Boosting Trees 353 pruned by cross-validation 8.7%, and MARS 5.5%. The standard error of these estimates is around 0.6%, although gradient boosting is significantly better than all of them using the McNemar test (Exercise 10.6). In Section 10.13 below we develop a relative importance measure for each predictor, as well as a partial dependence plot describing a predictor’s contribution to the fitted model. We now illustrate these for the spam data. Figure 10.6 displays the relative importance spectrum for all 57 predictor variables. Clearly some predictors are more important than others in separating spam from email. The frequencies of the character strings !, $, hp, and remove are estimated to be the four most relevant predictor variables. At the other end of the spectrum, the character strings 857, 415,

table, and 3d have virtually no relevance. The quantity being modeled here is the log-odds of spam versus email
f(x) = log Pr(spam|x) Pr(email|x) .

the partial dependence of the log-odds on selected important predictors, two positively associated with spam (! and remove), and two negatively associated (edu and hp). These particular dependencies are seen to be essentially monotonic. There is a general agreement with the corresponding functions found by the additive logistic regression model; see Figure 9.1 on page 303. Running a gradient boosted model on these data with J = 2 terminalnode trees produces a purely additive (main effects) model for the logodds, with a corresponding error rate of 4.7%, as compared to 4.5% for the full gradient boosted model (with J = 5 terminal-node trees). Although not significant, this slightly higher error rate suggests that there may be interactions among some of the important predictor variables. This can be diagnosed through two-variable partial dependence plots. shows one of the several such plots displaying strong interaction effects. One sees that for very low frequencies of hp, the log-odds of spam are greatly increased. For high frequencies of hp, the log-odds of spam tend to be much lower and roughly constant as a function of !. As the frequency of hp decreases, the functional relationship with ! strengthens. **For diagrams refer to the textbook**

Numerical Optimization via Gradient Boosting

Gradient boosting is powerful, it's essential to tune hyperparameters carefully to avoid overfitting and achieve optimal performance.

Step 1: Decision Trees
Gradient boosting is an ensemble learning method, and it usually starts with a decision tree. Decision trees are simple models that make decisions based on features. They split the data into subsets, and each subset is then split further until a stopping criterion is met.

Step 2: Residuals
After the first tree, we calculate the residuals (the differences between the actual values and the predicted values). These residuals represent what the first tree couldn't capture.

Step 3: Weighted Trees
We build a second tree to predict these residuals. This tree is then multiplied by a small learning rate (usually less than 1) to avoid overfitting.

Step 4: Update Predictions

The predictions from the second tree are added to the predictions of the first tree, and this updated prediction is used for the next round.

Step 5: Iteration

Steps 2-4 are repeated for a predefined number of iterations or until a certain level of accuracy is reached.

Step 6: Combining Trees

All the trees' predictions are combined to make the final prediction. The result is a strong predictive model built from many weak models.

Algorithm 10.3 *Gradient Tree Boosting Algorithm.*

1. Initialize $f_0(x) = \arg\min_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$.
2. For $m = 1$ to M:
   (a) For $i = 1, 2, \ldots, N$ compute
   $$r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}.$$
   (b) Fit a regression tree to the targets r_{im} giving terminal regions R_{jm}, $j = 1, 2, \ldots, J_m$.
   (c) For $j = 1, 2, \ldots, J_m$ compute
   $$\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L\left(y_i, f_{m-1}(x_i) + \gamma\right).$$
   (d) Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$.
3. Output $\hat{f}(x) = f_M(x)$.

Fast approximate algorithms for solving (10.29) with any differentiable loss criterion can be derived by analogy to numerical optimization. The loss in using f(x) to predict y on the training data is

$$L(f) = \sum_{i=1}^{N} L(y_i, f(x_i)).$$

The goal is to minimize L(f) with respect to f, where here f(x) is constrained to be a sum of trees (10.28). Ignoring this constraint, minimizing (10.33) can be viewed as a numerical optimization where the "parameters" f ∈ IRN are the values of the approximating function f(xi) at each of the N data points xi :

$$\hat{\mathbf{f}} = \arg\min_{\mathbf{f}} L(\mathbf{f}),$$

$$\mathbf{f} = \{f(x_1), f(x_2)), \ldots, f(x_N)\}.$$

Numerical optimization procedures solve (10.34) as a sum of component vectors where f0 = h0 is an initial guess, and each successive fm is induced based on the current parameter vector fm−1, which is the sum of the previously induced updates. Numerical optimization methods differ in their prescriptions for computing each increment vector hm ("step").

$$\mathbf{f}_M = \sum_{m=0}^{M} \mathbf{h}_m, \quad \mathbf{h}_m \in \mathbb{R}^N,$$

Key Points:

- Shrinking: The learning rate acts as a shrinkage factor, reducing the impact of each tree, and is a regularization technique.
- Tree Depth and Structure: Trees are usually shallow to avoid overfitting. Stumps (trees with one node and two leaves) are commonly used.
- Loss Function: Gradient boosting minimizes a loss function (e.g., Mean Squared Error for regression problems, or deviance for classification problems).
- Regularization: Gradient boosting includes regularization techniques to prevent overfitting.

California Housing

This data set (Pace and Barry, 1997) is available from the Carnegie-Mellon StatLib repository2 . It consists of aggregated data from each of 20,460 neighborhoods (1990 census block groups) in California. The response variable Y is the median house value in each neighborhood measured in units of $100,000. The predictor variables are demographics such as median income MedInc, housing density as reflected by the number of houses House, and the average occupancy in each house AveOccup. Also included as predictors are the location of each neighborhood (longitude and latitude),

and several quantities reflecting the properties of the houses in the neighborhood: average number of rooms AveRooms and bedrooms AveBedrms. There are thus a total of eight predictors, all numeric. We fit a gradient boosting model using the MART procedure, with J = 6 terminal nodes, a learning rate (10.41) of ν = 0.1, and the Huber loss criterion for predicting the numeric response. We randomly divided the dataset into a training set (80%) and a test set (20%). Figure 10.13 shows the average absolute error AAE = E|y − ˆfM(x)| (10.53) as a function for number of iterations M on both the training data and test data. The test error is seen to decrease monotonically with increasing M, more rapidly during the early stages and then leveling off to being nearly constant as iterations increase.

Thus, the choice of a particular value of M is not critical, as long as it is not too small. This tends to be the case in many applications. The shrinkage strategy (10.41) tends to eliminate the problem of overfitting, especially for larger data sets. The value of AAE after 800 iterations is 0.31. This can be compared to that of the optimal constant predictor median{yi} which is 0.89. In terms of more familiar quantities, the squared multiple correlation coefficient of this model is R2 = 0.84. Pace and Barry (1997) use a sophisticated spatial autoregression procedure, where prediction for each neighborhood is based on median house values in nearby neighborhoods, using the other predictors as covariates. Experimenting with transformations they achieved R2 = 0.85, predicting log Y . Using log Y as the response the corresponding value for gradient boosting was R2 = 0.86.

Note: please refer the textbook for diagramgms and other examples.