

# Generalized Additive Models and Trees

Shyue Ping Ong

University of California, San Diego

Fall 2022

# Overview

- 1 Preliminaries
- 2 Generalized Additive Models
- 3 Trees
- 4 Ensemble learning
  - Boosting
  - Loss functions and robustness
  - Bagging and Random Forests

# Preliminaries

- We have covered two broad categories of methods for regression - the highly rigid linear methods and the very flexible local methods such as kNN.
- There exist an entire spectrum of methods that assuming some structured form for the unknown regression function in between these two extremes.

# Generalized Additive Models

- A generalized additive model has the form:

$$E[Y|X_1, X_2, \dots, X_p] = \alpha + \sum_{j=1}^p f_j(X_j)$$

- If  $f_j$  are expanded in terms of basis functions, this reduces to a least squares fit.
- For generalized additive models, we fit each function using a scatterplot smoother, e.g., cubic spline or kernel smoother.
- Penalized residual sum of squares is given as:

$$PRSS = \sum_{i=1}^N \left( y_i - \alpha - \sum_{j=1}^p f_j(X_j) \right)^2 + \sum_{j=1}^p \int f_j''(t_j)^2 dt_j$$

- First term is our standard sum squared error, and the right term is penalizes discontinuities (recall section on smoothing splines).

# Fitting generalized additive models

- Each function  $f_j$  is a cubic spline of component  $X_j$ .
- To obtain unique solution, we impose a further convention that the functions average to zero over the data, i.e.,  $\sum_{i=1}^N f_j(x_{ij}) = 0 \forall j$
- Backfitting algorithm:
  - ① Initialize  $\hat{\alpha} = \frac{1}{N} \sum_{i=1}^N y_i$ ,  $\hat{f}_j = 0$ .
  - ② Cycle through  $1, 2, \dots, p, 1, 2, \dots, p$

$$\begin{aligned}\hat{f}_j &\leftarrow S_j \left[ \{y_i - \hat{\alpha} - \sum_{k \neq j} \hat{f}_k(x_{ik})\}_1^N \right] \\ \hat{f}_j &\leftarrow \hat{f}_j - \frac{1}{N} \sum_{i=1}^N \hat{f}_j(x_{ij})\end{aligned}$$

- Conceptually, fitting a cubic smoothing spline  $S_j$  to the residual  $y_i - \hat{\alpha} - \sum_{k \neq j} \hat{f}_k(x_{ik})$  for each  $f_j$ , and iterate until all  $\hat{f}_j$ 's stabilize.

## Extensions of Generalized Additive Models

- Note that we are not limited to cubic splines. E.g, local polynomial and kernel methods, linear regression, and surface smoothers etc. can be used with the appropriate choice of smoother  $S_j$ .
- GAMs can be used for classification as well, using the logit *link* function. For example, for binary classification:

$$\log \frac{P(Y = 1|X)}{P(Y = 0|X)} = \log \frac{P(Y = 1|X)}{1 - P(Y = 1|X)} = \alpha + \sum_{j=1}^p f_j(X_j)$$

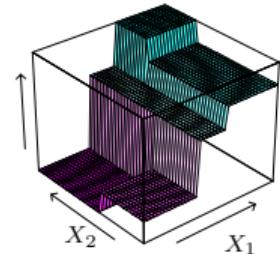
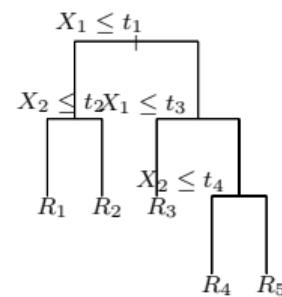
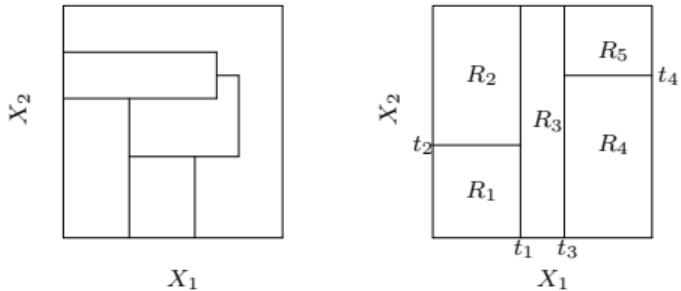
Very commonly used in medical research: outcomes encoded as 0 or 1 (e.g., death/relapse of disease).

# Tree-based methods

- Partition feature space into regions (e.g., rectangles for 2 features case), and simple model (e.g., constant) fitted into each rectangle.
- Classification And Regression Trees (CART)

$$\hat{f}(X) = \sum_m c_m I\{(X_1, X_2) \in R_m\}$$

- Main question: How to decide on partitions/topology?



## Regression tree fitting

- For CART, it is clear that each region should just be given by the average of the observations  $y_i$  in that region to minimize sum of squares.
- Best partition is usually not computationally tractable.
- Greedy algorithm: Start with all data, choose splitting variable  $X_j$  and split point  $s$  such that:

$$\min_{X_j, s} \left[ \sum_{x_i \in R_1(X_j, s)} (y_i - c_1)^2 + \sum_{x_i \in R_2(X_j, s)} (y_i - c_2)^2 \right]$$

- For each  $X_j$ , splitting point  $s$  can be found quickly via scanning of the variables.
- This process is repeated for each region to grow the tree.
- Choice of tree size determines complexity of model - too large a tree results in overfitting, too small results in underfitting.

# Cost-Complexity Tree Pruning

- Generate the tree until a minimum node size is achieved.
- Note: perfect performance on training data can always be obtained with an arbitrarily large tree, e.g., when the final ‘leaf’ nodes each contain only one training observation.
- Number of samples in a node is therefore an indicator of tree complexity.
- Let subtree  $T \subset T_0$  be any tree that can be obtained by pruning  $T_0$ .
- Cost-complexity criterion:

$$C_\alpha(T) = \sum_{m=1}^{|T|} \sum_{x_i \in R_m} (y_i - \hat{c}_m)^2 + \alpha |T|$$

- Find the subtree  $T_\alpha$  that minimizes  $C_\alpha(T)$ .  $\alpha$  controls complexity. Large  $\alpha$  results in smaller tree.
- Weakest link pruning: successively collapse each node that produces the smallest increase in  $\sum_{x_i \in R_m} (y_i - \hat{c}_m)^2$ .

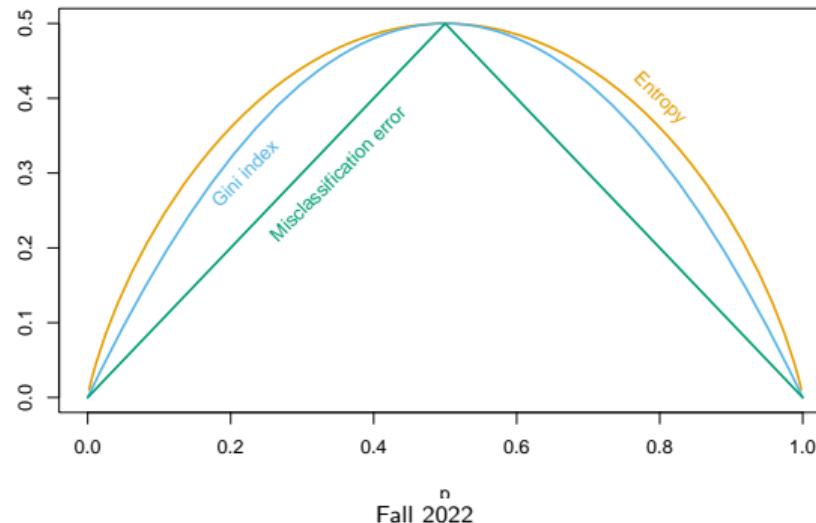
# Classification Trees

- Instead of squared error, we need to use alternative *node impurity* measures:

Misclassification error  $1/N_m \sum_{i \in R_m} I(y_i \neq k(m)) = 1 - p_{mk}(m)$

Gini index  $\sum_{k \neq k'} \hat{p}_{mk} \hat{p}_{mk'}$

Cross-entropy  $-\sum_{k=1}^K \hat{p}_{mk} \log \hat{p}_{mk}$



## Miscellaneous Issues with Trees

- Trees can be highly interpretable.
- Instability: small data changes can lead to very different splits.
- Lack of smoothness
- For some categorical problems, a misclassification in one category is more serious than another, e.g., it is better to have a false positive for a disease than a false negative. This can be handled by weighting the loss functions appropriately.

## Example: Lab 2 revisited

- We will now play around with the metal/insulator classification problem in Lab 2.
- However, we will make a few changes. First, we will not bother with NaN values. Imputing values is ok for many other domains. In materials science, imputing arbitrary values is a recipe for disaster.
- Second, we will only select a smaller subset of elemental properties to construct our decision tree with. Namely, '*AtomicRadius*', '*AtomicWeight*', '*Column*', '*Electronegativity*', '*Row*'. These properties are available for most elements and we avoid obviously correlated features, e.g., *AtomicRadius* and *AtomicVolume*.

# Decision Tree Regressor and Classifier in scikit-learn

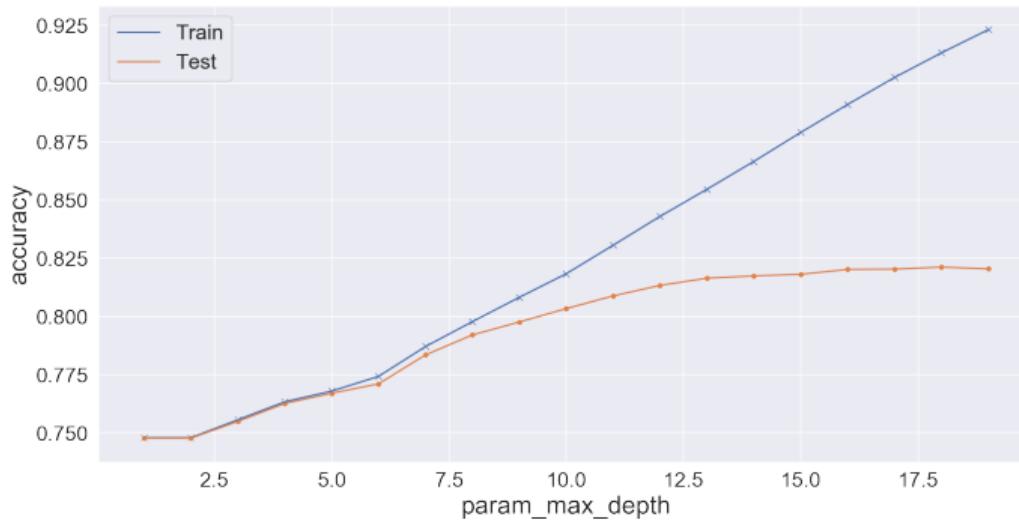
```
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
from sklearn.model_selection import train_test_split
from sklearn.tree import export_text

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.1)

decision_tree = DecisionTreeClassifier(criterion="entropy", random_state=0,
                                       max_depth=5)
decision_tree = decision_tree.fit(x_train, y_train)
train_accuracy = decision_tree.score(x_train, y_train)
test_accuracy = decision_tree.score(x_test, y_test)
r = export_text(decision_tree, feature_names=list(x.columns))
print("Train accuracy = %.3f; test accuracy: %.3f" % (train_accuracy, test_accuracy))
print(r)

decision_tree = DecisionTreeRegressor(criterion="mse",
                                       random_state=0, max_depth=10)
decision_tree = decision_tree.fit(x_train, y_train)
y_pred = decision_tree.predict(x_test)
```

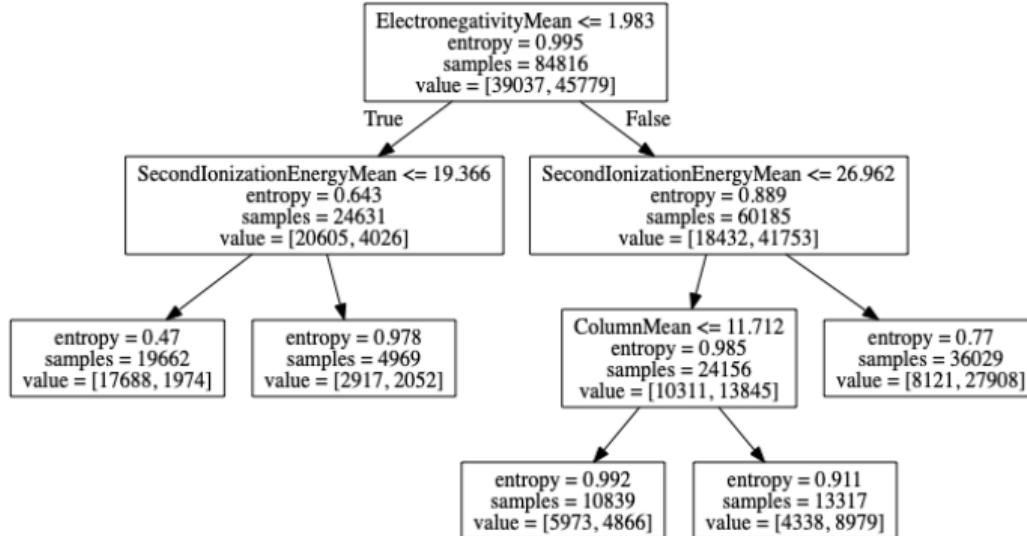
# Classification accuracy



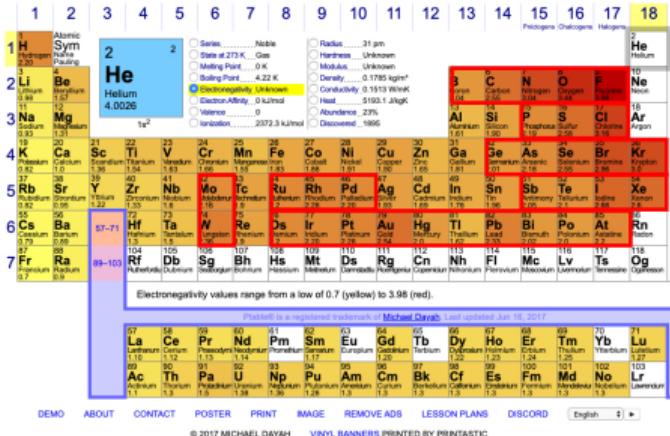
- Quite clearly, we cannot do much better than a  $\sim 82\%$  accuracy (test misclassification rate of about 18%) with a tree-depth of around 15.
- Also, the training and test errors diverge significantly after a depth of around 8, which indicates overfitting.

# Interpreting the tree

- A 8-deep tree is not very easy to read. Here, we will use cost-complexity pruning with a parameter  $\alpha = 0.01$  to prune the tree. The resulting tree has an accuracy of around 74%. Let's see how the decision is being made at the first few levels.



# Interpreting the tree, contd.



- Compounds with mean  $\chi \leq 2.03$  are mostly classified as metals.
- Compounds with mean  $\chi > 2.03$  are classified as insulators, i.e., mostly ionic compounds containing chalcogenides and halides with high  $\chi$ .

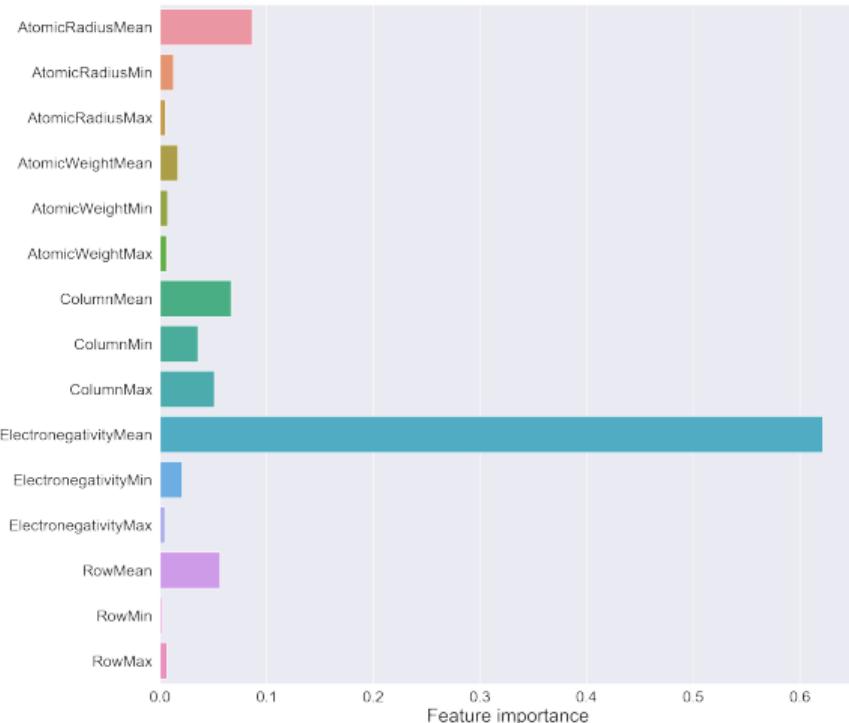
```

--- ElectronegativityMean <= 2.03
|--- ColumnMin <= 2.50
|   |--- ElectronegativityMax <= 5.09
|   |   |--- class: 0
|   |--- ElectronegativityMax > 5.09
|   |   |--- class: 1
|--- ColumnMin > 2.50
|   |--- ColumnMax <= 44.50
|   |   |--- class: 0
|   |--- ColumnMax > 44.50
|   |   |--- class: 0
|--- ElectronegativityMean > 2.03
|   |--- AtomicRadiusMean <= 0.98
|   |   |--- AtomicWeightMean <= 22.98
|   |   |   |--- class: 1
|   |   |--- AtomicWeightMean > 22.98
|   |   |   |--- class: 1
|   |--- AtomicRadiusMean > 0.98
|   |   |--- ColumnMax <= 31.00
|   |   |   |--- class: 0
|   |   |--- ColumnMax > 31.00
|   |   |   |--- class: 1

```

# Feature importance

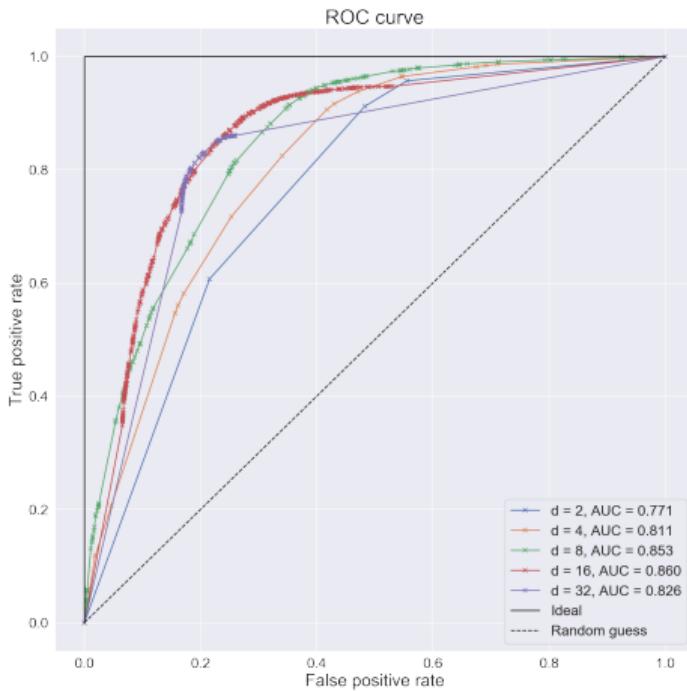
- Another way of interpreting trees is using the feature importance.
- The importance of a feature is the (normalized) total reduction of the criterion brought by that feature.



# Receiver Operating Characteristic (ROC) Curve

$$\begin{aligned} TPR &= \frac{TP}{P} = \frac{TP}{TP + FN} \\ FPR &= \frac{FP}{N} = \frac{FP}{TN + FP} \end{aligned}$$

- Plot of the TPR (*sensitivity*) vs FPR (*1-selectivity*).
- $y = x$  line denotes random guessing ( $TPR = FPR$ ).
- The greater the area under curve (AUC), better the performance.



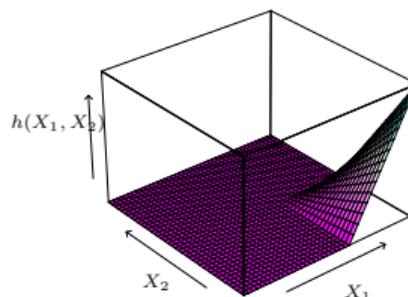
# Multivariate Adaptive Regression Splines (MARS)

- Essentially a modification of CART to use step-wise linear regression.
- MARS uses piece-wise linear basis functions:

$$(x - t)_+ = \begin{cases} x - t & , x > t \\ 0 & , \text{otherwise} \end{cases}$$

$$(t - x)_+ = \begin{cases} t - x & , x < t \\ 0 & , \text{otherwise} \end{cases}$$

- Implementation available in the [py-earth](#) package.



# Ensemble learning

- So far, we have covered the basics of using a single model (linear, kernel, tree) to perform an ML prediction.
- In *ensemble learning*, we use multiple models and average the results to improve prediction performance.
- Advantage: lower variance and in many cases, dramatically improved prediction performance.
- Disadvantage: some of the interpretability is lost in the process.
- Here, we will cover two of the most popular ensemble learning approaches - *boosting* and *bagging*.
- While ensemble learning can be applied to any of the previous ML methods, we will focus here on their application to decision trees.

# Boosting

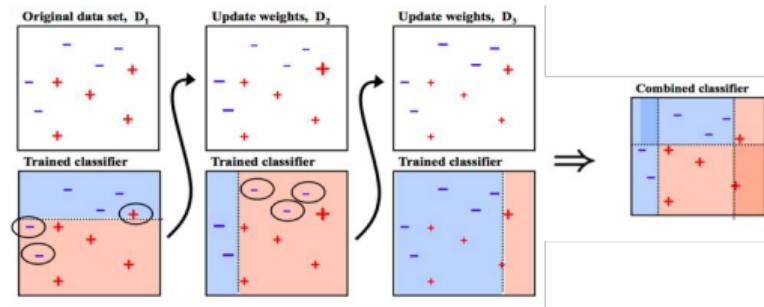
- One of the most successful ML approaches in the past few decades.
- Concept: combine many "weak" learners in a "committee".
- Can be used for either classification or regression.
- Weak classifier: One whose error rate is slightly better than random guessing.
- Apply weak classifier to repeatedly modified versions of data to produce a sequence of weak classifiers.
- Predictions from sequence are combined using weighted majority vote:

$$G(x) = \text{sign} \left( \sum_{m=1}^M \alpha_m G_m(x) \right)$$

- Weights  $\alpha_m$  are computed by boosting algorithm and is the contribution of each weak learner  $G_m(x)$ .
- While  $G(x)$  can be any classifier, we will focus here on using decision trees as the base classifier.

# AdaBoost.M1 Algorithm (Classification)

- ① Initialize observation weights as  $w_i = 1/N$ .
- ② For  $m = 1$  to  $M$ :
  - ① Fit classifier  $G_m(x)$  to training data using weights  $w_i$ .
  - ② Compute  $err_m = \frac{\sum_{i=1}^N w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^N w_i}$
  - ③ Compute  $\alpha_m = \log \frac{1 - err_m}{err_m}$ .
  - ④ Set  $w_i = w_i \exp[\alpha_m I(y_i \neq G_m(x_i))]$ ,  $i = 1, 2, \dots, N$ . Conceptually, increase weights in step  $m$  for observations that are misclassified in step  $m - 1$ .
- ③ Output  $G(x) = \text{sign} \left( \sum_{m=1}^M \alpha_m G_m(x) \right)$



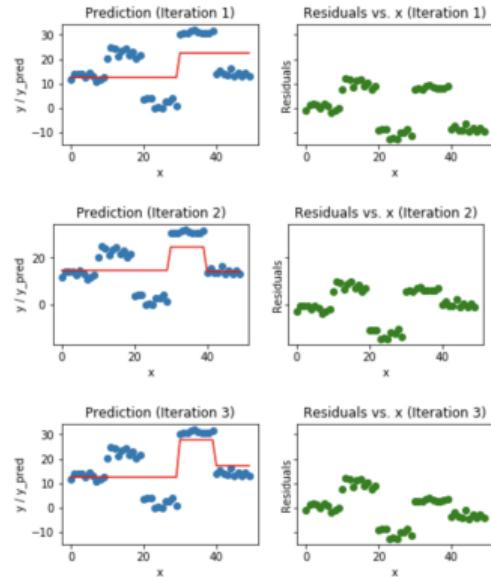
# AdaBoost in scikit-learn

```
from sklearn.ensemble import AdaBoostClassifier

x_train, x_test, y_train, y_test = train_test_split(x, y_class, test_size=0.2)

decision_tree = AdaBoostClassifier(DecisionTreeClassifier(criterion="entropy", random_state=0, max_depth=
                                         n_estimators=20))
decision_tree = decision_tree.fit(x_train, y_train)
train_accuracy = decision_tree.score(x_train, y_train)
test_accuracy = decision_tree.score(x_test, y_test)
```

# Gradient Boosting



- We can think of the algorithm in Slide 22 as essentially a forward stage-wise fit of an additive model  $f(x) = \sum_{m=1}^M \alpha_m G_m(x)$  (refer to [1] for details).
- Greedy approach in that it seeks to maximally reduce the loss at each step, i.e., steepest descent, by adjusting the weights iteratively.
- In contrast, *gradient boosting* attempts to fit a new learner to the residuals of the errors from the previous step.

Figure: Source: Gradient  
Boosting from Scratch

# Gradient Boosting in Scikit-Learn

```
from sklearn.ensemble import GradientBoostingClassifier, GradientBoostingRegressor

model = GradientBoostingClassifier(n_estimators=50)
model.fit(x, y_class)
model.predict(x)

model = GradientBoostingRegressor(n_estimators=50)
model.fit(x, y_reg)
model.predict(x)
```

# Loss functions for regression

- We have thus far focused on the squared error loss  $L(y, f(x)) = (y - f(x))^2$
- Another common loss function is the absolute error  $L(y, f(x)) = |y - f(x)|$
- MSE penalizes outliers with large observed residuals severely, and hence is less robust in data with long-tailed distributions.
- MAE is more robust against outliers.
- Other criteria include the Huber loss:

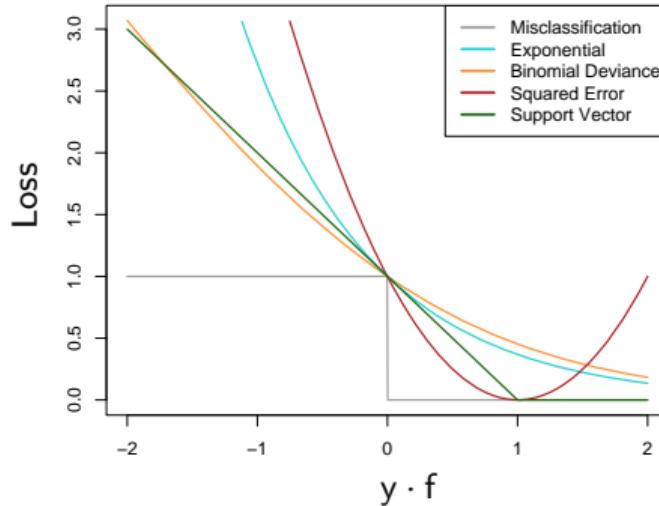
$$L(y, f(x)) = \begin{cases} (y - f(x))^2 & |y - f(x)| \leq \delta \\ 2\delta(y - f(x)) - \delta^2 & \text{otherwise} \end{cases}$$

## Loss functions for binary classification

- Consider a simple binary classification with two levels (-1, 1). The decision boundary is at 0.
- Using the square error does not make sense, since we only care about whether it is  $> 0$  or  $< 0$ .
- Margin  $yf(x)$  is positive when prediction and actual value is in the same class, and negative if they are in opposite classes.
- Need a loss that penalizes negative values much more than positive values for margins, i.e., monotone decreasing function.
- Exponential loss:  $L(y, f(x)) = e^{-yf(x)}$
- Binomial/multinomial loss (can be used for K-classes):

$$L(y, p(x)) = - \sum_{k=1}^K I(y = G_k) f_k(x) + \log \left( \sum_{l=1}^K e^{f_l(x)} \right)$$

# Loss functions for binary classification

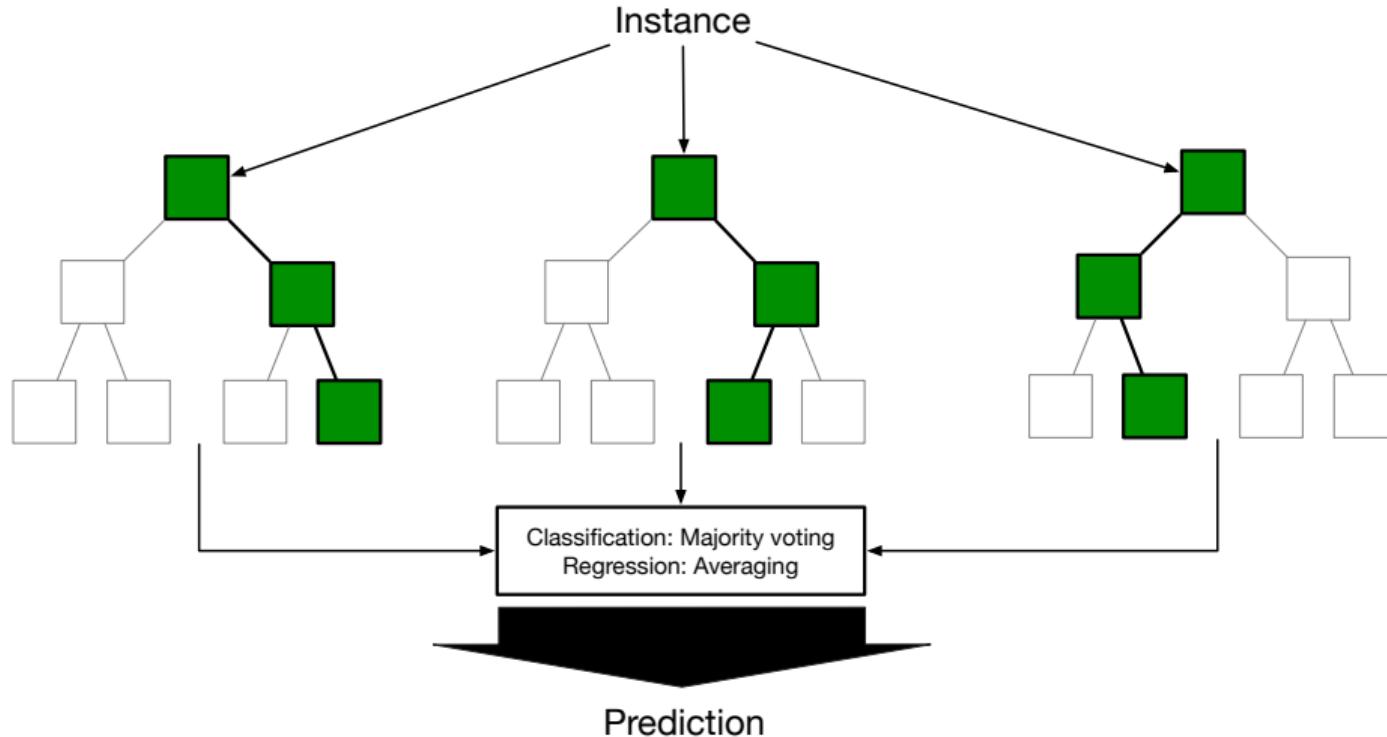


**Figure:** Loss functions for binary classification. Response:  $y = \pm 1$ . X-axis is the margin  $y \cdot f$ .  
Misclassification :  $I(\text{sign}(f) \neq y)$ ; exponential:  $e^{-yf}$ ; binomial deviance:  $\log(1 + e^{-2yf})$ ; squared error:  $(y - f)^2$  ; and support vector:  $(1 - yf)_+$ . Source: [1]

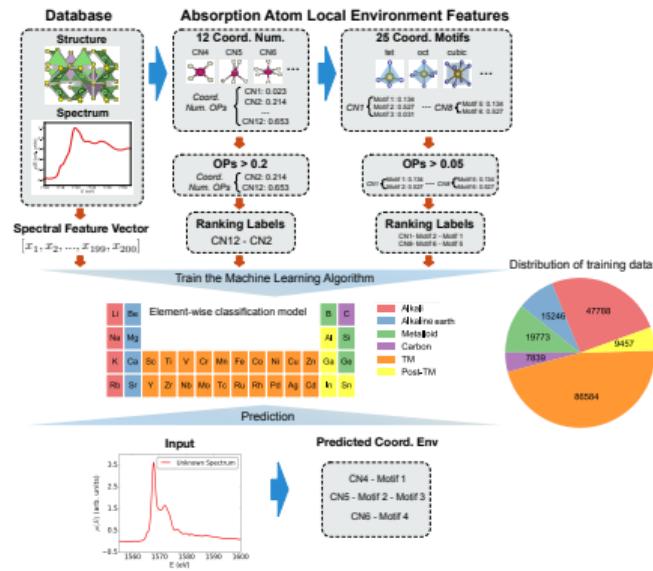
# Random Forests

- Bagging: average many noisy, unbiased models to reduce variance.
- Random forest: Grow  $B$  trees at random and average the results. Classification: majority vote (mode), regression: mean.
- Tree growing:
  - ① At each branch, select  $m$  variables at random from  $p$  variables.
  - ② Determine best split among the  $m$ .
  - ③ Split node into two daughter nodes.
  - ④ Repeat until minimum node size is reached.

# Random Forest Algorithm



# Example: Identification of Local Environments from K-edge XANES



**Figure:** Workflow for classification of K-edge XANES spectra into one of 25 coordination environments.[?]

# Example: Identification of Local Environments from K-edge XANES

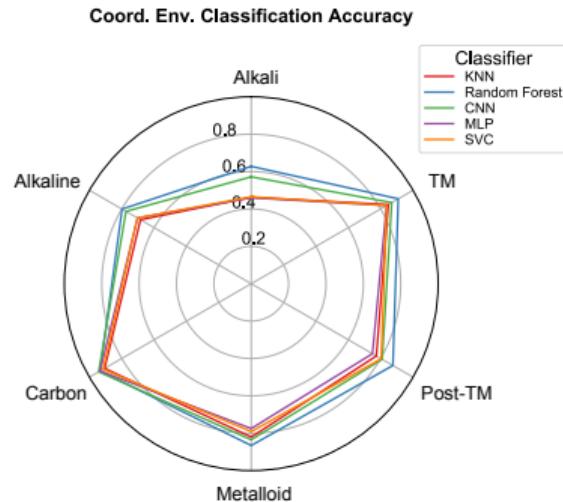


Figure: Comparison of different ML methods for K-edge XANAES classification.[?]

# Bibliography



Trevor Hastie, Robert Tibshirani, and Jerome Friedman.

*The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition.*

Springer, New York, NY, 2nd edition edition, 2016.

# The End