Modern Applied Statistics: Data Mining

Modern Applied Statistics: Data Mining

Introduction

- Boosting / additive trees very popular for unorganized data; now often most accurate for competitions and predictions
- Unobserved inputs Z that affect y in addition to X - why we cannot perfectly predict y. Instead we have a loss L. Alternatively can say $y = f(\mathbf{x}) + \varepsilon$, for $\mathbf{x} = (x_1, x_2, \ldots, x_m)$.
- ε usually approximated as normal, though almost never is and usually not symmetric. There exist procedures to more accurately estimate its distribution if needed.
- Important to remembered squared error loss is not necessarily the most appropriate for a regression problem but is the easiest to work with.
- Classification problem is simply assigning a name. There are no degrees of being wrong, either the right label assigned or it wasn't. Therefore the loss is a matrix of values, rows prediction, truth on the columns. All zeros on the diagonal since those are correct predictions, then losses elsewhere.
- Classification for assigning probabilities looks more like the regression problem since there are then degrees of wrongness.
- **Risk** $R(F) = E_{\mathbf{x}y} L(y, F(\mathbf{x}))$ - the average loss over future predictions. For each prediction I will lose something, and I want that to be as small as possible on average.
- There will be a function that has the minimum prediction risk, and this is the predicting function we want to use. Optimal target function $F^* = \arg\min_F R(F)$. The target function, though it's the best possible, may not be very good - the unseen Z contributors may overwhelm our observed predictors.
- The risk $R(F)$ is over the joint distribution of x and y, over all the variables, but in order to take an expected value over the joint distribution we need to know this distribution. Since we do not know this distribution, we estimate it from a training sample where we have x and corresponding y.
- Concept drift - the distribution may change over time compared to our training sample, but we often pretend otherwise for our procedures.
- All linear procedures can be written as $\hat{F}(x) = \sum_{i=1}^{N} H(x, x_i)\, y_i$ where H does not depend on y and is a function of the x where the y value is and the value of x at which we want to make a prediction. Nearly all theory relies on fixed x's - meaning the new x's will look just like the one's we have in our training set, exact overlap of values.

Choosing a Method

- If you average over all possibilities, no technique does better than any other. We just have information for our training data and the target function between points can be anything, then we have no way of knowing what it is. Have to impose assumptions and restrictions, say smoothness assumptions on top of other things.
- Do well if the real target function can be well estimated by a function in your restricted class of functions $F^*(x)$
- Each method has a situation where it works best. A **situation** is constituted by its real target function (unknown), training sample size (some methods have universality like NN or KNN methods as $n \to \infty$), signal / noise ratio (how much ε compared to F, ie. s2n = $\frac{V(F(x))}{V(\varepsilon)}$).
- With large noise, we need a more restricted procedure, while a NN might be great for high signal situation. Sample size is antidote to noise.
- How do we choose? Try a number of them and focus on one performing best through CV. Try a committee - blend a variety of methods together, though of course won't work always.

Comparison Caution

- Empirical performance comparisons between methods should be taken with skepticism. Theory often lags new methods, so this is a common argument
- There are no universal methods, consider the situations instead.
- Selection biases - examples are selected to show the chosen method is best. Many situations may have been tried before they settled on this one, a type of overfitting. Paper selection effect - we are estimating how this performs, and it may vary but we only see papers when the method performs best
- Expert bias - the person performing the analysis is often more important than the method used. The more familiar you are with a technique, the more likely you can be powerful with it than another one. For a new procedure, the author has best idea for how to make it perform without incentive to tune the competitive methods as well.

Machine Learning

- Every ML algorithm has 3 components
    - Model or pattern structure - the underlying functional form. Define a class of functions indexed by parameters, one of which we will use for our predictions.
    - Score function - judges the quality of fit. We then try to maximize or min the score. The risk is the "magic" score that is the population score. We work with an approximation for a sample when we cannot derive this from first principles. Average loss function over the training data is a natural approximation, $\frac{1}{N}\sum_{i=1}^{N} L\left(y_i, F\left(\underline{x}_i\right)\right)$. But we almost always use another loss criterion, a close convex criterion that is easier to optimize.
    - Search strategy - find the function in that class that optimizes the score. Matrix algebra, quadratic programming, etc. Often have non-convex criteria and a direct solution is not possible - NN, clustering, trees. Heuristic search strategies introduce computational problems but also statistical difficulties. Increases the variance - with multiple minima, our sample matters quite a bit to the minimum we settle in as well as the starting point. Small changes in the starting point can make big changes to the solution I get.
- Searching in a function class that does not contain the true function - irreducible error or bias. In principle we could find the closest function in the class to the true function over the joint distribution. Instead we have a sample, and different samples will give us different scores. With a small function class, the bias can grow, since bigger classes are likely to be closer to the true function. But the bigger the function class, the harder it is to find the best function in that class - go to 0 bias, infinite variance.
- The biggest intellectual content now lives in the search strategy. It also drives our selection of the other two pieces - what is computationally feasible can select our score and model.
- In one dimension, a smoothness restriction might be a good enough class restriction. But in more dimensions we usually need to narrow our class further.
- The score criterion is the part that statistics deals with - it brings in the data. We need a training sample and to estimate the population score.

Data Basics

- Set of measurements each made on a set of N objects. X_{ij} is the jth attribute on the ith object. This is matrix data, attribute-value data.
- Also can have proximity data - similarity or dissimilarity measure between objects, eg. networks, clustering.
- The objects are observations and the measurements are variables. N = # of observations, n = # of variables.

- Predictor variables can take on many types
    - Interval scale variables - signed real numbers. They have (1) logically valued order relation and (2) real values continuous distance function. The most informative of all variable types.
    - Periodic / circular variables - direction, time of day, angle, longitude, months. No order relation, but there is a distance relation. Shortest distance around the circle that defines the period.
    - Ordinal variables - grades, size, performance, ratings. Order relation but no explicit distance function. Ranks are a subclass of ordinal variables - most informative ordinal, since as many distinct values as observations. Binary also a subclass, least informative ordinal but also the most flexible.
    - Categorical / nominal / factorial variables - words, places, languages, etc. Least informative of all variables types - no order relation and distance function can only be 1 or 0, same or different.
- Sometimes the context matters - color could be categorical for perception but interval scale in physics.
- Many procedures are designed specifically for interval scale variables - power weakened for other types, especially categorical. We impute distances between categories that do not reflect reality.
- Decision trees assume ordinal or categorical variables. Uses only the order relation, and for interval scale ignores the distances (loss of power).
- Rule induction - treat all variables as categorical, ignoring distance and order. Lose a lot of information with this procedure for non-categorical predictors.
- Dummy encodings introduce correlations that don't exist - they have to sum to 1, so have pure collinearity. If one dummy is 1 then the rest are 0 - knowing one determines the other values.
- Contrasts - K-1 linear combinations of dummy variables - gets rid of the singularity introduced by using K.
- Outcome variable types
    - Interval scale -> regression problems.
    - Periodic -> periodic regression.
    - Ordinal outcomes -> ordinal regression. A separate literature in statistics
    - Binary -> binary classification / regression
    - Categorical -> multinomial regression or multi-class classification.
- Variable name type: When var name is categorical -> unorganized data. Organized data - where the name of the variable itself conveys information, eg time series, mass spectrum, image pixel values - the location, time, etc conveys information itself in addition to the value associated with it. This class deals with unorganized data.

Trees

- Decision trees can tell you exactly how it arrived at its decision - the opposite of a black box.
- Structural model: linear combination $F(\underline{x}) = \sum_{m=1}^{M} c_m I\left(\underline{x} \in R_m\right)$, breaking up regions into spaces. There is a coefficient c for each region, if x in that region it gets valued with the constant c.
- Score criterion - looking for the regions and coefficients st we minimize the squared error from the true y values - least squares.
- Search strategy: given a set of regions, getting the coefficients is trivial. We can simply use the regions in a linear regression, a simple least squares problem (see score criterion). But optimizing wrt regions is a very difficult problem - therefore we will have to limit the class of regions we can look for, this is limiting our function class.
- Restrictions
    - We restrict the regions to be disjoint and the regions must cover the input space $\implies$ the regions are a partition of the input space. This then means that each x can be in only a single region. When a new x

observation is entered, it will certainly be in a certain region and we can make a prediction for the new x.
    - Simple regions - take the jth predictor variable, set of all values that x_j can take on. Even if x_1, x_2 can take on a smaller range of values jointly than the square that bounds them, we restrict our regions to those rectangles.
- For categorical values, harder to define a subset of values in a range since no order relation. Then must explicitly delineate the subset of values, eg $s_j = \{a, b, c\}$
- Even a simplified problem is NP hard. We instead approximate with a greedy recursive partitioning. Initialize with full space S and the first region value is the global mean of y. At the mth iteration $F_M(\underline{x}) = \sum_{m=1}^{M} \bar{y}_m I(\underline{x} \in R_m)$. We choose one of the regions and partition it into two child regions, giving us $F_{M+1}(\underline{x}) = \sum_{m=1}^{M+1} \bar{y}_m I(\underline{x} \in R_m)$ and we can again choose a region to split, now also considering the child regions.
- Have to decide both which region to split next and where to split it. Since the regions are disjoint, we can look at the best split within a given region without worrying about the global space. Ultimately, we find our next region derived as $m^* = \text{argmax}_{1 \le m \le M} \frac{N_m^{(l)} N_m^{(r)}}{N_m} \left(\bar{y}_m^{(l)} - \bar{y}_m^{(r)} \right)^2$. The fraction makes sure we partition such that we divide the number of observations more equally to make the split mean something.
- Where to split region, we look at just the data in our considered region, and consider a subset of values within our region along a given predictor. Consider a split at these values and consider how it affects our squared error loss. I'll pass over all regions, within a region pass over all variables, consider all possible splits and pick the region (m^*), variable (j^*), and split point ($t_{j^* m^*}$) that is optimal among our choices.
- If our predictor is orderable numeric, can perform an exhaustive search for every value since finite. If our predictor is interval scale, we don't have to consider infinite values, only points that would change where a datapoint would fall in region. But our choice could still affect future data, so we simply choose the middle of the two datapoints since we have no information about this future data.
- Categorical variables - no order relation. We need to consider all combinations of splitting our categories into left and right. This grows exponentially with the number of levels. We have a trick approximation instead: consider the average y for each of the levels of a given categorical variable, then rank them on the means. We can then treat the variable as ordinal and run the algorithm normally.
- Sensitivity: probability of predicting disease given true state is disease. Recall, true positive rate. Number of true predicted positives over all ground truth positives: $\frac{\text{TP}}{\text{TP+FN}}$
- Specificity: probability of predicting non-disease given true state is non-disease. True negative rate = $\frac{\text{TN}}{\text{TN+FP}}$
- Precision: $\frac{\text{TP}}{\text{TP+FP}}$. Fraction of retrieved documents that are relevant to the query

Tree Representation

- Split from parent region to children region corresponds to a parent node into children nodes
- Builds a binary tree that contains the same information as the regions.
- With each internal node, we have a split, the subset of the associated values telling you to go the left or right. With categorical values we need to know which subset of values goes to each child node.
- Get a final set of regions, where all X's in a region share a predicted Y value, the mean of the associated Y values in the region.
- We don't have to tell trees what variables are important - it determines on its own which are important. If it is in the tree with our constraints, it is among the important variables. The tree also allows us to take an X and walk through the logic that leads to its eventual prediction value. Decision trees -> $\hat{F}(x) = \sum_{m=1}^{M} \bar{y}_m I(x \in R_m)$
- Decision trees give complete information for how a decision was determined. Could tell someone exactly why

they were rejected for a loan say.
- Trees are a universal estimator - with enough data, it can approximate any underlying function.

Missing Predictor Values

- If various predictor values are missing, trees have a simple way to handle them.
- Would like the quality of the model to degrade gently as the number of missing values increases, not totally collapse given a trained model. We additionally would like to build trees using training sets with missing values
- Say we are in a region and some values of the predictor we are looking at are missing. We can perform surrogate splits as a local imputation
    - Take X_j vs X_k - say these predictors are highly correlated. If we have a certain split for X_j from the model, the observed values go left or right (assign 1 or 0). Then for values of X_k, we essentially want to build a predictor that guesses whether the X_j label would be 1 or 0. Then when values of X_j are missing, we can use our predicted value from the observation's X_k value.
    - Once we find that we have a missing value for a split, we pass over all the other variables for which we have values and use the one with the highest correlation with our missing predictor. This is the first surrogate. If the first surrogate is missing, we continue to the second surrogate, etc.
- Keep in mind this is being done locally. Once we have done some splitting and we are looking at correlations among segments of the predictors, we may find associations that we did not see before. A loopy convex curve could become linear once we have performed some splits.
- Sometimes missing could be informative - extreme ends of income spectrum might be missing, not at random. This procedure does not take this into account, we would want another procedure if that matters to us.

Right Sized Trees

- Recall we are working with piecewise constant predictions. If working with a diagonal line, could make huge errors at the boundaries, could be fixed with more regions but eventually run out of data before we could split all variables.
- If we have too few regions, we have representational bias - our function is not represented by a piecewise constant. If we have too many splits, we are in danger of overfitting. If $y = f^*(X) + \varepsilon$, we train on y since this is what is observed. This means the ε's are included in training, so if we train too far, we fit to the idiosyncratic error that will not generalize to other data.
- The index controlling the size of the function class for trees is the number of regions. The larger we expand our function class (regions) the more susceptible to noise we are. Adjusting the size of the function space is regularization generally, and here our tuning parameter is # of regions (# of terminal nodes). The bigger the function class, the harder it is to find the best function approximation within it - this is the bias variance tradeoff.
- Two region models are less flexible than 3 region models. The space of 3 region models contains all 2 region models
- Prediction risk: $risk \sim bias^2 + variance$. As tree gets larger bias down, variance up, and there is an optimal number of regions for a particular problem. We want to estimate that.
- Could simple bound the number of regions which would bound our variance. But some areas will need more splits in areas where function changes a lot, while others do not. We cannot address this with this strategy.

- Stopping rules: could determine if next split is not worthwhile. There is always some improvement with our next split (even if just due to ε). Let $\hat{e}_m = \frac{1}{N}\sum_{\underline{x}_i \in R_m}(y_i - \bar{y}_m)^2$ be the contribution of region m to the error from the truth. Then the improvement of another split to the squared error risk is $\hat{I}_m = \hat{e}_m - \hat{e}_{mL} - \hat{e}_{mR}$ which will always be positive - optimistically biased estimate of I_m. We could threshold this value, but there may be later splits that are actually quite significant that we would miss. This can be seen with a symmetric interaction effect between two predictors - the estimated Y value is the same on both sides with one split and leave error, but 2 splits would leave no error.
    - $\sum_{m=1}^{M}\hat{e}_m$ = total risk of the tree = total loss over the entire tree
    - There will always be improvement from another split due to the noise. We then would be fitting our model quite closely to idiosyncratic noise.
    - The threshold $\hat{I}_m \geq k$ is not a sufficient condition for optimal splits.
- Look ahead one level: Then $\hat{I}_m = \hat{e}_m - \hat{e}_{m+1} - \hat{e}_{m+3} - \hat{e}_{m+4}$ where m + {1,3,4} are the terminal nodes after 2 splits, with threshold $\hat{I}_m > 2k$, then accept the two splits else make m terminal. If risk from those child terminal nodes is less than $\hat{e}_m$ then we may exceed our threshold.
- Full look ahead: Let $\hat{r}_m \equiv \hat{e}_m$ then for terminal m $C_m = \hat{r}_m + k$ else $C_m = \sum_{m' \in m} C_{m'}$ for terminal descendents. We charge a cost k for each terminal node. Make choice to minimize C_m. Termination rule can then be if $\hat{r}_m + k \leq C_{m_L} + C_{m_R}$ then make m terminal and set $C_m = \hat{r}_m + k$. Else accept the split and set $C_m = C_{m_L} + C_{m_R}$
    - The point is to observe this rule, we grow the largest possible tree and then apply this rule to determine the number of terminal nodes. We end up applying the rule in inverse order of depth.
    - In practice, split until we are left with identical Y's in terminal nodes (then can split no more) - maximal variance tree. Look at the deepest split, and look at the contribution to risk from the two children compared to the parent. If the sum of the child risks + 2k less than the parent risk + k, then keep the split.
    - Since we are splitting all the way down, we can implement it with recursion - we continue to go left until no more steps, backtrack then continue. In practice often only split until a certain min number of observations in a terminal node but this is a computational approximation - we can split until all nodes are pure.
- Choosing k - the bigger k, the smaller tree we will have, since it is the cost charged for each terminal node.
- All this is equivalent to an optimization problem, **cost complexity pruning**: T = set of all possible tree arbitrarily terminated. $t \in T$ and $|t|$ = number of terminal nodes = M. Our job is to select one of those trees. Then $\hat{r}(t) = \sum_{m=1}^{|t|}\hat{r}_m$ = empirical risk for tree t. Then we seek to find $t_k^* = \min_{t \in \mathcal{T}}[\hat{r}(t) + k|t|]$ for k = complexity parameter (penalty). Regularizing on k, not exactly the same as the number of terminal nodes.
    - If we have two values of k, then the size of the tree for the larger k value will be less than or equal to the tree formed from the smaller k: $k' > k \Rightarrow \left|T_{k'}^*\right| \leqslant \left|T_k^*\right|$
    - Starting with k=0 and raising k. At some point, raising k reaches a point where 2k overwhelms k for the parent no matter what the comparitive risk is. Then the node collapses. We can keep raising k to collapse more nodes. We get a nested seqnece of trees indexed by k. Penalizes for increased variance associated with more complex model $\operatorname{Var}\hat{y} \sim |T|\sigma^2/\mathrm{N}$
    - Instead of exhaustive tree search, we are left with a sequence. Still left with how much to charge for each node k, but this will depend on the signal to noise ratio.
    - In high dimensional space, we really cannot just look and see whether we are fitting a signal better or noise better like we can in 2D. Instead we rely on CV.
- With small sample size, often introduce too much bias with a validation set. Then can turn to k-fold CV.

Classification

- Outcome y takes on 1 of k non-orderable values, simply names.

- Structural model will be quite similar to regression - partition space into regions, all observations in a region receive the same prediction. $\hat{c}(\underline{x}) = \sum_{m=1}^{M} \hat{c}_m I(\underline{x} \in R_m)$
- Score criterion: prediction risk is still the master score criterion, $E_{y\underline{x}} L(y, \hat{c}(\underline{x}))$, expected misclassification cost. The difference here is that L is a matrix, K x K, the 2,7 entry is the loss when you classify as 2 and it is really a 7. L has 0's along diagonal and off diagonal elements provide the misclassification costs. In two class problem $\begin{bmatrix} 0 & c_1 \\ c_2 & 0 \end{bmatrix}$
  - Can use misclassification error, treat all misclassifications the same. This is rarely true in real life - the costs for some misclassifications are much higher than others.
  - Data score criterion, $\hat{r} = \frac{1}{N} \sum_{i=1}^{N} L(y_i, c(\underline{x}_i))$ natural score criterion
- Search strategy - just like in regression, consider how a partition improves the prediction risk, recursive splitting + pruning.
- Again, the critical problem is finding a good set of regions, then assigning the class is trivial.
- For categorical X's, recall we need to use value subsets instead of split points.
- For entire tree $\hat{r}_M = \sum_{m=1}^{M} \hat{p}_m \hat{r}_m$ the estimated risk is the sum over regions of the probability of being in that region p times the estimated risk for that region, gives expected loss for that tree. $\hat{p}_m = N_m / N$ and $\hat{r}_m = \frac{1}{N_m} \sum_{\underline{x}_i \in R_m} L(y_i, \hat{c}_m)$, which is the risk given $\underline{x} \in R_m$.
- Improvement then is $I(j_m, s_{jm}) = \hat{p}_m \hat{r}_m - \hat{p}_{lm} \hat{r}_{lm} - \hat{p}_{rm} \hat{r}_{rm}$. Seems like we have everything we need, but this works terribly!
  - If I do a split into two children, I cannot improve my prediction risk if the children have the same predicted class with 0-1 loss.
  - Example dataset has majority class 1's for every split, even though 3 splits could get us 0 error rate. But when looking for a first split, we cannot find a split where the children nodes will have different predictions, since class 1 is majority on both sides.
  - Misclassification risk is not a continuous function of model parameters. We can only do combinatorial optimization, for larger trees this is intractable
  - Each split minimizes estimated risk assuming it is the final split, does not account for better partitioning in future splits. Bad greedy strategy
- If we use a differentiable surrogate criterion, we can still use our greedy strategy.
- Define the loss $\text{loss}(y = c_l, c(\underline{x}) = k) = \sum_{l=1}^{K} L_{lk} I(y = c_l | \underline{x}) = L_{lk}$ for when we predict k and the true class is c_l.
  - Risk (the expected loss) in predicting $c(x) = c_k$ when truth is $y = c_l$ is $r_k(x) = EL(y = c_l, c(x) = c_k) = \sum_{l=1}^{K} L_{lk} EI(y = c_l | \underline{x}) = \sum_{l=1}^{K} L_{lk} Pr(y = c_l | \underline{x})$ - the expected value of an indicator of an event is its probability that the event happens. We are summing over the classes 1 through k. If I classify as class k, the loss is the sum over the classes of the L_{lk} times the probability that it really is an ℓ. If I know the probability of y taking on each of its values at x, I can compute the optimal risk - this is the **Bayes** optimal decision rule.
- We do not know these probabilities, but we will estimate them as $\hat{p}_l(x)$. This changes our categorical problem to a interval scale problem. Then using asymptotic arguments can justify the use of our estimates. Our probabilities will be off with finite data, but our tree can still be optimal if the probability orderings, and therefore decision boundaries, are the same.
- We redefine our population score criterion: $E_{y\underline{x}} L(y, c(\underline{x})) \leftarrow E_{y\underline{x}} J(y, \{\hat{p}_k(\underline{x})\}_1^k)$. Now we need an empirical score criterion that is differentiable and matches this expectation

- Squared error loss is a candidate, where we estimate k numeric target functions, one for each class. Estimating the probability of each class at each value of x. We now have k variables we are trying to predict that are numeric, instead of 1 variable to predict that has k distinct values. We are using the probabilities as a device to do a minimum risk classification, but often the probabilities themselves are useful.
- Given a region, we simply estimate the probabilities by counting the number of observations in each class in the region: $\hat{p}_{km} = \frac{N_{km}}{N_m}$. However we do not just classify to the highest probability - the loss also depends on L_{lk} for each comparison of class, if some misclassification are much more costly, we still might predict a lower probability class.
- When we plug in, the squared error risk reduces to $\frac{N_m}{N} \sum_{k=1}^{K} \hat{p}_{km} \left(1 - \hat{p}_{km}\right)$, which is the Gini index of diversity.
    - Max diversity (min purity) when all classes equally probable, $G = 1 - \frac{1}{k}$.
    - Min diversity (max purity) when all 1 class, $G = 0$
- If the node is pure, it contributes nothing to the risk of the tree. If the node is not pure, it contributes a positive amount to the risk. Therefore our objective becomes making pure nodes. The improvement from a split becomes
  $\hat{I}_m(j, s_{jm}) = \hat{e}_m - \hat{e}_{ml} - \hat{e}_{mr} = \hat{P}\left(\underline{x} \in R_m\right) \text{ Gini } \left(R_m\right) - \hat{P}\left(\underline{x} \in R_m^{(\ell)}\right) \text{ Gini } \left(R_m^{(\ell)}\right) - \hat{P}\left(\underline{x} \in R_m^{(r)}\right) \text{ Gini } \left(R_m^{(r)}\right)$
- This is differentiable - we are no longer trying to classify better, we are trying to purify. Splits that improve the confidence of your classification will be made, even if it does nothing to improve the misclassification error.
- Entropy: $H = -\sum_{k=1}^{K} P_k \log p_k$ also a fine criterion / purity measure, similar to Gini.

Tree Advantages

- Relatively fast, can use all types of predictor variables - numeric, binary, categorical, missing values
- Invariant under monotone transforms of the predictor variables - building a piecewise constant model so order matters but scale does not.
    - Thus we do not have the issue of choosing transformations, simplifying the search process (eg in regression, we can make a lot of transformations)
    - Immunity to outliers in predictors. As long as the observations remain in the same order, the decision boundary is the same. Especially important for outliers that are typical in a single predictor dimension but are odd in 2 predictor dimensions or more - we cannot see them and we do not have to worry when using trees
    - Scales are irrelevant, allowing us to pull in data from different sources without worrying about rescaling
- Note we are not talking about outliers / transformation in y - those can matter since that determines the predicted value assigned
- Resistence to noise variables - we saw lasso, SVM etc degrade with high number of noise variables. Trees really do not require variable selection - the tree itself tells you what is important. Bet on sparsity principle: “Use a procedure that does well in sparse problems, since no procedure does well in dense problems.” Some people use trees for variable selection and place those into another model
- Few tunable parameters - essentially off the shelf. Interpretable model representation.
- No problems with highly correlated variables. The identifiability problem. The tree will just pick one a move on, so may hurt interpretation a bit but the predictive model should be fine.

Tree Disadvantages

- Inaccuracy - piecewise constant approximation can lead to big bias. Think of the linear target function, requires many splits to get arbitrarily close, but in practice you do not get that many splits on each variable. So piecewise constant functions do not always map well to the target - more bias.
- Hyper rectangular regions - oscillating overshooting and undershooting a linear function - bias issue
- Data fragmentation - each split reduced the training data in a subregion. Run out of data pretty quickly. Cannot model local dependency on more than a few variables, not good for target functions that have dependencies on many variables. Again another bias problem.
- Variance problems - we have to find a function in our function class and to do that we use data. Trees have very high variance caused by its greedy search strategy that results in a local optimum. $\hat{f}(x) = c\prod I(x_{j(l)} \in s_{jl})$ - we are multiplying our errors, causing the to get much larger! Small changes in data cause big changes in the tree - unstable, high variance, high error.
- Look to bagging, boosting, and MARS to solve these issues.

Bagging and Random Forests

Bagging

- Goal: improve performance of unstable procedures by stabilizing them. Ie, high variance procedures with multiple optima
- Given a convex criterion, our sample may have different minima. We get a distribution of solutions over samples, and this is our variance. Suppose instead we have a non-convex multiple minima criterion - this is our situation with trees, NNs. The solution we end up in will depend on where we start, which is not true in our convex case. Procedures that are convex in their parameters are much more stable than those that are not.
- We have some $\hat{F}(x) = argmin\frac{1}{N}\sum_{i=1}^{N} L\left(y_i, F(\underline{x}_i)\right)$. Unstable procedure - small change in training data, big change in F hat.
- Bagging repeatedly makes small changes in data and averages the results. The average should be much more stable
- Now have $\hat{F}_b(x)$ for each iteration of our model fit to perturbed data. Then our bagged estimate is $\hat{F}_B(\underline{x}) = \frac{1}{B}\sum_{b=1}^{B}\hat{F}_b(\underline{x})$. Typically we do a bootstrap sampling procedure to get perturbed data, but the procedure is not especially important.
- Here we aren't using the bootstrap for its original purpose of estimating population parameters. Just a convenient way to shake up the data, so other procedures could be just as effective. Dropout for NN's is equivalent to bagging - sampling from your parameters, averaging many solutions as the training goes along. We don't bag NNs since they take so long to train, use a concurrent regularization while training instead.
- Notice I have not changed my function space - bagging is pulling functions out of the same class. We do not reduce bias then, only change the variance. Why then not make the function class as big as we can, if we can reduce the variance with bagging? No reason - we do increase our function class and throw out pruning our pre-bagged trees!

Random Forests

- Bagged trees where we randomize the available predictors to split over for a given tree fit.
- Typically restrict to $\sqrt{p}$ where p is the number of predictors. Not especially chosen for theory but helps with computation.
- Increasing the randomization increases our variance control, but we pay with some bias.
- Bootstrap samples have a bias variance tradeoff - fitting on a portion of the dataset available introduces bias but we gain more from variance reduction. Fitting on the whole dataset lowers bias but we cannot control the variance.

- Work much better for classification than regression. This is true for trees in general, since classification is almost a piecewise constant function anyway.

Boosting

Boosting

- You can boost any procedure, but trees are especially improved by boosting.
- Let $\{y_i, z_i\}_1^N$ be our outcome and predictor variables, $\hat{F}(\underline{z}) = \text{argmin}_1 \, S\left(\{y_i, F\left(\underline{z}_i\right)\}_1^N\right)$ is our function approx in a function class, and S is our score criterion
- For trees, $F(z) = T(z) = \sum_{m=1}^M c_m I\left(z \in R_m\right)$. Then our boosted model is $F(z) = \sum_{j=1}^J a_j T_j(\underline{z})$, a linear model where we are finding the weights a for each tree - it is a linear regression problem. This thing that defines our function in this space are the coefficients - we are considering all possible trees in our tree class (in principle).
- We are not expanding our tree class like in bagging, we are weighting the models in our given class.
- Then our population optimization is $\left\{a_j^*\right\}_1^M = \underset{\{a_j\}_1^M}{\text{argmin}} E_{yz} L(y, \sum_{j=1}^J a_j T_j(\underline{z}))$ and on our training data approximated by $\hat{F}(\underline{z}) = \arg\min_{\{a_j\}_1^M} \frac{1}{N} \sum_{i=1}^N L\left(y_i, \sum_{j=1}^J a_j T_j\left(\underline{z}_i\right)\right)$
- Treat trees as fixed, weights are the parameters to solve for. So we can let $X_j = T_j(z),\ X_{ij} = T_j(z_i) \implies F(\underline{x}; \{a_j^*\}_1^J) = \sum_{j=1}^n a_j^* x_j$.
- This parameter space is way too large - need to regularize to solve this optimization. We are essentially fitting a regularized linear regression. If N >> n, we will end up with a high variance answer - $\hat{R}$ is random and the optimization $\hat{\mathbf{a}} = \arg\min_{\mathbf{a}} \hat{R}(\mathbf{a})$ will vary widely.

Regularization

- $\hat{R}(\mathbf{a}) = \frac{1}{N} \sum_{i=1}^N L\left(y_i, a_0 + \sum_{j=1}^n a_j x_{ij}\right)$ is the average loss over the data - empirical risk. With regularization we minimize this with a constraint $P(\mathrm{a}) \leq t$. Note $t \geq P(\hat{\mathbf{a}})$ imposes no constraint and max variance, t = 0 is max constraint and we have max bias (all coefs are 0)
- Define equivalent optimization problem $\hat{\mathbf{a}}(\lambda) = \arg\min_{\mathbf{a}}[\hat{R}(\mathbf{a}) + \lambda \cdot P(\mathbf{a})]$. For a given data set, we have a fixed risk for a set of coefficients and the solution will change only with lambda. Given this set up, our set of possible solutions follows a 1 dimensional path $\in S^{n+1}$. We can then examine solutions along the path to find the best point - the one that minimizes the prediction risk.
- We cannot find the solution in the entire space of a, but our path restriction makes this problem tractable. This is different from a Lasso for example.
- If we were optimizing on finite data, lambda would be 0 - we obviously need to use CV. Construct the path using a subset of the data and use the left out data to approximate the predicted risk along the path.
- Make a grid of lambda values. For each lambda value we will solve our optimization problem. That gives us a set of coefficients, and we use the model given these coefficients to predict the left out data. That's how we choose lambda, but we still need a penalty function p.
- We say $a^* =$ point in S^{n+1}. Different penalties will produce different paths in a space. Want to use the penalty that brings us closest to the actual optimal solution target point. We then need to know something about the properties of the true solution. What is this property? Sparsity
- Sparsity: the fraction of non-influential variables. Even if we have measured 10k variables, assume only a few are actually influencing the outcome. We don't know which but we assume our solutions are sparse. In the case of trees, we pretty much need to assume sparse solutions - there are very few tress in our huge function class

that will have good predictions on our outcome. Bet on sparsity principle.

- There are no true zero valued coefficients - with infinite data, each coefficient would have some weight, but we are forcing some to 0 for sparse solutions.
- Choose $P(a)$ that induces sparse paths - one that hugs one of the axes in a space (say a has two dimensions). Keeps one axis near 0 and allows the other to have large influence. In higher space, hugs all axes but a few.
- **Power family**: $P_\gamma(\mathbf{a}) = \sum_{j=1}^n |a_j|^\gamma$ indexed by gamma. This is quite familiar, $\gamma \in (2, 1, 0)$ produces ridge, lasso, and subset respectively going from densest to most sparse. Note we never penalize the intercept, just the slope parameters. With 2 highly correlated predictors, doesn't matter which we use in a prediction regression model - the data does not tell us which is "important", this is not a causal statement, merely the same for prediction's sake given the data. The L2 penalty will average the weight across these variables, while L0 would choose one with full weight and eliminate the other (L1 produces zeros, but generally less than L0).
- For L0, there is no penalty on coefficient size once the coefficient is non-zero. $(\hat{a}_0, \hat{\underline{a}}) = \underset{a_0, \underline{a}}{argmin}\ R(a_0, \underline{a}) + \lambda \sum_{j=1}^m I(|a_j| > 0)$ -when $\lambda = \infty$ all a's set to 0, just intercept. Eventually reducing lambda reduced enough to point where risk reduction from non zero coefficient large enough to overwhelm the penalty. Eventually risk reduction will be greater than 2λ as we reduce the penalty - best 2 variable solution. Can continue to find best n-variable solution. The path will be piecewise - our penalty term is non differentiable and we have discrete jumps changing values of lambda. You get more discontinuities for smaller γ : $\gamma = 0$ is simply most extreme with no continuous sections. Pure combinatoric optimization, and best 3 model may not contain best 2 model eg.
- Lasso is so popular because it is convex and produces sparse solutions - best of both worlds. Even if L1 and L0 pick coefficients in similar order, you will get different coefficient values - one has shrinkage and the other none. Elastic net produces denser solutions than L1 but still sets some coefficients exactly to 0.

Bridge Regression

- We need to solve repeatedly: $\hat{\mathbf{a}}_\beta(\lambda) = \arg\min_{\mathbf{a}} \left[\hat{R}(\mathbf{a}) + \lambda \cdot P_\beta(\mathbf{a})\right]$ for $0 \leq \beta \leq 2, 0 \leq \lambda \leq \infty$
- We look at penalties and point on the path constructed for that penalty that minimizes that criteria. Model selection criteria: $(\hat{\beta}, \hat{\lambda})$
- This is hard to construct with a fast algorithm, and very unreasonable for boosting or non-convex P. We could try ten different penalties and 100 different lambdas for each - this is not feasible.
- There are other ways for path construction: suppose we want $a^* = argmin_{\underline{a}} \hat{R}(\underline{a})$. Could use gradient descent - move towards lowest gradient, recompute, move, etc. We no longer need to solve the actual optimization at every point and could use our gradient path as our optimization path.
- Early stopping - follow some descent path, each step making the risk smaller. Stopping short of the minimum often produces a better result - early stopping is a form of regularization. Before we get to $\lambda = 0$ we find our optimum, but **following the gradient isn't producing sparse solutions** - closer to ridge. What we want is an optimization that would follow our lambda optimizing path, just much faster than the full optimization.
- **Direct path seeking**: rapidly produce path given P(a) such that the path is equivalent to the fully optimized solution.
    - Start with all coefs equal to 0 - solution for $\lambda = \infty$
    - At every step, we compute direction in coefficient space d and move a small amount $\Delta\nu$ in that direction
    - Update our location and repeat. until $\hat{R}$ is minimized
- Examples? Partial least squares, LARS

Generalized Path Seeking

- Generalized path seeking: fast for any convex loss function $L(y, F)$ and any penalty P(a) st $\frac{\partial P(\mathbf{a})}{\partial |a_j|} \geq 0$ - if you hold all the coef values the same and increase the value of one them, you increase the penalty - penalty is monotone increasing in the coefficient size for all coefs.
  - $\nu \geq 0$, point on path, orders the solution. $g_j(\nu) = -\left[\frac{\partial \hat{R}(\mathbf{a})}{\partial a_j}\right]_{\mathbf{a}=\hat{\mathbf{a}}(\nu)}$ - this is the negative gradient of the risk, derivative of the score function wrt each coefficient. $p_j(\nu) = \left[\frac{\partial P(\mathbf{a})}{\partial |a_j|}\right]_{\mathbf{a}=\hat{\mathbf{a}}(\nu)}$ - derivative of the penalty wrt the coefficient solutions.
  - $\lambda_j(\nu) = g_j(\nu)/p_j(\nu)$ - the jth component of the negative gradient divided by the jth component of the penalty at point ν. Lambda for each coefficient at each point on the path.
  - Then our algorithm: start with all coefs equal to zero, $\lambda = \infty$ solution
  - Compute lambdas at point ν using our penalty and gradient derivatives. Identify lambda values that have sign opposite of its corresponding coefficient (assume no sign for coef = 0). If that set is empty (and it usually is) then identify the coefficient j that has the largest value of lambda: $j^* = \arg\max_j |\lambda_j(\nu)|$. Otherwise, just look for this j in our opposite signed set.
  - Now we ID'd a single coef. Increment this coef by its estimated value $\hat{a}_{j^*}(\nu)$ at ν plus an increment $\Delta\nu$ times the sign of its $\lambda_{j^*}(\nu)$. Hold all other coefs to the same value. This gives us the next point on the path - $\nu \leftarrow \nu + \Delta\nu$
  - Repeat until $\lambda(\nu) = 0$.
- In a normal steepest descent, I would compute the gradient and move in that direction. Here taking a gradient ratio (if we are using Lasso penalty, then the penalty derivative is 1 for all coefs and we are actually using the gradient). And unlike normal steepest descent where we move in the gradient direction across covariates, here **we move in the direction of a single predictor at a time**, holding the others to their current value at a step. Picking the largest component of the gradient and only moving that direction. We do this bc it's not the destination that counts, it is the journey - the path is what matters to us. This path follows the optimized lambda path very closely.
- Why do we expect this to follow closely the exact path? Say $\hat{\mathbf{a}}(\lambda)$ is the exact path and $\hat{\mathbf{a}}(\nu)$ is the GPS path - if the path as a function of lambda are monotone, then this algorithm produces the exact path. If as we relax lambda, the coefs either stay the same or increase, we get the exact optimized path. If not montone, at the point the coef begins to decrease (notice now the lambda value is the opposite of the sign of the coefficient since it is dragging it back towards zero), the GPS algorithms will stay constant for some period before decreasing and rejoining the exact path - creating an error distance between paths wherever the first derivative changes signs.
- The opposing lambda and coefficient signs - allows catch up to non-monotonic paths to be much faster. Since lambda denominator $\frac{\partial P(\mathbf{a})}{\partial |a_j|} \geq 0$, the sign of lambda determined by the sign of the risk gradient. If the risk is pulling it in the oppositive direction to the sign of the coefficient, then the risk wants the coefficient to change direction. A signal that our path should not be monotonic and we will correct it faster if we pay attention to this signal.

Gradient Boosting

- Define class of weak learners, for us trees. Boosting dramatically improves their performance
- Define a small function class $f(x; b) \in F_B = \{f(\underline{x}, \underline{b})\}_{\underline{b} \in \mathbb{B}}$. Our models will be linear combinations of these functions.
- Sum over all models in the class combined with linear coefficients $F(\underline{x}) = \sum_{j=1}^{J} a_j f\left(\underline{x}; \underline{b}_j\right)$ with the constraint $\frac{1}{N}\sum_{i=1}^{N} f^2\left(\underline{x}_i, \underline{b}\right) = 1$ on training data.

- We are going to need to regularize, so the GPS is a good candidate. Letting $\nu =$ number of steps, where the increment $\Delta\nu$ is a step. The penalty component is easy to calculate, but computing $g_j(\nu)$ is more complicated
- Letting $\hat{R}(\underline{a}) = \frac{1}{N}\sum_{i=1}^{N} L\left(y_i, F_i\right)$ - this is our a_j.
- The derivative of the loss wrt its prediction is called the pseudo response or **generalized residual**: $l_i = -\frac{\partial L}{\partial F_i}(y_i, F_i)$. On the path, this becomes $l_i(\nu) = -\frac{\partial L}{\partial F_i(\nu)}(y_i, F_i(\nu))$. This quantity is fundamentally important to boosting. We want to minimize the risk, or if too hard, a Taylor expansion of the risk. First order $R = R_0 + \frac{\partial R}{\partial F}(F)$ then we are directly using the generalized residual. Alternatively, view through gradient descent in function space: $\sum_{i=1}^{n} L\left(y_i = (\underline{x}_i)\right) = R(F(x))$, want to minimize this in discrete space.
- Generalized residual: our ultimate loss is least-squares. For the squared error loss the generalized residual is the ordinary residual. For absolute value loss, we get the sign of the residual for the generalized residual. For any differentiable loss, we can get some generalized residual. The F in the GR is the prediction based on the K trees already in the model - GR is a function of true y and the current prediction at step K.
- Call $f\left(\underline{x}_i; \underline{b}_j\right)$ the jth base learner. Let $\hat{a}_j(\nu)$ be the coefficient of the jth base learner at ν and $F_i(\nu)$ the prediction for x_i at our path point. The active set of coefficients are the ones that are non zero - they are the only ones that contribute to prediction at our step. We only have to know the coeffients for the base learners in the active set to compute the model, so simplifies our needed computation if many zeros.
- In the general boosting algorithm, we start at the beginning of the path with empty active set and base learner set and $\nu = 0$. We first look over the active coefficients, which should be small compared to total number of coefficients. Compute the lambdas for those - for each $j \in A(\nu)$. In the active set, check which coefs have sign opposite their respective lambdas. If some exist, pick the one with largest lambda, just as in regular GPS. Increment its coefficient by a small amount and repeat. If there were not opposing signs, we pick the coef with the largest lambda in the active set. Then we need to compute the lambdas for all coefs in the inactive set (this is huge), and find the largest lambda. If this is smaller than our largest active set lambda, else we need to add it to the active set.
    - Once a lambda is nonzero, it does not return. Since we take finite incremental steps, even if the coef path is not monotonic, it is too unlikely we would step exactly back on zero - instead pass over it.
- At any point on the path we construct, we have a set of active coefficients. Along that path we check performance on left out set to see where we should stop.
- Now note that computing $\{g_k(\nu)\}_{k \notin A(\nu)}$ to get $k^* = argmax|g_k(\nu)|$ where $g_k(\nu) = \frac{\partial \hat{R}(a)}{\partial a_k}$. There are too many $k \notin A(\nu)$ to do this efficiently so we have to rethink the algorithm.
- Can think of lambda as along the path, every step I am reducing the risk and decreasing the value of the penalty. **The lambda ratio is the reduction in risk for a change in penalty**. Can also look at the ratio through the KKT conditions.
- Our Trick: For each base learner indexed by b, we need the gradient of the risk wrt its corresponding coefficient. So the gradient for base learner b is defined by $g(\underline{b}; v) = \frac{1}{N}\sum_{i=1}^{N} l_i(v) f\left(\underline{x}_i; \underline{b}\right)$. And we want to find $\underline{b}^*(v) = \text{argmax}_b \left|g(\underline{b}; v)\right|$. The base learner amongst all possible with the largest gradient is the one that best fits the generalized residual where best is defined by square error loss: $(\underline{b}^*(v), \rho^*(v)) = \underset{\underline{b}, \rho}{\text{argmin}} \sum_{i=1}^{N} \left(l_i(v) - \rho \cdot f\left(\underline{x}_i; \underline{b}\right)\right)^2$ (The rho is ancillary, not needed for trees necessarily)
    - This is a non linear least squares operation. We can use some optimizer to find our best b. We find the best learner to add by the one that best predicts the generalized residual. This could still be complicated given its non linearity, but is a better problem to solve.
- Lasso Gradient Boosting

    - We choose penalty $P(\underline{a}) = \sum_{j=1}^{J} |a_j|$ and for a given j, $p_j\left(a_j\right) = \frac{\partial P(\underline{q})}{\partial |a_j|} = 1$. We no longer have to care about penalty values since they are 1.
    - We don't bother checking for lambdas opposite the sign of the coefficient. This simplification just means we won't track non-monotone paths as closely, but large efficiency increase.
    - Step size: some strategies simply choose a small number, but this is quite relative and arbitrary. A lot of implementations find the optimal coef for adding a given variable - the value that minimizes the risk. Then scale it back by some small ϵ - say (0.01), this allows us to place the update on the proper scale as determined by the data units.
- Gradient Boosting
    - Compute the generalized residuals at each point
    - Find the base learner that best predicts that GR
    - Find the optimal coef to add the base learner to determine step size $\Delta\nu$; we are looking over all possible line segments along our path using our tree and determining the optimal line segment size that minimizes the loss.
    - Add base learner to the expansion and give it coefficient $\hat{a}_k = \Delta\nu$
    - Repeat until we decide to stop.

Boosting Regression Trees

- Our base learner function class restricted to all M terminal node trees. Now $b = \{c_m\}_1^N$ and split variables or subsets defining regions $\{R_m\}_1^M$
- At each boosting step, find the region set and region assignments (c's) that best predicts the generalized residual by squared error loss. This is a least square regression tree problem, which we solved previously with an approximate CART greedy top down approach. We will use this procedure again, except instead of best predicting the outcome we are predicting the generalized residual - simply treated as an outcome.
- Algorithm: Compute the generalized residuals at each point. Use tree building routine CART with GR as outcome and it returns a region set. Decide on step size $\Delta\nu$ - finding the optimal coefficient by which to add our tree, multiplied by some $\varepsilon \in (0, 1)$, then add our (coef x tree) to the model. This changes the GR and we repeat.
- This is the basic boosting algorithm - we could sub out CART and boost over another method to fit the generalized residual. Our optimal coefficient step ensures our procedure is scale invariant, but not everyone implements this step and sometimes just use a small constant
- Once we have a tree optimized for the generalized residual, have to find a prediction for y still. We optimize a separate update for each node
- In a region R_m, take $argmin_{a_m} \sum_{x_i \in R_m} L(y_i), F_i + a_m)$ where F_i is the current prediction of y up to current iteration. Find the best constant update in our region that minimizes the prediction risk in that region. We get away with that since regions are disjoint. We perform this argmin for each region.
- This makes things much easier, minimize the risk in each region instead of trying to convert the generalized residual to y predictors. Instead of taking the value that the tree has returned, jointly optimize for the update in each region, which gives us the update for the tree over all terminal nodes. If our loss is squared error loss, then the optimal update is the mean of the residuals in each region. OTOH, absolute loss has the optimal update of the median of the residuals in each region. This is the MART procedure.

MART

- Initialize with predictions at 0 for all regions
- Iterate the boosting loop for a large number of times. For each step and point, get generalized residual at each point.

- Now CART returns the regions as fit to the GR. In each region separately, we find the update that minimizes the risk for that region. We find the best constant in each terminal node to add to the current prediction that reduces the risk the most.
- We have to decide when to stop, otherwise will perfectly fit the training data eventually. At every step, use the tree we have so far, predict on the test set to get the test error. We trace a test error curve and can then pick the model that minimizes the test error. (Originally thought it couldn't overfit, because mostly used for classification and misclassification risk is so crude that changes to the model did little to the error rate. But look at the estimated probabilities instead of predicted classes and we would see overfitting.)
- We can see boosting is taking an output of a not very good model, operating on the output instead of the internals of the model to make it more powerful.
- Each tree is a sum $\sum_{m=1}^{M} c_m I\left(\underline{x} \in \mathbb{R}_m\right)$ and we want to find a coefficient to multiply this tree by α to scale it in the model. But because of the specific tree structure, a linear combination, we can say $a_m = \alpha c_m$ and optimize directly the combined coefficient.
- Want to make learning rate as small as possible, paying the price of many more trees as the learning rate declines. Will have a sequence of many similar trees. Learning rate typically not cross validated. The final result is a boosted piecewise constant, and the smaller the learning rate, the smoother the approximation.
- Two biggest points: If we define the generalized residual as $\ell(y, F) = -\frac{\partial L(y,F)}{\partial F}$, when we build the tree, we **pick regions** $\{R_m\}_1^M = CART(\{\ell(y_i, F_i)\}, x_i)$, keeping just the regions and not caring about the constants predicted, since these are residual predictions. Then within each region, the **update** is given by $\sum_{x_i \in R_m} \ell\left(y_1, F_i + a_m\right) = 0$ - the reason for this is that the constant update that minimizes the risk in a region is the same as setting its derivative to 0.

Generalizing Loss Functions

- Instead of squared error loss, could use absolute loss. Since the minimizer is the median of y given x, it is absolutely robust against outliers in y. Trees themselves are already immune to outliers in x so MART with median is totally immune to outliers in predictors and response. This is quite useful for data mining, early portions when you might not have discovered everything about your dataset.
- The generalized residual is the $l_i = \tilde{y}_i = \text{sign}(y_i - F_{m-1}\left(\underline{x}_i\right))$ - so as long as we haven't changed the sign of the residual the solution won't change.
- However, if your residuals are well behaved, say an additive error model $y = F^*(x) + \varepsilon,\ \varepsilon \sim N(0, \sigma^2)$ the squared error loss is simply the best you can do. Using absolute value loss will incur error - 60% efficiency. But if you don't have this, fatter tails, exponential loss, etc, then absolute loss can be quite a bit better.
- Huber M-Loss: $L(y, F) = M(y, F) = \begin{cases} 1/2(y-F)^2 & |y-F| \le \delta \\ \delta(|y-F| - \delta/2) & |y-F| > \delta \end{cases}$ where δ is a trimming factor defining outliers. Squared error loss is more sensitive to very extreme positions (look at its slope) which is the opposite of what we want. Instead Huber takes a convex loss for small residuals and absolute loss for large residuals. This is a very good loss for regression to get the best of both worlds. To determine δ, could look at sorted residuals and see where you might want to cut off, but boosting will change the distribution of the residuals at each step. Better to make it proportional to residuals, need to adjust along the way.

Boosting Classification Trees

- Let $d_k = I(y = c_k)$ - dummy indicator variable for y in a given class. We are going to try to predict the probabilities that y takes on each class value for a given x, just as we did for single trees.
- Then the prediction risk is the sum of class pair-specific losses times the estimated probabilities, this is what we minimize
- So we let $d = \{d_k\}_1^K$, indicators of being in each class, and the Gini criterion could be a loss we use such that $L(d, p) = \sum_{k=1}^{K} \left(d_k - \hat{p}_k(x)\right)^2$ - we use this in CART because of rapid computation. But in boosting we can use

any loss (the tree will use squared error)

- Let $d_k \in \{0, 1\}$ a multinomial RV - then we use negative multinomial log likelihood (just like when we have a binomial outcome). $L(\underline{d}, \underline{\hat{p}}(\underline{x})) = -\sum_{k=1}^{K} d_k \log \hat{p}_k(\underline{x})$. This is nice, but must have probabilities between 0 and 1 and the probabilities need to sum to 1 - not going to easy in a tree expansion.
- Instead, express each probability as a ratio of some function F_k for each class - these can take on any value and we ensure that the ratio sums to one across all probabilities: $\hat{p}_k(x) = exp(F_k(x)) / \sum_{l=1}^{K} exp(F_l(x))$
- We can rewrite the loss criterion in terms of the F's - $L(d, F(x)) = -\sum_{k=1}^{K} d_k F_k(\underline{x}) + \log \sum_{k=1}^{K} e^{F_k(x)}$. We have a single loss function but based on k predictions. Have to generalized the GR to take into account multiple functions
- The new GR $\ell_{ik} = -\left[\frac{\partial L(\underline{d}_i, F(\underline{x}_i))}{\partial F_k(\underline{x}_i)}\right]_{F(x)=F_{m-1}(x)} == d_{ik} - \hat{p}_k(x)$. K is the number of classes, take partial of L wrt each F. Turns out this is the simple residual on the probability scale - much easier.
- The one step update turns out to be the generalized residual

K-MART

- We are approximating the F's, one for each class. Start with all F's equal to 0
- For M trees in each class expansion (for total of KM), loop M times. Compute probabilities from the F's. Then pass over the classes, compute GR for that class (ordinary resid on prob scale, estimated from first step). Then generate tree predicting GR to get regions, calculate estimated a, then update the function F_k by taking a step.
- K trees, one for each class. Each p_k depends on all of the F's through the constraint of adding to 1. At every step we are adding a tree for each class and adding it to the F for that class. This in turn updates the probability estimates.

Right Sized Trees for Boosting

- Cannot just build the largest tree - then the residuals would be zero! (with enough data)
- Turns out you want small trees for boosting - this is a hyperparameter of the procedure. XGBoost throws in an additional regularizer
- Functional ANOVA decomposition
  - Take function $F(x) = F(x_1, x_2, \ldots, x_n)$ and we want a special approximation of kind $\approx \sum_{j=1}^{n} f_j(x_j)$. This is the additive (main effect) approximation. The dependence of f on each variable does not depend upon the values of the other variables.
  - If we want a closer approximation, take main effects approximation and add $\approx \sum_{j=1}^{n} f_j(x_j) + \sum_{j,k} f_{jk}(x_j, x_k)$ a sum over all pairs of predictors. This is a bigger function space than the first approx but likely smaller than the true function. This term is called 2 variable interaction effects. This could continue to higher order interaction effects. Going up to all variables would give us the true function.
  - Most target functions we encounter have weak higher order interaction effects. Restricting the main effects only, we can often do extremely well (think of GAMs!).
  - We can look at being in a terminal node as a product of indicators: $I(X_5 \leq s_5) I(X_1 6 \leq s_1 6) I(X_3 \geq s_3) I(X_2 1 \leq s_2 1)$ - this is a fourth order interaction effect. Therefore each level we add to a tree adds another order of interaction.
  - A single split tree (stump) is main effects only model. The question comes down to how high an order to we want to capture, and almost all of the time the answer is very low orders. This is why we use small trees in boosting. This is of course another bias-variance tradeoff.

Interpretation

- Boosting turns trees into a black box model - very hard to interpret models built on residuals
- With trees we had a very easy way of determining variable importance. First, the ones actually included in the tree are the important variables. Second, the splits near the top tend to improve things more than the bottom. Third, the number of times a variable is split over is an indication. If each split has some improvement I_r then the importance of variable J is the sum of the splits of the improvement to the model of that split times an indicator of the variable used for the that split: $\hat{J}_j^2(T) = \sum_{t=1}^{L-1} \hat{I}_t^2 \cdot 1(v_t = j)$.
- With boosted trees, we just average over all trees: $\hat{J}_j^2 = \frac{1}{M} \sum_{m=1}^{M} \hat{J}_j^2(T_m)$. This works better than the formula for a single tree, since it is much more stable.
- Once we know which variables are important, we want to know how our function depends on them. Best way is to make a plot, but for many variables, we cannot make a many dimensional plot.
- Partial Dependence Functions
    - Suppose I have $F(X; a)$ for vector X and a parameters. This defined a function class, possibly large, could we get a function of X that characterizes what functions in this class look like.
    - $\bar{F}_A(x) = \int_{a \in A} F(X; a) p(a) da$ - averaging over all a's, holding X fixed, get a characteristic function. For $F(X) = F(x_1, \ldots, x_m)$ what is the characteristic function of some of these predictors averaging over the others. Just want a characteristic in terms of x_1, x_2. Pretend x_1, x_2 are the variables and $x_3, \ldots$ the parameters a and perform the same process:
      $\bar{F}_{12}(x_1, x_2) = \int_X F(x_1, \ldots, x_m) p(x_3, \ldots, x_m) dx_3 dx_4 \ldots dx_m$. This is the partial dependence of F(X) on x_1, x_2.
- Then I can plot the partial dependence of $\bar{F}_1(x_1)$, say. These are trivial to compute on the data.
- Emprically estimated by $\bar{F}_{12}(x_1, x_2) = \frac{1}{N} \sum_{i=1}^{N} F(x_1, x_2, x_{i3}, \ldots x_{im})$. Plug in the data values for $x_3, \ldots, x_m$ then see how the function estimates the relationship of x_1, x_2.
- We are averaging over the values $x_3, \ldots, x_m$ via an integral:
  $\bar{F}(X_1, X_2) \int F(X_1, X_2, \ldots, x_m | P(x_3, \ldots, x_m)) dx_3, \ldots x_m$.
- Then $\hat{\bar{F}}(X_1, X_2) = \frac{1}{N} \sum_{i=1}^{N} F(X_1, X_2, x_{i3}, \ldots, x_{im})$. Can turn up a heat map or a graph depending if you have 2 or 1 variables that vary.
- For single variable, y axis is F, x axis is the variable. HIgh values on the plot are high dependence for that range of predictor.
- Effectively the best additive approximation of F. Properties similar to coefficient estimation in linear regression.
- Note partial dependence is a function of the variables of interest accounting for the others. Not the same as ignoring all the others. Example: 2 hospitals A and B. A has higher success rate when success rate is the only variable considered. But A only has a higher success rate because B gets all the difficult cases since they are actually much better. So the conclusion of a model of success rate and difficulty comes to a very different conclusion.

Notes

- Compared to backfitting: backfitting cycles through each function/parameter and updates holding the others constant in an iterative fashion. It does not have a search path and does not seek to update a single coeffient direction.

Neural Networks

- Structural Model: $F(x) = \sum_{m=1}^{M} b_m S\left(a_{0m} + \sum_{j=1}^{m} a_{jm} x_j\right) + b_0$
    - Parametrized function with x, starting with a single hidden layer feed forward network. Linear combination of functions of linear combinations of the data.
    - Function class generated by space of a's and b's
    - Activation / Transfer / Squashing functions: sigmoid, tanh, relu
    - Note: Friedman doesn't see relu as good for structured data we are working with since its linear slope is sensitive to outliers. His examples use the sigmoid.
    - For M arbitrarily large, we can approximate any function with a single layer - universal approximation. But of course for M large, we might be better off using another method. Trees, boosting can also approximate any function.
- Score Criterion: squared error, or any custom score function
    - If you use a convex loss function, convex in the b's but not in the a's - always running into local minima when optimizing over both.
    - Eg $\text{argmin}_{a,b} \sum_{i=1}^{N} \left[y_i - b_0 - \frac{b_m}{1+\exp\left[-a_{0m} - \sum_{j=1}^{m} a_{jm} x_{ij}\right]} \right]^2$
- Search Strategy: numerical optimization -> gradient descent
- Each edge in the graph has a weight a_{ij} connecting node i in one layer to node j in the next. Each node sums the weighted inputs (along with a constant), applies the activation and passed it as output.
- Aside: Function of 1 parameter with infinite Vapnik–Chervonenkis dimension. $g(x) = sin(ax)$. I can always choose a value of a that interpolates the data for any dataset. Point: VC dimension is a better measure of complexity than number of parameters.
- Prediction: with a new input vector, propogate signals forward through connection weights to output.

Search Strategy

- Collect all the weights into a matrix $\{W_k\}_1^K$
- Score $Q(w) = \frac{1}{N} \sum_{i=1}^{N} [y_i - F(x_i; w))]^2$, here as example is squared error. The estimated weights are the arg min over the cost / loss function.
- Take gradient $g_k = \frac{\partial Q(w)}{w_k}$ - derivative of criterion wrt each of our parameters. Vector in the same dimension as the number of parameters.
- Then gradient descent:
    - Take initial guess w_0
    - Loop $w \leftarrow w - \epsilon g(w)$ until $|g(W)| \approx 0$
    - ϵ is the step size (search parameter)
- We can write our criterion as a sum of contributions to score - one from each observation: $Q(\underline{w}) = \sum_{i=1}^{N} q_i(\underline{w})$. So instead of taking the derivative of a sum, we take the sum of derivatives over parameters.
- Online gradient descent - make initial guess, make a pass over the data, sequentially look at each observation i and compute the gradient just for one observation and make a step. Advantage: don't have to pass over all data just to take a small step. Disadvantage: order of observations matters, no guarantee we end at a local minimum
- Iterated online gradient descent - we keep making passes using the rule above. A single epoch is a full pass over all observations. We have as many epochs until $|w - w_s|$ < some threshold.
- Momentum: tweaks the algorithm for faster convergence.
    - Start with initial guess for weights and gradient

    - The gradient is updated as well as the weights in the epoch loop: $g \leftarrow \alpha g_i(w) + (1-\alpha)g$ With $\alpha = 1$, we just have stochastic gradient descent.
    - Why does this work so well? Consider the batch approach instead, where every step is certainly monotonic descent (stochastic not guaranteed to be monotonic since data will point us in wrong directions on occassion). We can write this as the average gradient over the observations. This is equivalent to a sequential updating formula to update the mean - after all observations we will end in the same place: $\bar{g}_m(\underline{w}) = \frac{1}{m}\underline{g}_m\ (\underline{w}) + \frac{m-1}{m} \quad \bar{g}_{m-i}(\underline{w})$. This may be less efficient but we get an estimate of the mean at each step instead of waiting for the end.
    - In our stochastic descent, we often have elliptic gradient contours and our data points will point us at off angles. With an updating rule for the gradient, we can make the gradient estimate less jittery by smoothing it over observations. Each step tends to get better and better since it is an average over more observations.
    - Remaining problem - every time I take a step, I update the weight matrix. So we are getting an average of the current gradient over gradients calculated on different weight values. We really just want the gradient calculated on the last observation processed, so we have a declining weight, exponential decay, such that early calculated gradients on just a few observations are downweighted on the gradient.
    - This is a classic bias-variance tradeoff. The decay reduces the bias in the approximation but have higher variance since we are averaging over fewer observations. Big alpha increases the variance.
- In the beginning, we want alpha to be small since we are far from the minimum and any step is likely to be productive. Towards the last steps, we want alpha to be larger, relying much more on the end values.

Calculation of the Gradient

- For our criterion $Q(w) = \sum_{i=1}^{N} q_i(w)$ need to calculate $-g_{ik} = -\frac{\partial q_i(w)}{\partial w_k}$
- We get $-\frac{\partial q_i(w)}{\partial b_m} = (y_i - F(x_i))S_m(.)$ where $(y_i - F(x_i))$ is the error for i and S_m is the output of the mth internal node.
- For the input weights, $-\frac{\partial q_i(w)}{\partial a_{jm}} = (y - F(x_i))b_m S_m(1-S_m)x_{ij}$ where $(y - F(x_i))$ is the error of the network. Multilplied by b_m edge weight, we interpret this as the share of error due to the mth internal node. It can then ascribe that errors to each of the inputs to node m - passing the error back through the network.
- Each hidden node can operate with no other knowledge of the network. It receives its share of the error, and divies the error to pass back to its inputs. Allows for full parallelization
- Training has two steps: the feed forward and the back propagation
- Propagate signals forward through to generate an estimate of our function. Propagate error back to inputs to update the weights.

Network Features

- Handling of categorical variables vs trees. NN's use a one hot encoding unlike trees. If the target function predicted similar values for one subset of categories and another similar value for a second subset. The tree could see that and split once, never considering that variable again. If we use an encoding, turning these variables into interval-scale, we suddently need a complicated function of these variables to separate the relationship.
- Different topologies / network structures define different function classes. Changing number of units / layers approximates different functions. Generally adding nodes decreases bias / increases variance, with the antidote being more data. Changing nodes by adding layers or adding nodes to static layers - these change bias and variance in different ways that depend on the actual target function.
- The formula notation - we get a model made of a linear combination of functions of a linear combination of functions,... Nested structure.

- For each layer, can update the nodes separately. They still need to connect to all of the nodes in previous and next layers though.
- Advantage of NN: expression of model as a function of functions of functions. If your target is some function of functions, and perhaps you know this in advance, a NN would be certainly the best option. This is exactly the case for CNNs, since we are wrapping the localities of the picture into higher abstractions. This calls back to our discussion of organized vs unorganized data - just using the pixel intensities but not their locations, we are treatin organized data as unorganized and throwing away large amounts of information.

Regularization

- We of course have the danger of overfitting. Because y depends on other things other than x, our model F(x) will contain noise. Fitting to this dataset with relaxed constraints on the function class from which we choose and approximator, eventually will interpolate the data. Fits the training data better but fits the target function worse - the noise does not generalize.
- If we train the network to minimize the prediction risk, we will overfit, not equal to minizing theoretical risk $E_{yx}[y - \hat{F}(x)]^2$
- We add penalty to cost function $Q_\lambda(F) = \frac{1}{N}\sum_{i=1}^N \left[y_i - F\left(\underline{x}_i\right)\right]^2 + \lambda \cdot P(F)$. We could do this directly with a penalty added to the cost explicitly.
- Early stopping: a more implicit regularization - stop the search process early, before it finds the local minimum in the training data.
    - Divide data into training and validation set.
    - We take the t^{th} step $\underline{w}_t = \underline{w}_{t-1} + \underline{\Delta w}_t$
    - Look at cost function in training and cost function on validation, say at the end of each epoch. Over time we see that the trainin cost continues to decline but the validation set eventually the cost begins to increase again.
    - Can stop where the validation set has its cost minimized over iterations tried.
- Can be viewed as an explicit penalty on the number of steps $Q_\lambda(\underline{\omega}) = Q_L(\underline{\omega}) + \lambda \underline{P}(t)$ with λ chosen by CV on validation set.
- The penalty function is some monotone increasing function. Weight decay makes this explicit, and we add a penalty to the cost and train until full minimization $Q_\lambda(\underline{w}) = \frac{1}{N}\sum_{i=1}^N \left[y_i - F\left(\underline{x}_i; \underline{w}\right)\right]^2 + \lambda \underline{w}^t \underline{w}$. Of course do not get the time savings of early stopping.

Basis Expansions

- $F(x; \underline{a}, b) = b_0 + \sum_{m=1}^M b_m B\left(\underline{x}; \underline{a}_m\right)$
- Linear combinations of basis function expansions - equivalent to the NN structure
- One popular one has been the radial basis function: $B(\underline{x}; \underline{a}) = exp[-\frac{1}{2a_0^2}\sum_{j=1}^m \left(x_j - a_j\right)^2]$. Produces concentric circles around a point. Opposite of NN signmoid which varies in only one direction - along some combination of predictors. The RBF varies in all directions at the same rate - arranges the centers and widths of the circles to approximate the function.
- RBF is perfect for certain target function structures, capturing something that would take a huge net to capture with a NN. RBF perfect for isolated peaks, seen more in the hard sciences. Converse is also true - all dependent on the problem structure

Classification

- Map to dummy variables $y_i \in \{c_1, c_2, \cdots, c_K\} \Rightarrow \{d_{ik} = I\left(y_i = c_k\right)\}_1^K$

- Score criterion - can still use least squares, but given we are using gradient descent could plug in another function.
- In 2 class case, just use a sigmoid as our output activation and we get the same probabilities we got out of prior models.
- Multi-class: have k target function $\{F_k\left(\underline{x};\underline{w}_k\right)=P_r\left(y=c_k|\underline{x}\right)\}_i^K$ (ie K output nodes) or we use a softmax activation, just like we used in multi-class logistic regression.

Nearest Neighbor Methods

- Do not follow the paradigm of structural model, score, search
- Data structure is the same, how it is done is different
- There is no training in nearest neighbors - the data is the model.
- Take an object O with response y; have database of objects $\{y_i,o_i\}_{i=1}^N$. Given a new datapoint, want to find the object in the database most similar and use its response value
- To use this approach, we need proximity data instead of value-attribute data. Between any pair of objects, we compute a similarity (or dissimilarity). Say $D(O,O')$ = dissimilarity between O and O′. Then given an O_i, can rank the objects in the database by dissimilarity relative to this point
- When y is interval scale we can take the average response of objects with rank below a certain threshold: $\hat{y}=\text{ average }\{y_i|r_i\leq M\}$ (for squared error loss). For another loss could generalize to $\hat{y}=argmin_a\ \frac{1}{N}\sum_{R_i\leq M}L(y_i,a)$.
- When y is a class label, we turn to dummy variables again. We then treat the dummies as interval scale, and the average of an indicator of an event is just the probability of the event happening. So we end up estimating the probability of being in each class by the fraction of class k in Mth nearest neighbor of O. Then $\hat{k}(O)=argmin\ \sum_{l=1}^K L_{l,k}\hat{D}_k(O)$
- Basic assumption: objects that are similar have similar outcome values. This is not always true - analogy of knowing the target function, but the target function is not very good.
- If we had proximity data, this is the method to choose - hard to do anything else. But with attribute - value data, which is most common, we must construct dissimilarity from the attributes. Say we have two objects in terms of their attrbutes, we want our measure to have properties $D(X,X)=0$ and $D(X,X')<<$ for X, X′ similar.
- We also assume a smooth target function - where the dissimilarity function is smooth, whereas with attribute-value data we assume the function class is smooth. With a smooth target, small change is X correspond to small changes in f(x).

Bias-Variance Tradeoff

- Neighborhood size M: subbing in the model $y_i=f^*(x_i)+\epsilon$ then $\hat{f}\left(x\right)=\frac{1}{M}\sum_{r_i(x)\leq M}f^*(x_i)+\frac{1}{M}\sum_{r_i(x)\leq M}\epsilon_i$. Assuming independent errors then $E(f^*(x)-\hat{f}\left(x\right))^2=E(f^*(x)-\frac{1}{M}\sum_{r_i(x)\leq M}f^*(x_i))^2+\frac{1}{M^2}\sum_{r_i(x)\leq M}E(\epsilon_i^2)$ where $E(\epsilon_i^2)=\sigma^2$. σ^2 is a fixed population quantity that we do not observe.
- Then $E(f^*(x)-\hat{f}\left(x\right))^2=E(f^*(x)-\frac{1}{M}\sum_{r_i(x)\leq M}f^*(x_i))^2+\frac{1}{M}\sigma^2$ where $E(f^*(x)-\frac{1}{M}\sum_{r_i(x)\leq M}f^*(x_i))^2$ is squared bias and $\frac{1}{M}\sigma^2$ is the variance. This is the cleanest representation of the bias-variance tradeoff. Increasing neighborhood size decreases variance, but bias tends to go up as we allow the approximation of f to stray from the truth. If $f^*(x)$ is just linear and we increase the neighborhood symmetrically, then the average over the neighborhood wouldn't change and the bias would be unchanged. But for non-linear targets or non-symmetric neighborhood expansions, bias is affected. In high dimensions, the density of the data is very asymmetric, leading to highly asymmetric neighborhoods when we take top Z points by rank.

- We choose M to optimally trade off bias and variance. We turn to CV just like any other method. Very easy to use LOOCV with nearest neighbors. Since we are directly using the k-closest points to predict the value at a given point, can easily get the prediction for 1 neighbor, 2 neighbors, 3 neighbors, etc and pick the one with the smallest error. Leaving one vs 10 out is the same in NN methods computationally, so we can easily use LOOCV.
- Aside: Suppose you have a dataset to divide into training and test. How big should each be? The bigger the test set, the more accurrate the estimate of future error will be. But it will be more biased - we are fitting a model to a smaller subset of the data, but our end model will be fit on the whole dataset. We are not estimating the right error, since our model is approximating our final model. If the test size is reduced, the variance of our estimate of error increases. This is the decision in CV - if we use k-fold, our choice of k is bias-variance tradeoff. LOOCV is lowest bias, since each model has just 1 fewer observation than the end model.
- Learning curve of the procedure: accuracy vs N for training. Generally this curve goes up, quite steep at the beginning but has diminishing returns. Towards the end, the sample size used for training does not significantly affect accuracy. Lesson: with a lot of data, we can afford a larger test set but with smaller data we face a real tradeoff.

Using Attribute-Value Data

- Given 2 datapoints with attributes, how to define dissimilarity

1. For each variable x_j define $d_j(x_j, x'_j)$ on that attribute alone. Can be different for interval scale, circular, categorical, ordinal.... Depends on scaling of variables too
2. $D(X, X') = average\ \{d_j(x_j, x'_j)\}_1^M$, though not quite this simple in practice

- Average usually uses weight L_p norms: $D_p(X, X') = [\sum_{j=1}^m w_j d_j^p(x_j, x'_j) / \sum_{j=1}^m w_j]^{1/p}$. Typically p = 1,2,$\infty$ defining manhattan norm, euclidean norm, or max norm.
- P=1 is a diamond norm. P=2 is a circle norm. P=∞ is a square. Generally does not make a lot of difference. The weights make all the difference - this is the crucial choice to make the technique work.
- Often you have proximity data and attribute value data - need to convert attribute value to proximity in order to use both
- The weights control the relative influence of each x_J in determining the distance. Maybe naive approach is to give equal weight - but this does not give equal influence!
- Influence is proportional to the weight times the average pairwise dissimilarity for that variable. For p = 2, the influence is the weight times twice the variance of x_j over all observations.
- For interval scale variables, we can scale the variables - method depends on the units used to measure x_j. With euclidean distance say, the variance increases by the square of the scaling to x_j - therefore influence is changed since influence $I_j \sim w_j Var(\{X_{ij}\}_{i=1}^N)$. This a scaling problem but also a big statistical problem
- Suppose target function depends on x1, x2. If x1 has huge variance compared to x2, then f(x1, x2) using KNN is only going to capture a selection of x1 - the nearest neighbors may take on any value in x2 since the variation is so small. So in effect the model has no dependence on x2, but the reality may be entirely different that this approximation.
- A solution: give all variables equal influence. $w_j \sim 1/Var(\{X_{ij}\})$ for p =2 say. Equivalent to standardizing each of the variables to have the same variance!
- But equal influence may not be the best thing to do - the target function may not rely on all predictors equally. In data mining often have large number of predictors but small number have any relevance. We need to introduce some sparsity into our model - assume a few of the variables are driving our prediction. Including noise variables will actively harm our predictions. If the target function depends only on X1 then an equal influence model will have high bias (but the same variance for a fixed M).
- In high dimensions this gets much worse due to the curse of dimensionality. This is especially problematic in kernel methods; there is no automatic discovery of relevant variables. Suppose you have one dimensions X with

a uniform distribution. If you want a region on the line with 10% of the data, you can take a small neighborhood. Along 2 dimensions, taking 10% of the data defines a much bigger interval. With 10 dimensions, a region with 10% of the data takes a 90% range on each of the predictors.
- This also depends on the number of observations. In princple, large n could afford you a smaller neighborhood in high dimensions. But curse of dimensionality increases exponentially so it doesn't solve the problem completely. Nearest neighbor methods are universal.

Missing Values

- Take the average where neither the future observation nor the database point has missing values. Look just at the predictors for which we have values
- Use $\tilde{w}_j = w_j \mathbb{I}(X_j \neq missing) \mathbb{I}(X_{ij} \neq missing)$
- With correlated variables, we will get very good approximation. If there are not highly correlated variables, the approximation will not be as good

NN Problems

- There are high density regions and low density regions. Taking the NN of a high density point will have a much smaller neighborhood than a sparse region. Additionally the regions for sparse regions will be pulled towards the density. The variance of the neighborhoods will be the same because it just depends on the number of points, but the bias will be way bigger for the bigger neighborhood.
- Could instead try to control the bias and let the variance fluctuate - fix the radius of the neighborhood but now contain different numbers of points. If you define a radius with no neighbors within it at prediction time, how do you make a prediction.
- Turns out fixing a radius is not a NN method, it is a kernel method

Kernel Methods

- For regression $\hat{f}(\underline{x}) = \sum_{i=1}^{N} y_i K(\underline{x}, \underline{x}_i) \,/\, \sum_{i=1}^{N} K(\underline{x}, \underline{x}_i)$. Taking a weighted average over a space.
- Here $K(\underline{x}, \underline{x}') = I\left[D(\underline{x}, \underline{x}') \leq D_M(\underline{x})\right]$. For NN, our kernel is an indicator if the point is in our neighborhood but now they can be different. Fix a radius and consider only points within a given radius.
- We can give equal weight to all points in a neighborhood with a step function (1 or 0), but instead we could have declining influence with distance.
- Gaussian kernel, Cauchy kernel, exponential kernel, etc. Does not tend to matter than much. Just a smoother version of weighted average, weighted by the distance to the future data point.

Support Vector Machines

- $\hat{f}(\underline{x}) = \sum_{i=1}^{N} \alpha_i K(\underline{x}, \underline{x}_i)$ - linear combination of kernels near a future point. The α are coefficients fit to the data, such that we put a little neighborhood around every training point.
- Kernel around these points has to have support everywhere, like Gaussian. When we make a prediction, take the values of the kernels for all training points and take a weighted sum for the point where we want to make a prediction.
- The farther kernels will have very little influence, and in practice many of the alphas come out 0. The ones that are not 0 are the support vectors.
- This is simply another interesting way to view support vector machines.

NN / Kernel Advantages / Disadvantages

- Advantages:
  - Very simple to implement
  - Explainable, but not interpretable
  - Very fast since no training
- Limitations
  - Curse of dimensionality - adding irrelevant variables hurts in a big way
  - No model summary - model = training data. Prediction is slow, since it is a pass over all of the data. With a database size N, can find nearest point in $clog(N)$ but constant grows exponentially in dimension
  - No interpretation, except partial dependence plots. But no derived variable importance - you in fact have to provide the variable importance.

MARS

- Mutlivariate adaptive regression splines. Another attempt to make trees more accurrate
- Recall CART was a linear combination of basis functions, where those basis functions were indicators of whether a datapoint was in a given region.
- It has problems with accuracy
  - Approximation was piecewise constant causing boundary errors due to discontinuity
  - Data fragmentation - each split reduces amount of data remaining, local patterns not fitted.
  - High order interaction model since each level is an interaction
  - Instability with high variance
- MARS is another attempt to correct these problems along the lines of bagging and boosting.
- CART is discontinuous because $B_m(\underline{x}) = I(\underline{x} \in R_m)$ basis functions are discontinuous. If we use a smooth functions instead, we achieve continuity.
- Let $H(\eta) = \begin{cases} 1 & if\ \eta \geqslant 0 \\ 0 & \text{Otherwise} \end{cases}$. CART can be viewed as basis function multiplication instead of regions and splitting
  - Start out with one basis function in the model with the value 1 everywhere (akin to region is entire space)
  - Given a certain # of basis functions at an iteration m, there are m-1 basis functions, indicators of being in a region.
  - Pass over the regions, consider all predictor variables. For each predictor, consider split points in the region
  - For a given basis, predictor, split, consider an improved model removing current basis function and replacing with 2 new basis functions - ie. an additional split. This is multiplying our basis function by our step function H.
  - If new model is better than what we have seen so far, we remember these parameters. After looping over all regions / splits, we can choose the best to update the model.
- What is left is a tree where each terminal node is a basis function. Our final model is a linear combination of these basis functions, which are made up of products of our step function H for different η values and predictors.
- Continuity
  - So the only thing that makes the final model discontinuous are the H's. We then notice that H is a spline function - a zero degree spline.
  - If we choose a q degree spline instead of $H[\pm(x_j - s)] = [\pm(x_j - s)]_+^0$ we get

$b[\pm(x_j - s)] = [\pm(x_j - s)]_+^q$. A 1 degree spline is a relu, but as we add to degrees we get 0 outside of the indicator and higher order polynomials within.
    - This is the trick - replacing with piecewise linear instead of constant. Going to higher orders for continuous derivatives is a smaller improvement and just serves to complicate the algorithm for most applications.
    - Our new basis functions take the form $B_m^{(q)}(x) = \prod_{k=1}^{K_m}[t_{km}(X_{j(k,m)} - s_{km})]_+^q$
- Data Fragmentation and Interaction Order
    - Instead of replacing the parent basis function, leave it in and just add the child basis functions.
    - All of the basis functions, including previously split parents, are eligible for further splitting at each iteration
    - No longer force interaction to split on a given variable. Some nodes will have many splits coming off of them, others can have fewer
    - Tree is no longer binary as the regions can overlap. An additive, main effects, model would only split the root node.
    - We consider existing nodes for additional splits, but we don't have the same predictor appear at multiple points in the tree as allowed in CART. If a parent node is split on a variable, we won't consider that predictor for a new split on a downsteam child node (to keep bases as tensor product splines).
    - Thus these trees have a limiting depth, the total number of variables. But this does not limit the number of splits
- Full MARS algorithm
    - Uses relu's instead of 0 order splines.
    - Passes over all nodes in the tree at every pass, not just terminal nodes
    - Start with a single basis function that is 1 everywhere
    - Loop over M times for max M basis functions. At each step pass over those already in the model.
    - Pass over variables not in the tree and split points
    - For a given basis, variable, split, consider a new model with a linear combination of a negative relu and a positive relu. When we find a better split than we have found so far, this is remembered as the best split so far.
    - When we loop over everything for a given m, we add two basis functions that performed best at improving the model.

Interpretation

- Recall the ANOVA decomposition, where we can decompose into the level of interaction.
- Most functions do not involve many high order interactions. Many are well approximated just by main effects. Procedures that are very good at representing low order interactions, can get a very good model.
- Splits off of the root are main effects in MARS. The splits of the root children are second order, etc.
- We could tell the algorithm to limit the order of interaction - ie, don't grow the depth of tree further.
- How many basis functions should you have?
    - Need to regularize, with a parallel to early stopping
    - At every step we add 2 basis functions. We end up with many then prune. If we remove one at a time, see how much worse the model becomes. Find the basis function st when we remove it, the model degrades the least. Iteratively continue to remove until we have none.
    - This gives us a sequence of models, starting with the largest model ending with the smallest. Pick the best model via cross validation.
- Price of using MARS over CART: computation.
    - Before when we split a region and had two daughters and performed linear least squares fit for coefs, the

coefs for the other regions did not change due to disjointness of the regions.
    - Now regions are overlapping and we need a full linear least squares fit to evaluate the average least-squares residual.
    - This takes a lot of time and we do it repeatedly.
    - Computation of least square fit $O(nm^2 + m^3)$ for m basis functions being considered and n data points. But we do this repeatedly in a loop for a total of $O(m(N^2M^4 + NM^5))$ - this is very slow!
    - Speed it up: at each point we consider adding a basis function that is 0 up to that point and 1 after and its complementary negative basis function. By just considering the data points instead of continuous splits, we reduce to $O(NM^2)$.
- Note that MARS is still unstable - non-convex in the split points. Not scale invariant as continuity implies a scale.