TREE-BASED METHODS

Dr. Aric LaBarr

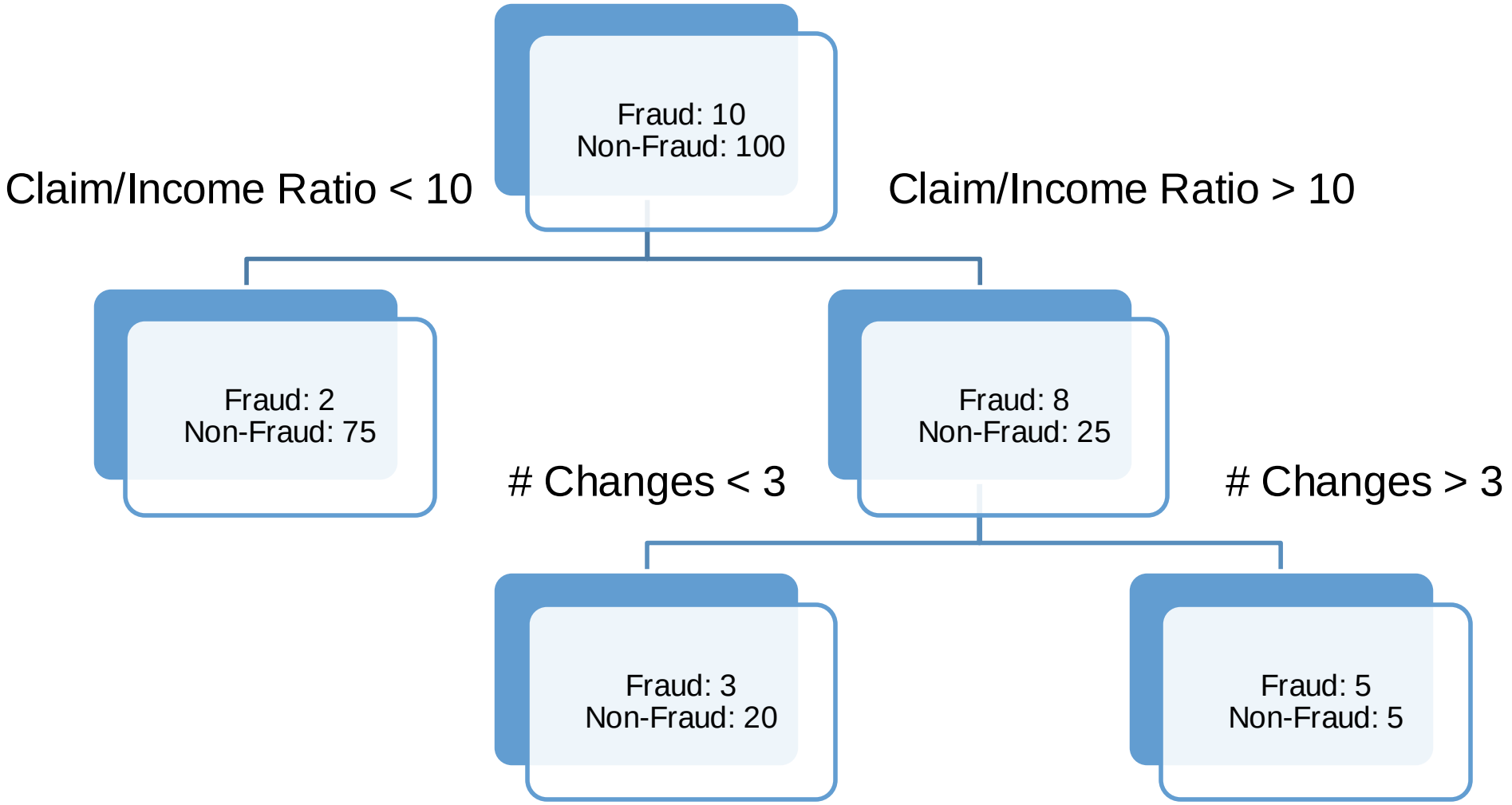
Institute for Advanced Analytics

DECISION TREE REVIEW

Decision Trees

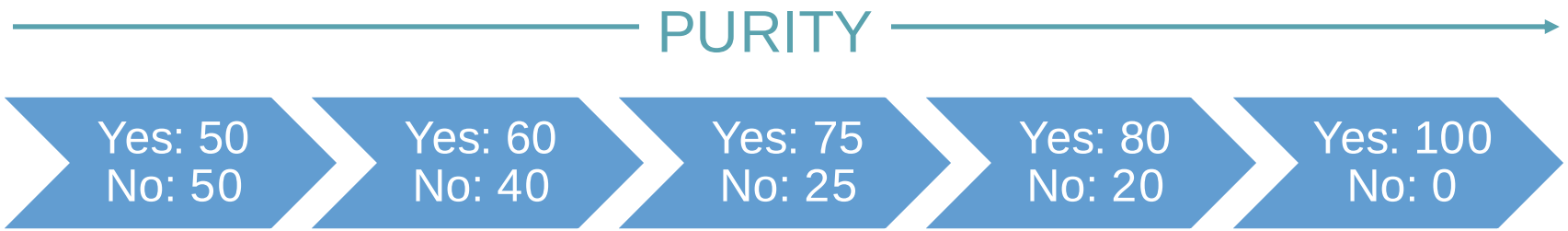
- A tree is built by recursively splitting the data into successively **purer** subsets of data.
- Splitting is done according to some condition.
  - Measurements of purity – Gini, entropy, misclassification error rate
  - Chi-Squared tests (CHAID)

Decision Trees



Decision Trees – Selecting the Split

- Variety of purity measures used to select the best split, but all look at **impurity** of a node.



- More pure a leaf is, the less error we make in that leaf.
- Entropy, Gini, Classification Error

BAGGING

Bootstrap Samples

- Random samples of your data *with replacement* that are the same size as original dataset.
- Some observations will not be sampled – called **out-of-bag observations**.

- Example: 10 observations (labeled 1 through 10)

Bootstrap Sample	Training Observations	Out-of-Bag Observations
1	7, 10, 10, 5, 3, 8, 6, 2, 1, 5	4, 9
2	1, 6, 10, 6, 8, 7, 7, 7, 8, 10	2, 3, 4, 5, 9
3	10, 2, 1, 4, 8, 10, 2, 4, 3, 3	5, 6, 7, 9

Bootstrap Samples

- Proven (**BY MATH**) that a bootstrap sample will contain approximately 63% of the observations.
- Sample size is the same as original as some observations are repeated.

- What is the value of bootstrap sampling?
  - Simulation (ex: used in time series for estimating confidence intervals of complicated forecasts quickly)
  - Create ensemble models using different training datasets (bagging)

Bagging (**B**ootstrap **Agg**regat**ing**)

- Take *k* bootstrap samples.
- For each of the *k* bootstrap samples, create a model (classification for example) using that sample as training data.
  - Builds *k* different models!

- **Ensemble** the *k* different models.

Bagging Example

- 10 observations in original dataset as follows:

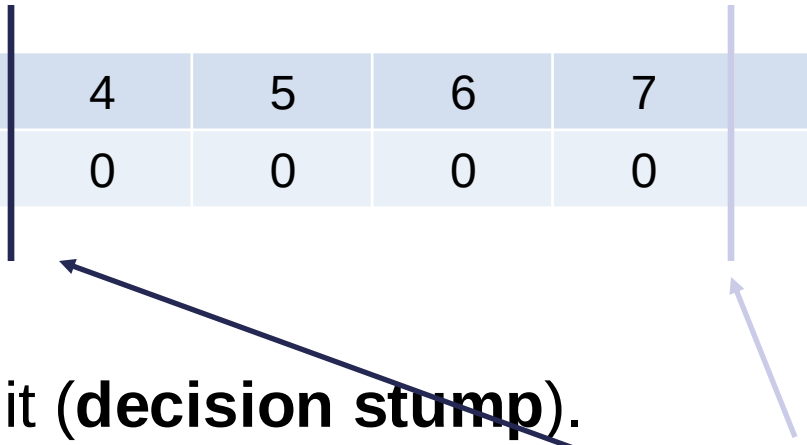
X	1	2	3	4	5	6	7	8	9	10
Y	1	1	1	0	0	0	0	1	1	1

- Build a tree with only one split (**decision stump**).

Bagging Example

- 10 observations in original dataset as follows:

X	1	2	3	4	5	6	7	8	9	10
Y	1	1	1	0	0	0	0	1	1	1

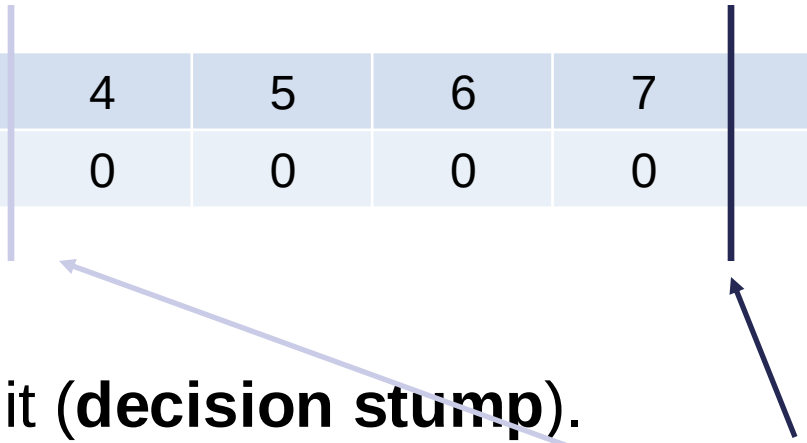


- Build a tree with only one split (**decision stump**).
- Best accuracy we can get is 70% → split at either 3.5 or 7.5.

Bagging Example

- 10 observations in original dataset as follows:

X	1	2	3	4	5	6	7	8	9	10
Y	1	1	1	0	0	0	0	1	1	1



- Build a tree with only one split (**decision stump**).
- Best accuracy we can get is 70% → split at either 3.5 or 7.5.

Bagging Example

- Bagging might be able to help:
  1. Take 10 bootstrap samples.
  2. Build decision stump for each.
  3. Aggregate these rules into a voting ensemble.
  4. Test the performance of the voting ensemble on the whole dataset.

First Bootstrap Sample

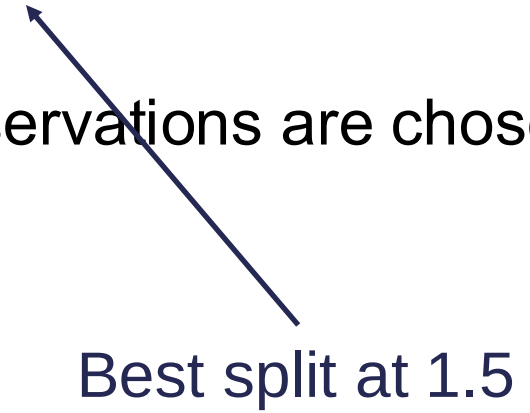
X	2	8	8	8	8	8	9	9	9	10
Y	1	1	1	1	1	1	1	1	1	1

- Some observations are chosen, and some are not…

First Bootstrap Sample

X	2	8	8	8	8	8	9	9	9	10
Y	1	1	1	1	1	1	1	1	1	1

- Some observations are chosen, and some are not…



First **5** Bootstrap Samples

X	2	8	8	8	8	8	9	9	9	10	100%
Y	1	1	1	1	1	1	1	1	1	1	
X	2	3	4	4	5	5	7	10	10	10	80%
Y	1	1	0	0	0	0	0	1	1	1	
X	2	2	4	4	4	5	5	6	8	8	80%
Y	1	1	0	0	0	0	0	0	1	1	
X	1	1	1	1	2	4	6	7	7	10	90%
Y	1	1	1	1	1	0	0	0	0	1	
X	1	3	5	5	6	7	7	8	9	10	80%
Y	1	1	0	0	0	0	0	1	1	1	

Next 5 Bootstrap Samples

X	2	3	3	5	5	6	6	8	9	10	70%
Y	1	1	1	0	0	0	0	1	1	1	
X	2	2	5	5	6	7	7	8	8	9	80%
Y	1	1	0	0	0	0	0	1	1	1	
X	1	4	4	5	5	6	8	8	9	10	90%
Y	1	0	0	0	0	0	1	1	1	1	
X	2	2	3	5	6	7	8	9	10	10	70%
Y	1	1	1	0	0	0	1	1	1	1	
X	1	1	2	3	5	8	10	10	10	10	90%
Y	1	1	1	1	0	1	1	1	1	1	

Bootstrap Summary

Round	X = 1	X = 2	X = 3	X = 4	X = 5	X = 6	X = 7	X = 8	X = 9	X = 10
1	0	1	1	1	1	1	1	1	1	1
2	0	0	0	0	0	0	0	1	1	1
3	0	0	0	0	0	0	0	1	1	1
4	1	1	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	1	1	1
6	1	1	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1	1	1
8	0	0	0	0	0	0	0	1	1	1
9	0	0	0	0	0	0	0	1	1	1
10	1	1	1	1	1	1	1	1	1	1
Avg.	0.3	0.4	0.3	**0.2**	**0.2**	**0.2**	**0.2**	0.8	0.8	0.8
Pred.	1	1	1	0	0	0	0	1	1	1
Truth	1	1	1	0	0	0	0	1	1	1

Cut-off Above 0.2 →

Bagging Summary

- Improves generalization error on models with high variance (simple tree-based models for example).
- If base classifier is stable (not suffering from high variance), bagging can actually make it worse!
- Does not focus on any particular observations in the training data (unlike boosting).

RANDOM FORESTS

Random Forests

- **Random forests are ensembles of decision trees** (similar to bagging example).
- Ensembles of decision trees work best when they **find different patterns in the data**.

- Bagging tends to create trees that pick up the same pattern… ☹

Random Forests

- Random forests get around this correlation between trees by not only using bootstrapped samples, but **subsets of variables for each split** and **unpruned decision trees** in each ensemble.

1 obs per leaf for classification → perfect prediction
5 obs per leaf for regression → not perfect

Random Forests

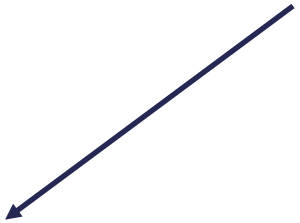
- Random forests get around this correlation between trees by not only using bootstrapped samples, but **subsets of variables for each split** and **unpruned decision trees** in each ensemble.

- Results from the trees are ensembled together into one voting system.

- Parameters to tune:
  1. Number of trees
  2. Number of variables for each split
  3. Depth of tree (defaults to unpruned)

Ames Data

```
set.seed(4321)

training <- ames %>% sample_frac(0.7)
testing <- anti_join(ames, training, by = 'id')

training <- training %>%
  select(Sale_Price,
         Bedroom_AbvGr,
         Year_Built,
         Mo_Sold,
         Lot_Area,
         Street,
         Central_Air,
         First_Flr_SF,
         Second_Flr_SF,
         Full_Bath,
         Half_Bath,
         Fireplaces,
         Garage_Area,
         Gr_Liv_Area,
         TotRms_AbvGrd)

training.df <- as.data.frame(training)
```



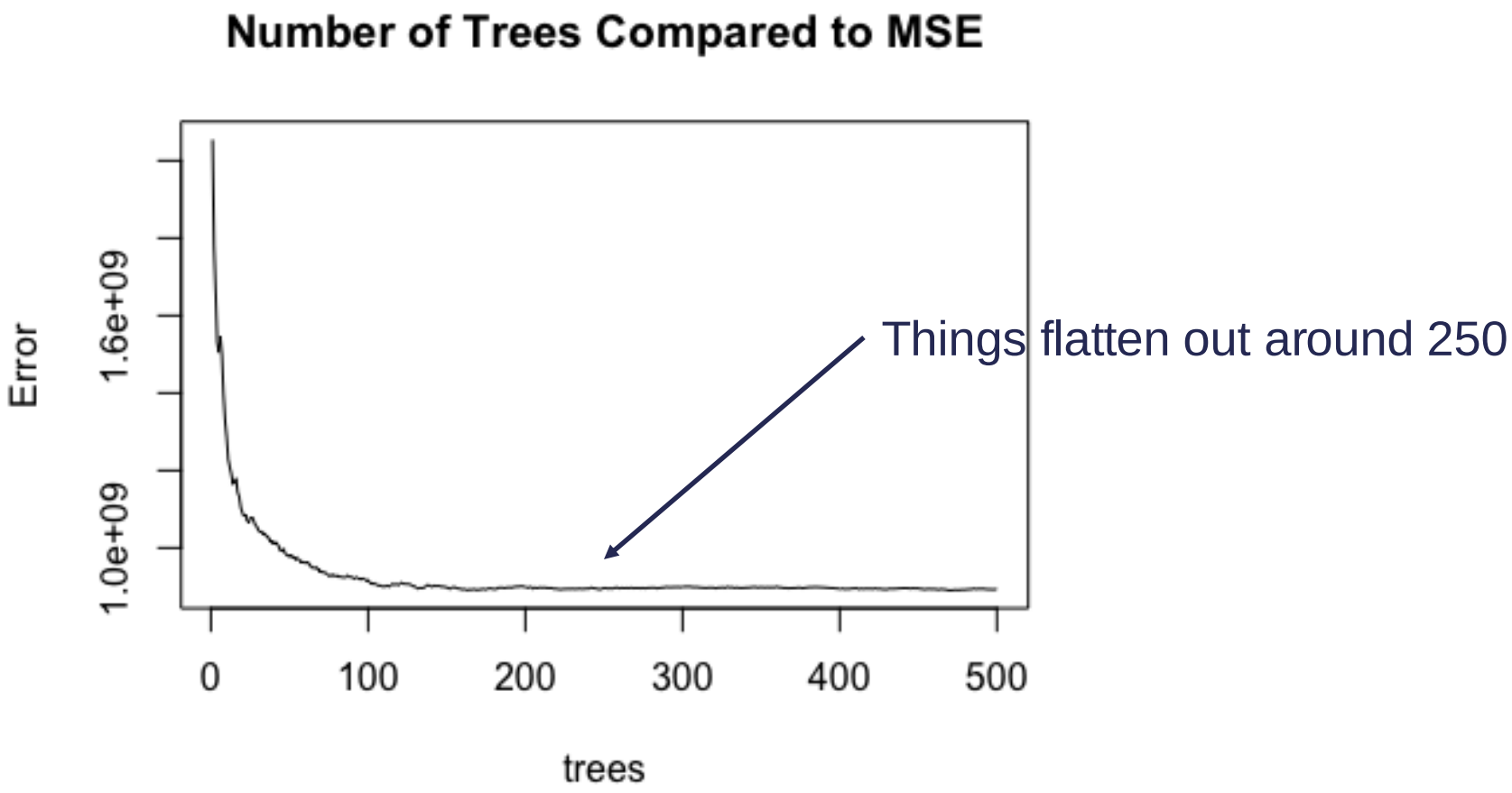
Need a data frame structure for random forest function

Random Forests

```
set.seed(12345)
rf.ames <- randomForest(Sale_Price ~ ., data = training.df,
                        ntree = 500, importance = TRUE)
```

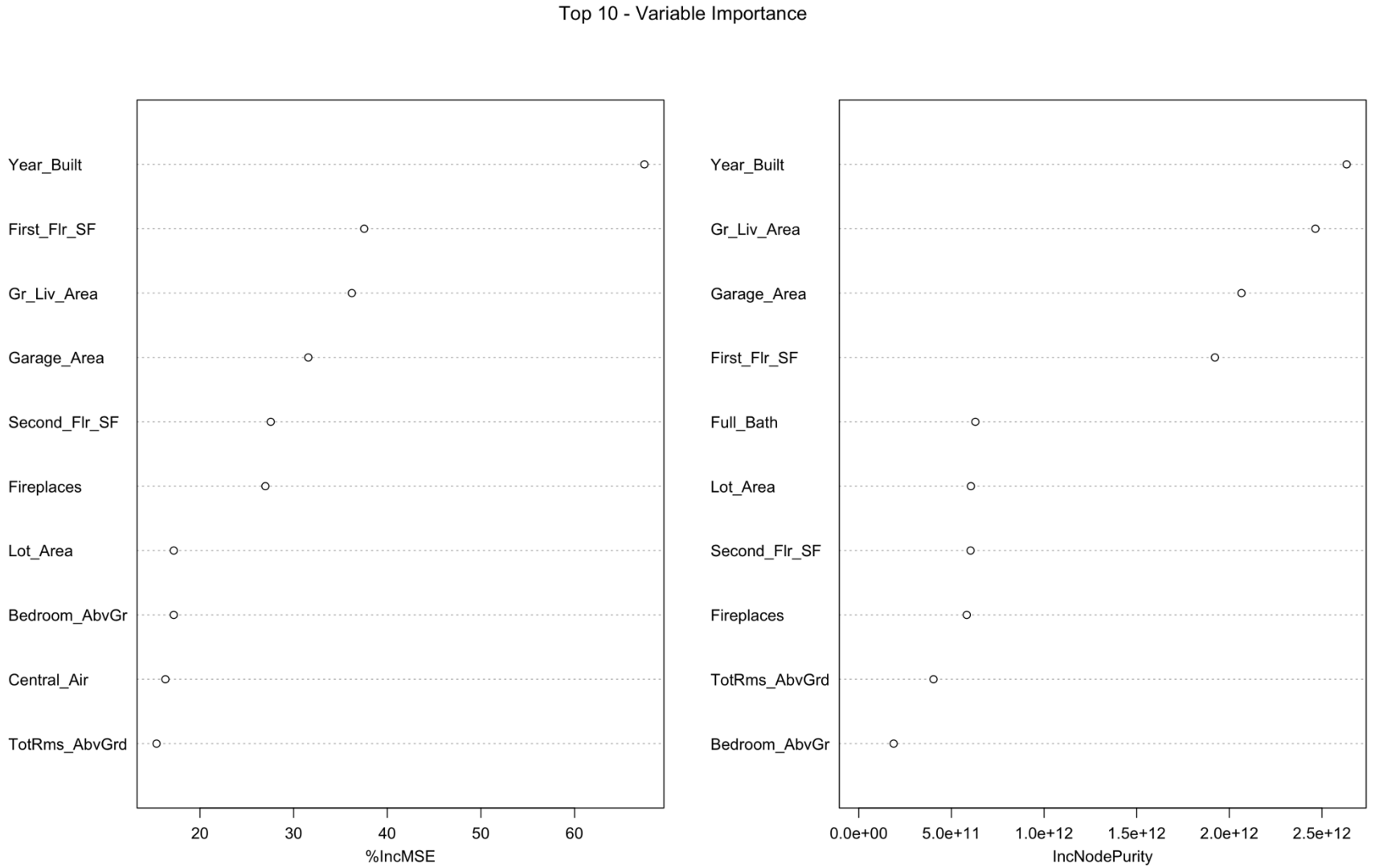
Random Forests

```
plot(rf.ames, main = "Number of Trees Compared to MSE")
```



Variable Importance

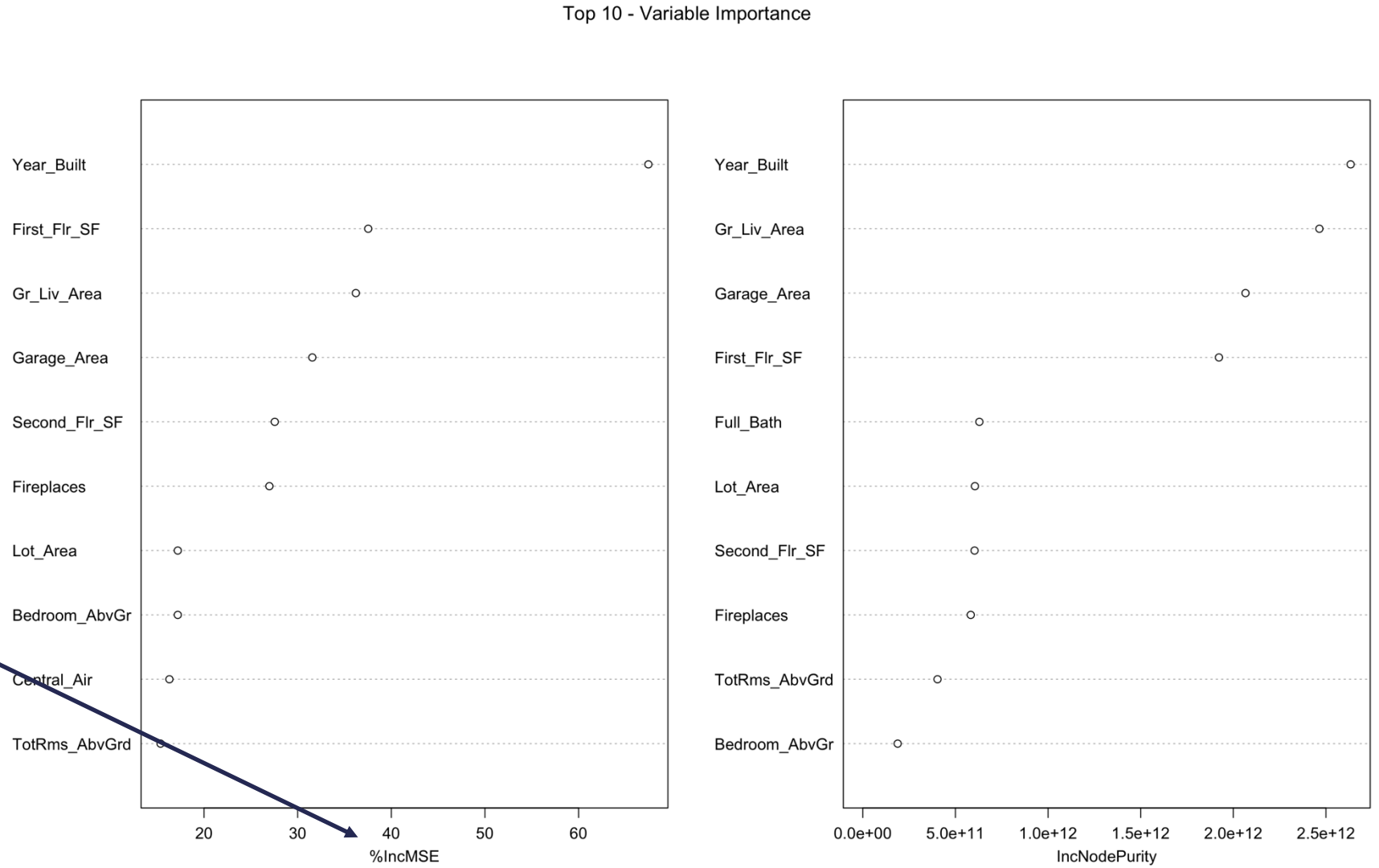
```
varImpPlot(rf.ames,
           sort = TRUE,
           n.var = 10,
           main = "Top 10 -
Variable Importance")
```



Variable Importance

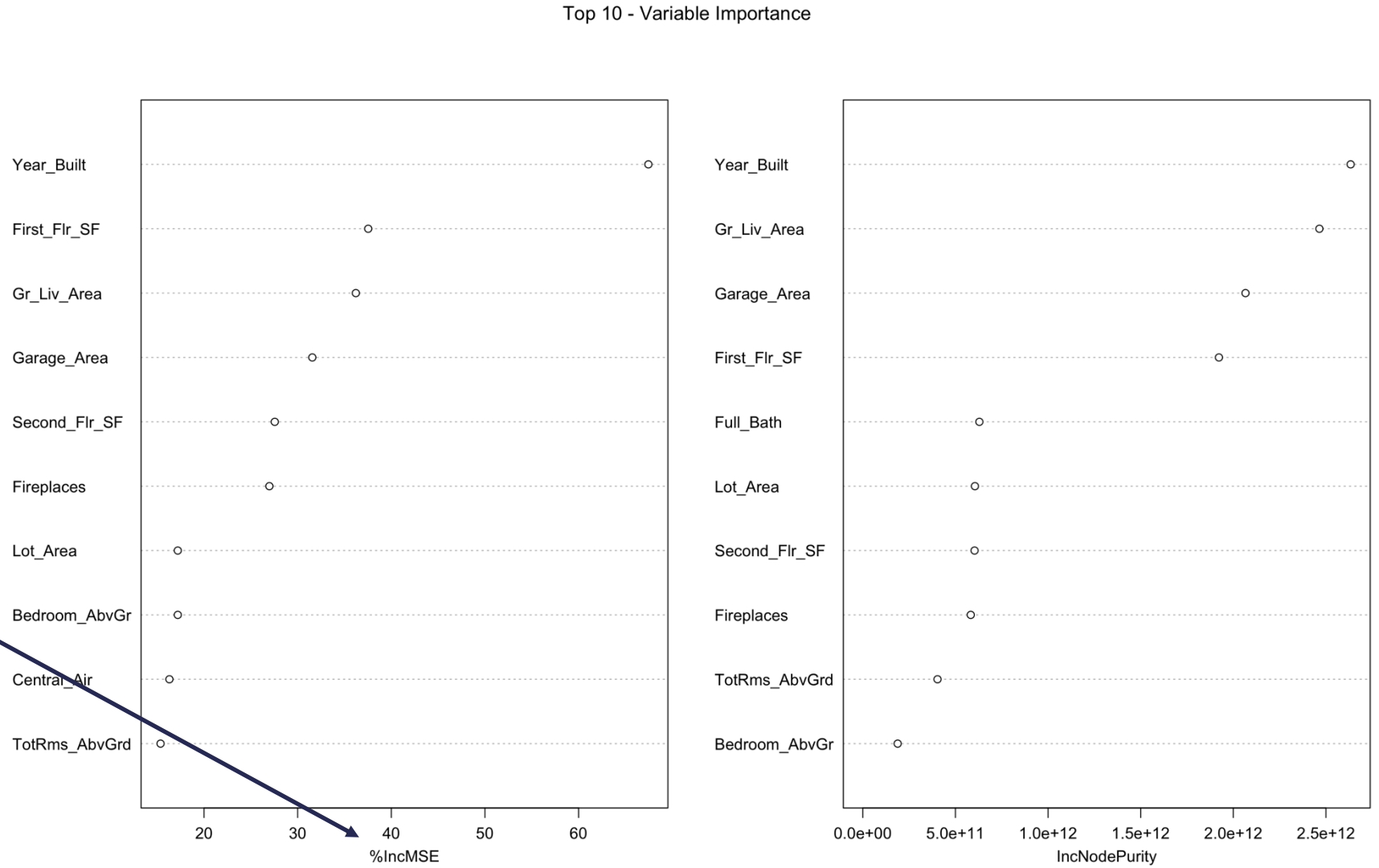
```
varImpPlot(rf.ames,
           sort = TRUE,
           n.var = 10,
           main = "Top 10 -
Variable Importance")
```

Percentage increase in MSE when variable "excluded"



Variable Importance

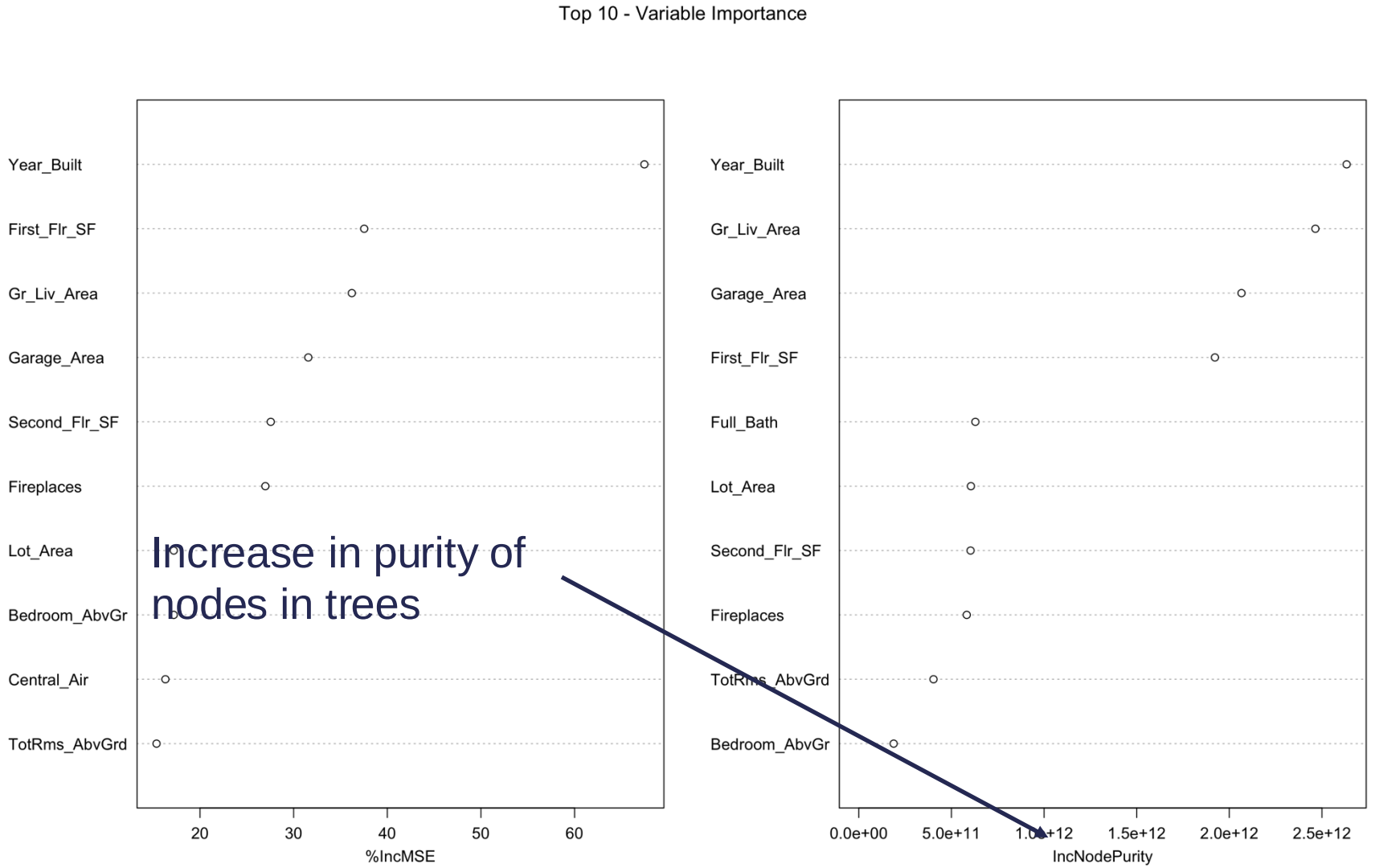
```
varImpPlot(rf.ames,
           sort = TRUE,
           n.var = 10,
           main = "Top 10 -
Variable Importance")
```



Excluded? – the variable is permuted to “remove” relationship with target

Variable Importance

```
varImpPlot(rf.ames,
           sort = TRUE,
           n.var = 10,
           main = "Top 10 -
Variable Importance")
```



Variable Importance

```
importance(rf.ames)
```

```
##                  %IncMSE IncNodePurity
## Bedroom_AbvGr 17.196874  1.883174e+11
## Year_Built    67.454195  2.633279e+12
## Mo_Sold        2.784443  1.851229e+11
## Lot_Area      17.197520  6.052979e+11
## Street         5.653205  4.434557e+09
## Central_Air   16.304359  9.601244e+10
## First_Flr_SF  37.534894  1.922377e+12
## Second_Flr_SF 27.556173  6.032976e+11
## Full_Bath     14.814608  6.292031e+11
## Half_Bath     13.736134  1.042685e+11
## Fireplaces    26.979404  5.827696e+11
## Garage_Area   31.565869  2.065680e+12
## Gr_Liv_Area   36.214537  2.464645e+12
## TotRms_AbvGrd 15.357145  4.033601e+11
```

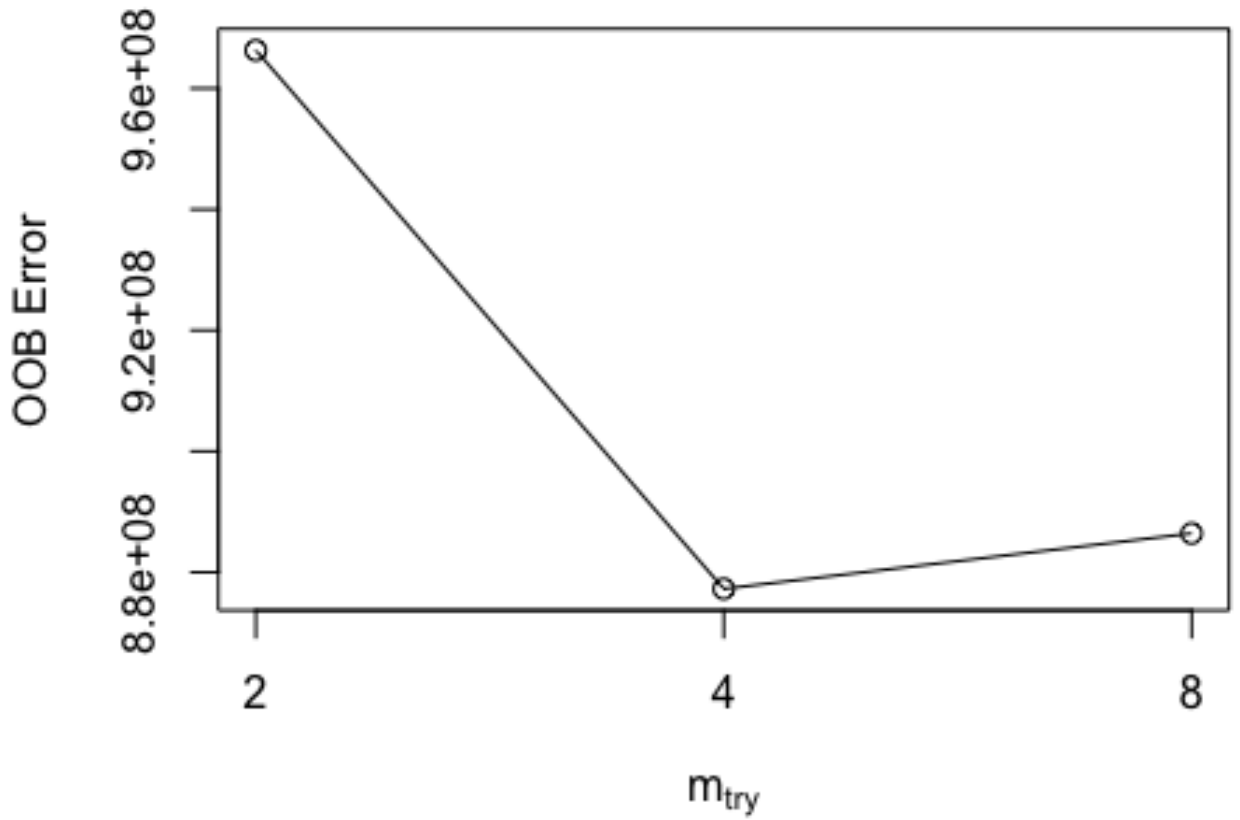
Tuning Random Forests

- The number of variables considered for each split is called *mtry*.
- DEFAULT → *mtry* = $\sqrt{p}$, with p being the number of variables.
- Use validation to tune along with number of trees.

Tuning Random Forests

```
set.seed(12345)
tuneRF(x = training.df[,-1], y = training.df[,1],
       plot = TRUE, ntreeTry = 500, stepFactor = 0.5)
```

```
##   mtry  OOBError
## 2    2 966299090
## 4    4 877257819
## 8    8 886420675
```



Variable Selection

- Random forests use all the variables since they are averaged across all the trees used to build the model.
- Variable selection can be performed by a variety of methods.
  - Many permutations of including/excluding variables → time consuming!
  - Compare variables to random variable → much easier!

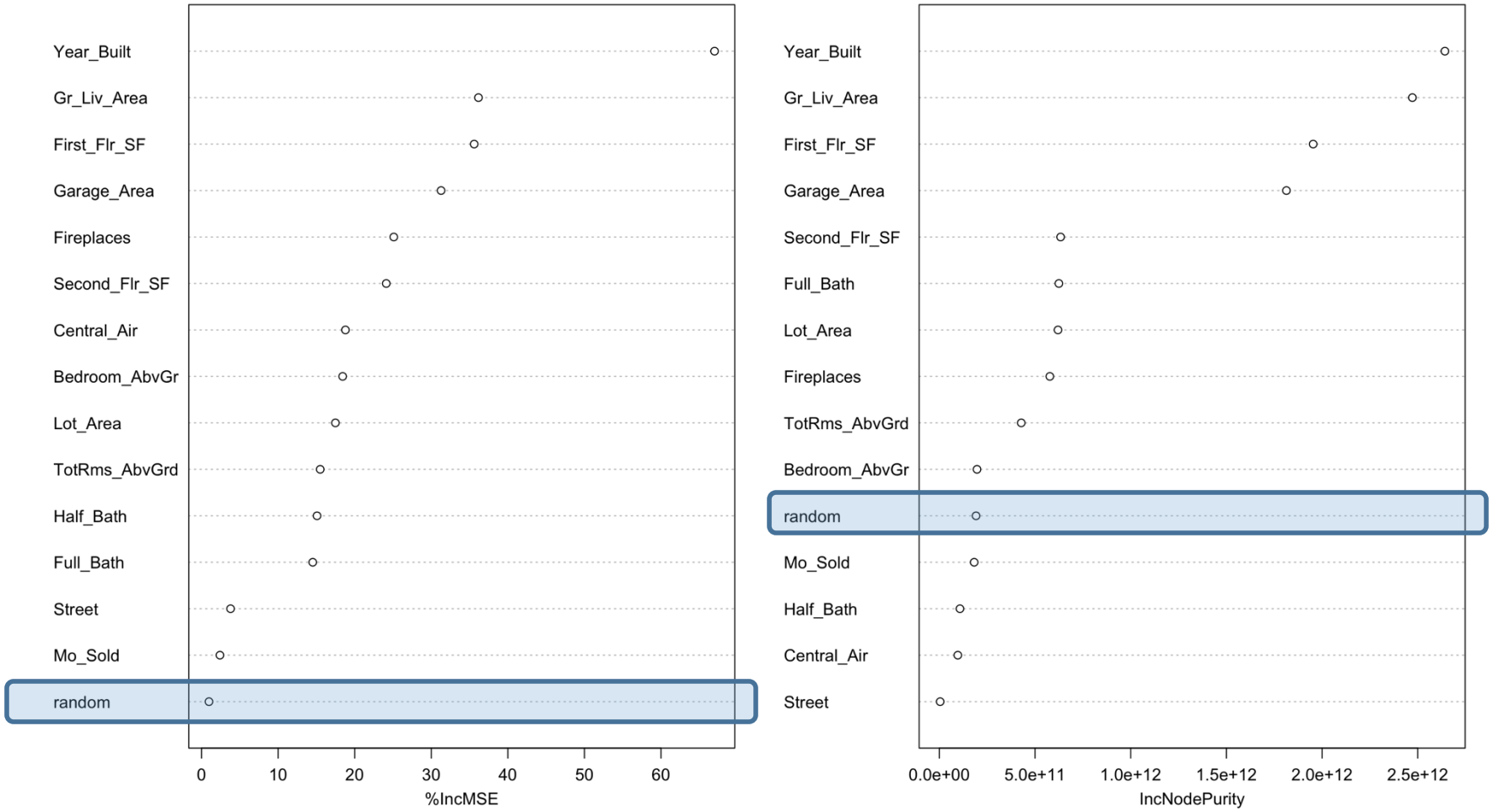
Variable Selection – Random Variable Comparison

```
training.df$random <- rnorm(2051)

set.seed(12345)
rf.ames <- randomForest(Sale_Price ~ ., data = training.df,
ntree = 500, mtry = 4, importance = TRUE)

varImpPlot(rf.ames,
           sort = TRUE,
           n.var = 15,
           main = "Look for Variables Below Random Variable"
)
```

Completely random variable that shouldn't be related to target

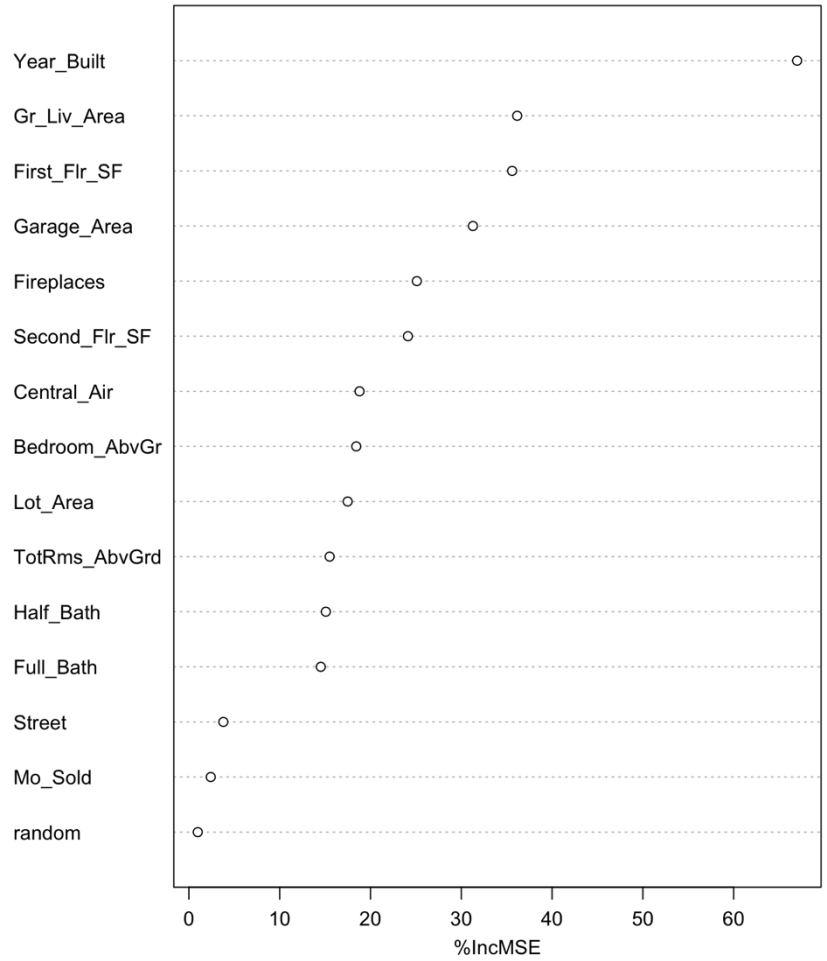
Variable Selection – Random Variable Comparison

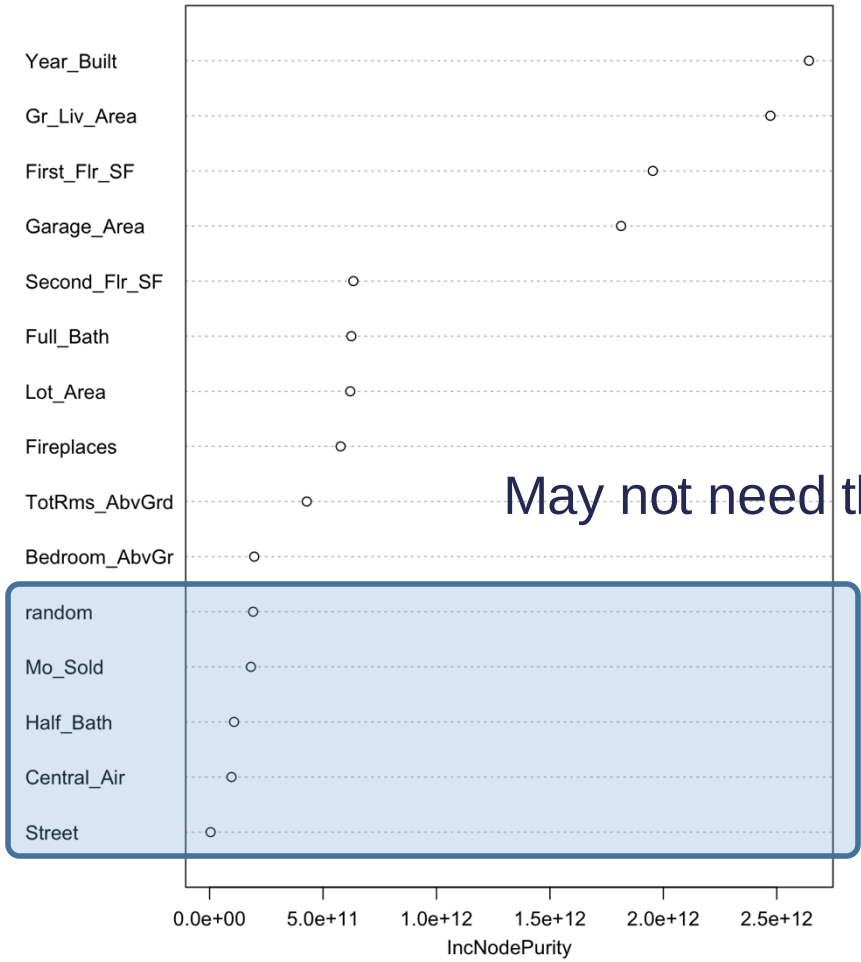
Look for Variables Below Random Variable



Variable Selection – Random Variable Comparison

Look for Variables Below Random Variable





May not need these variables!

Random Forests Summary

Advantages

- Computationally fast (handles thousands of variables)
- Trees trained simultaneously
- Accurate **classification** model
- Variable importance
- Missing data is OK

Disadvantages

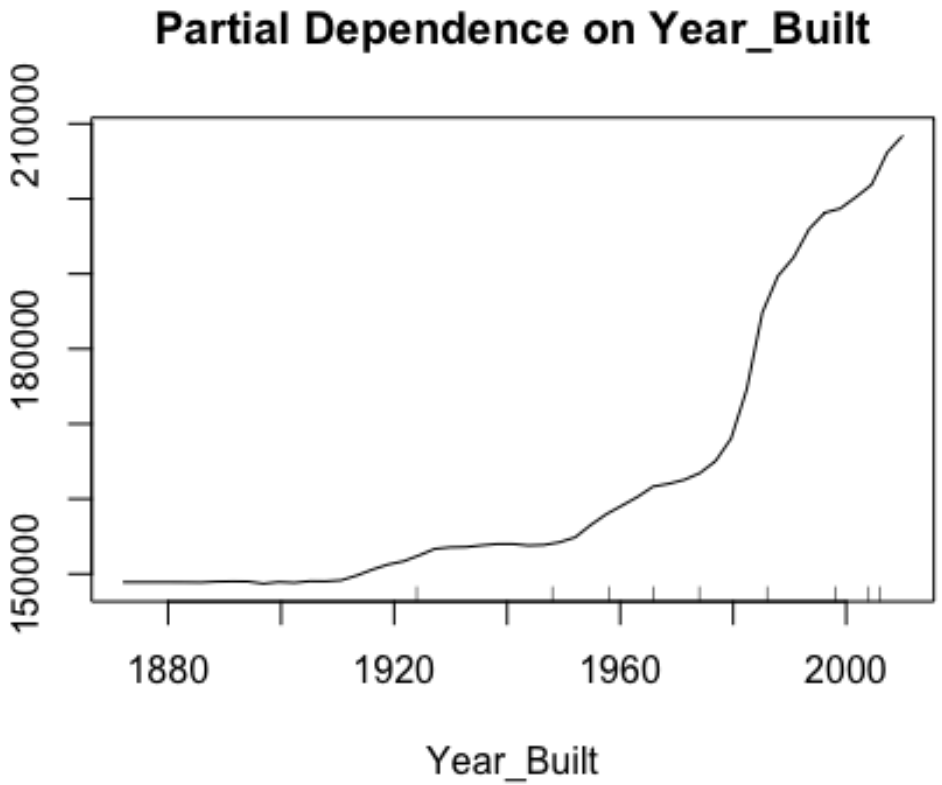
- No “interpretability” other than variable importance
- Tuning parameters

"Interpretable"

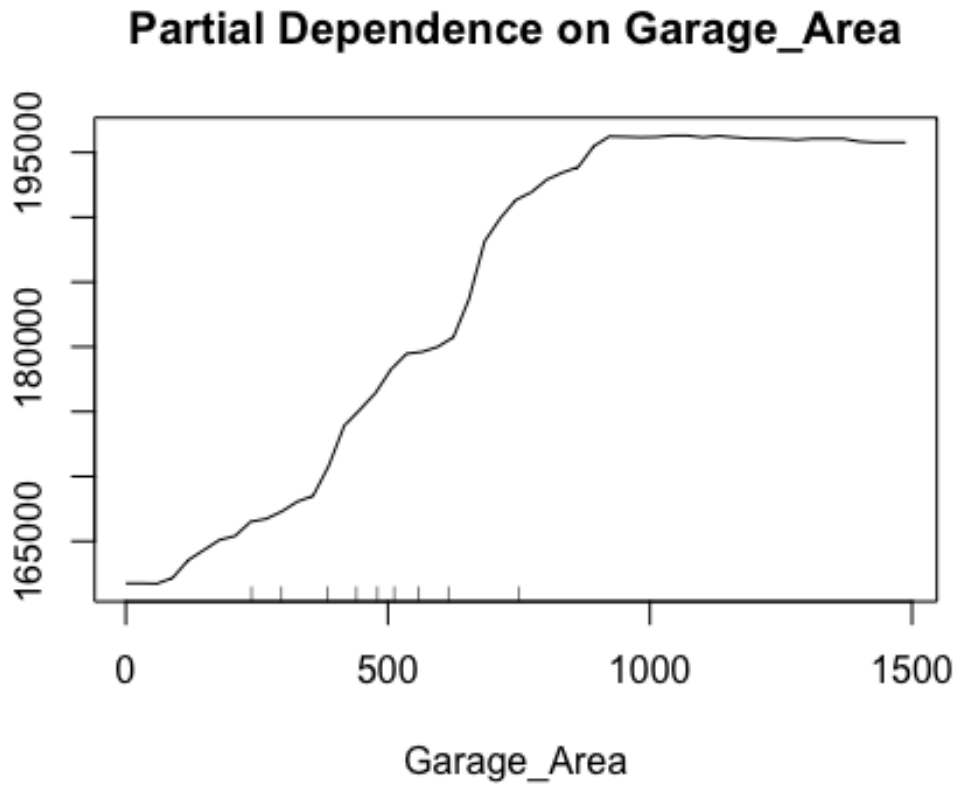
- Most machine learning models are **not** interpretable in the classical sense – as one predictor variable increase, the target variable always does…BLAH.
- This is because the relationships are **not linear**.
- The relationships are more complicated than a linear relationship, so the interpretations are as well.
- Similar to GAM's we can get a general idea of overall pattern for a predictor variable compared to a target variable – **partial dependence plots**.

Partial Dependence Plots

```
partialPlot(rf.ames, training.df, Year_Built)
```




```
partialPlot(rf.ames, training.df, Garage_Area)
```



BOOSTING

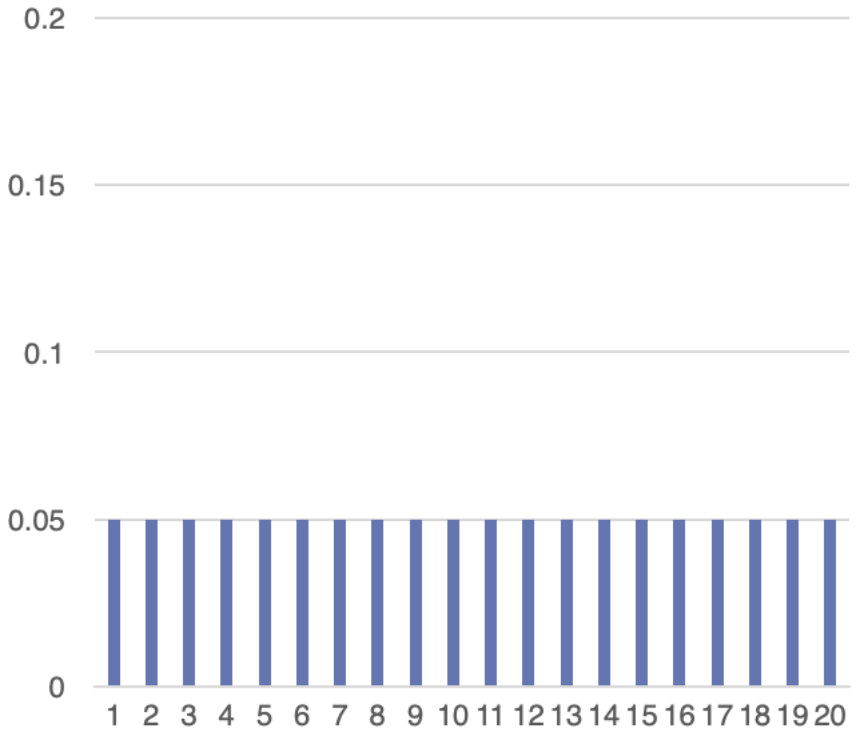
Boosting Overview

- Similar to bagging → draw a sample of observations from dataset with replacement.
- Unlike bagging → observations are **not sampled randomly.**

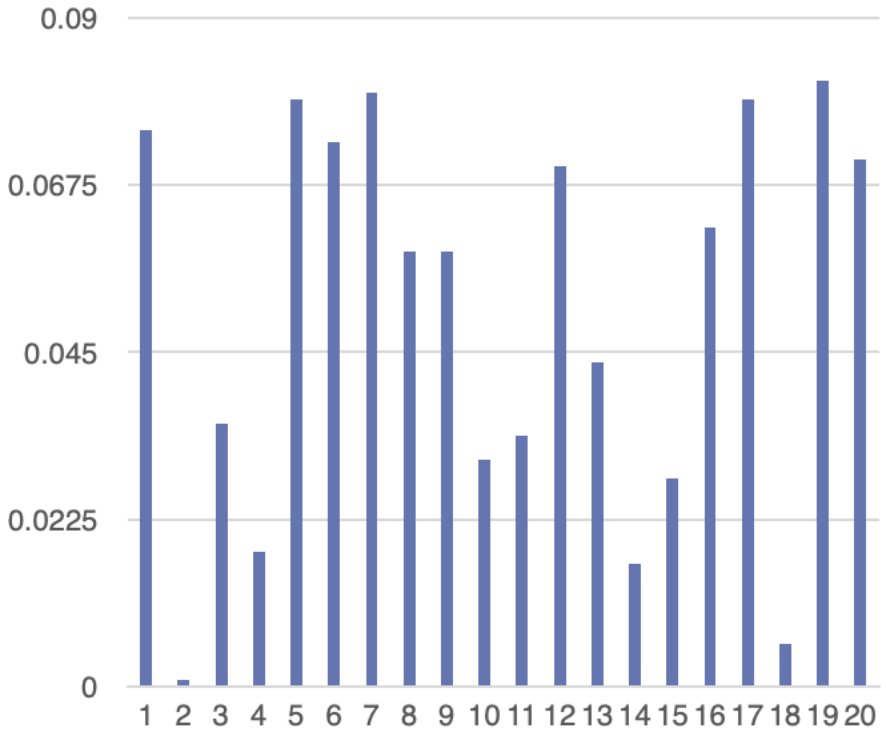
- Boosting assigns weight to each training observation and uses the weight as a sampling distribution.
  - Higher weighted observation is more likely to be drawn.
  - Adaptively change weight in each round.
  - **Weight is higher for observations that are harder to classify.**

Probability of Selecting Observation

Bagging



Boosting



Probability of Selecting Observation

Bagging

- Only trying to create variability in the models by using training dataset variation (bagging).
- Ensemble model built **simultaneously** → no time to evaluate accuracy.

Boosting

- Point with higher sampling probability were harder to predict accurately (boosting).
- Want a chance to improve predictions **sequentially**.

Boosting Example

- 10 observations in original dataset as follows:

X	1	2	3	4	5	6	7	8	9	10
Y	1	1	1	0	0	0	0	1	1	1

- Build a tree with only one split (**decision stump**).
- Start with equal weights for each observation.
- Update weights each round based on the classification error.

First **3** Boosting Rounds

X	1	4	5	6	6	7	7	7	8	10
Y	1	0	0	0	0	0	0	0	1	1

Round Weights	X = 1	X = 2	X = 3	X = 4	X = 5	X = 6	X = 7	X = 8	X = 9	X = 10
1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1

First **3** Boosting Rounds

X	1	4	5	6	6	7	7	7	8	10
Y	1	0	0	0	0	0	0	0	1	1

Round Weights	X = 1	X = 2	X = 3	X = 4	X = 5	X = 6	X = 7	X = 8	X = 9	X = 10
1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
2	0.311	0.311	0.311	0.01	0.01	0.01	0.01	0.01	0.01	0.01

First **3** Boosting Rounds

X	1	4	5	6	6	7	7	7	8	10
Y	1	0	0	0	0	0	0	0	1	1

X	1	1	2	2	2	2	3	3	3	3
Y	1	1	1	1	1	1	1	1	1	1

Round Weights	X = 1	X = 2	X = 3	X = 4	X = 5	X = 6	X = 7	X = 8	X = 9	X = 10
1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
2	0.311	0.311	0.311	0.01	0.01	0.01	0.01	0.01	0.01	0.01
3	0.029	0.029	0.029	0.228	0.228	0.228	0.228	0.009	0.009	0.009

First **3** Boosting Rounds

X	1	4	5	6	6	7	7	7	8	10
Y	1	0	0	0	0	0	0	0	1	1

X	1	1	2	2	2	2	3	3	3	3
Y	1	1	1	1	1	1	1	1	1	1

X	2	2	4	4	4	4	5	6	6	7
Y	1	1	0	0	0	0	0	0	0	0

Round Weights	X = 1	X = 2	X = 3	X = 4	X = 5	X = 6	X = 7	X = 8	X = 9	X = 10
1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
2	0.311	0.311	0.311	0.01	0.01	0.01	0.01	0.01	0.01	0.01
3	0.029	0.029	0.029	0.228	0.228	0.228	0.228	0.009	0.009	0.009

BOOSTING

AdaBoost

Boosted Ensembles (AdaBoost)

- Unlike bagging, boosted ensembles usually weight the votes of each classifier by a function of their accuracy.
- If the classifier gets the higher weighted observations wrong, it has a higher error rate.
- More accurate classifiers get higher weight in the prediction.

- SIMPLE TERMS → More accurate guesses are more important.

Classifier Weights

X	1	4	5	6	6	7	7	7	8	10
Y	1	0	0	0	0	0	0	0	1	1

X	1	1	2	2	2	2	3	3	3	3
Y	1	1	1	1	1	1	1	1	1	1

X	2	2	4	4	4	4	5	6	6	7
Y	1	1	0	0	0	0	0	0	0	0

Error on obs 1, 2, 3

Error on obs 4, 5, 6, 7

Error on obs 8, 9, 10

Round Weights	X = 1	X = 2	X = 3	X = 4	X = 5	X = 6	X = 7	X = 8	X = 9	X = 10
1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
2	0.311	0.311	0.311	0.01	0.01	0.01	0.01	0.01	0.01	0.01
3	0.029	0.029	0.029	0.228	0.228	0.228	0.228	0.009	0.009	0.009

Classifier Weights

X	1	4	5	6	6	7	7	7	8	10
Y	1	0	0	0	0	0	0	0	1	1

X	1	1	2	2	2	2	3	3	3	3
Y	1	1	1	1	1	1	1	1	1	1

X	2	2	4	4	4	4	5	6	6	7
Y	1	1	0	0	0	0	0	0	0	0

Lowest error weights → "Best" model

Round Weights	X = 1	X = 2	X = 3	X = 4	X = 5	X = 6	X = 7	X = 8	X = 9	X = 10
1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
2	0.311	0.311	0.311	0.01	0.01	0.01	0.01	0.01	0.01	0.01
3	0.029	0.029	0.029	0.228	0.228	0.228	0.228	0.009	0.009	0.009

Classifier Weights

- Observations are each weighted based on how well they were predicted in the previous round with decision tree.
- Let the weights for each round be denoted as ω_i and the predictions for round 1, 2, etc. as $\hat{y}_{1,i}$, $\hat{y}_{2,i}$, etc.
- The prediction for each observation is derived from a classification as follows:

$$\hat{y}_i = \omega_1 \hat{y}_{1,i} + \omega_2 \hat{y}_{2,i} + \cdots$$

- Weight calculations are not detailed here…

GRADIENT BOOSTING

Gradient Boosting Machine

- Build a simple model to predict target (VERY SIMPLE, not trying to be anything close to perfect):

$$y = f_1(x) + \varepsilon_1$$

Gradient Boosting Machine

- Build a simple model to predict target (VERY SIMPLE, not trying to be anything close to perfect):

$$y = f_1(x) + \varepsilon_1$$

- Model has error, ε_1. What if we tried to **predict that error** with another simple model (AGAIN, SIMPLE)?

$$\varepsilon_1 = f_2(x) + \varepsilon_2$$

Gradient Boosting Machine

- Build a simple model to predict target (VERY SIMPLE, not trying to be anything close to perfect):

$$y = f_1(x) + \varepsilon_1$$

- Model has error, ε_1. What if we tried to **predict that error** with another simple model (AGAIN, SIMPLE)?

$$\varepsilon_1 = f_2(x) + \varepsilon_2$$

- This model has error too…

Gradient Boosting Machine

- Continue to add model after model, each one predicting the error (residuals) from the previous round…

$$y = f_1(x) + f_2(x) + f_3(x) + \cdots + f_k(x) + \varepsilon_k$$

Original model

Predict error from model 1

Predict error from model 2

Gradient Boosting Machine

- Continue to add model after model, each one predicting the error (residuals) from the previous round…

$$y = f_1(x) + f_2(x) + f_3(x) + \cdots + f_k(x) + \varepsilon_k$$

Original model

Predict error from model 1

Predict error from model 2

- Do this until there is really small error → OVERFITTING!!!

Overfitting Protection

- Gradient boosting regularization through parameters to tune to prevent overfitting.
  1. Reduce weight of each of the error models for prediction:

$$y = f_1(x) + \eta * f_2(x) + \eta * f_3(x) + \cdots + \eta * f_k(x) + \varepsilon_k$$

     - Smaller values of η lead to less overfitting.

  2. Number of trees (classifiers) used in the prediction
     - Larger number of trees → more prone to overfitting.

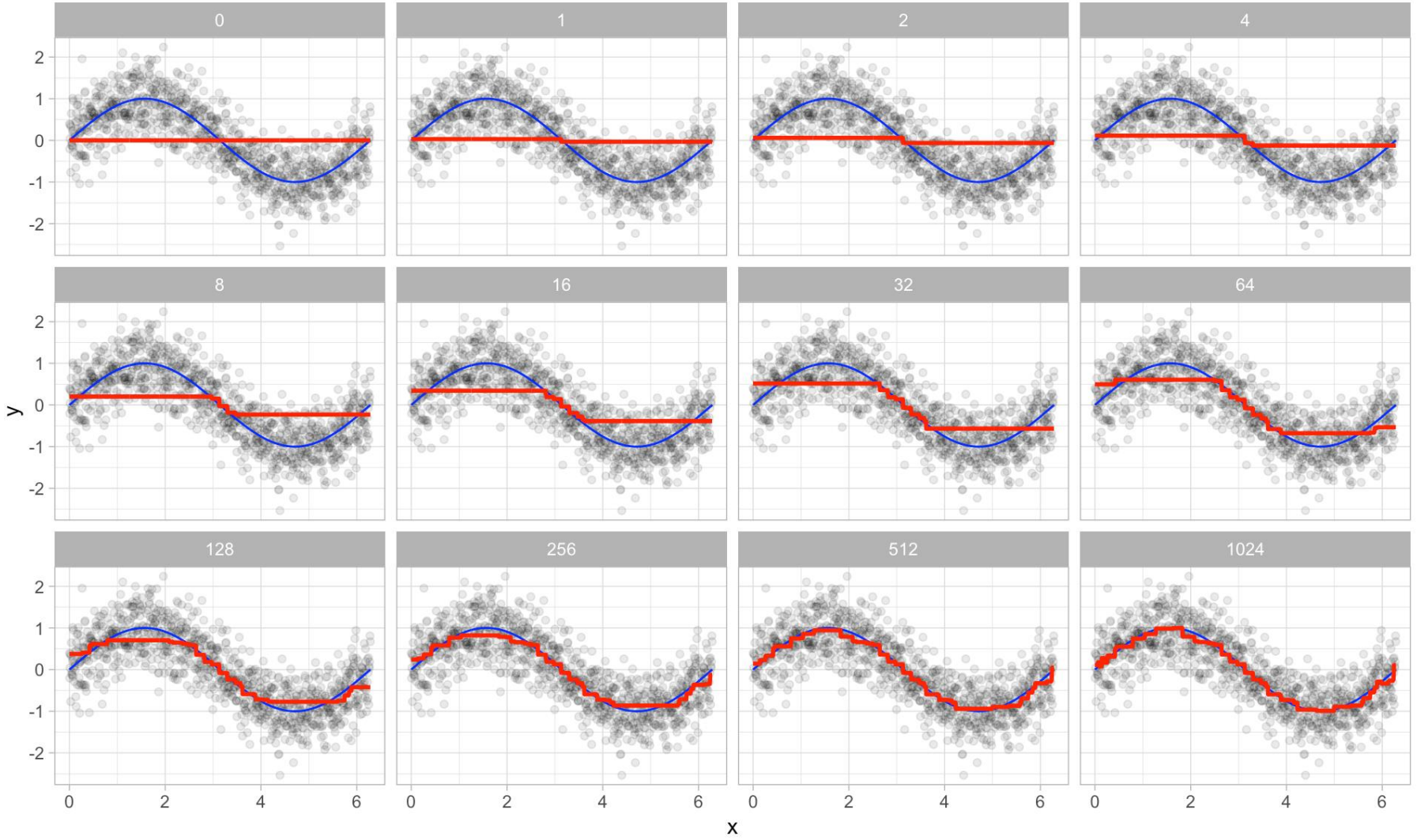
  3. Other parameters introduced over years to help (λ, γ, L_2)…

Gradient Boosted Trees

- Gradient boosting yields an **additive ensemble model**:
  - No voting or averaging of individual models.
  - Predictions from each model are summed together for final prediction.

- Key to gradient boosting is **using weak learners**:
  - SIMPLE MODELS! Shallow regression/decision trees are the best.
  - Each of these simple models would make poor predictions on their own, but additive fashion of ensemble provides really good predictions.

Ensemble of Weak Learners



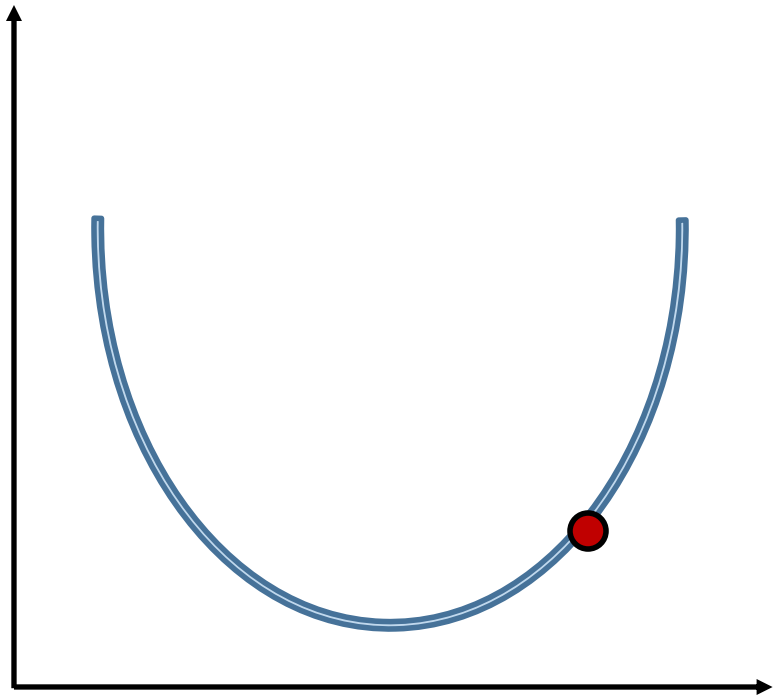
Training the Model (All Those Trees!)

- Most models are optimized to some form of a **loss function**.
- For example, linear regression and decision trees typically look at minimizing the SSE – our loss function.
- The SSE aggregates errors from many predictions into a single number – the **loss** of the model.
- Can use many different loss functions!

- How do we find the model with the lowest loss function (lowest error)?

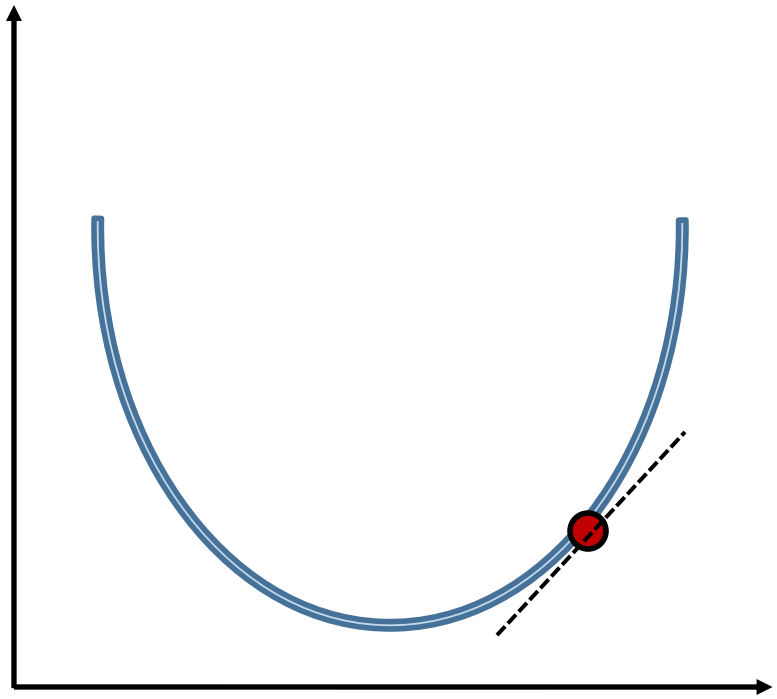
Gradient Descent

- Gradient descent is a method that iteratively updates parameters in order to minimize a loss function (the errors) by moving in the direction of steepest descent.
- Similar in idea to maximizing the likelihood function.



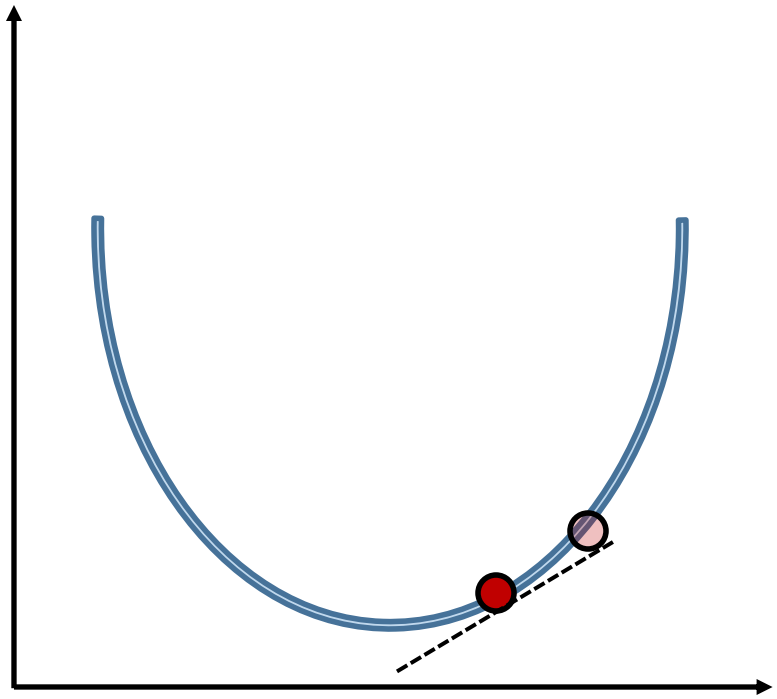
Gradient Descent

- Minimizes the loss function by taking iteratively smaller steps until it finds the minimum.



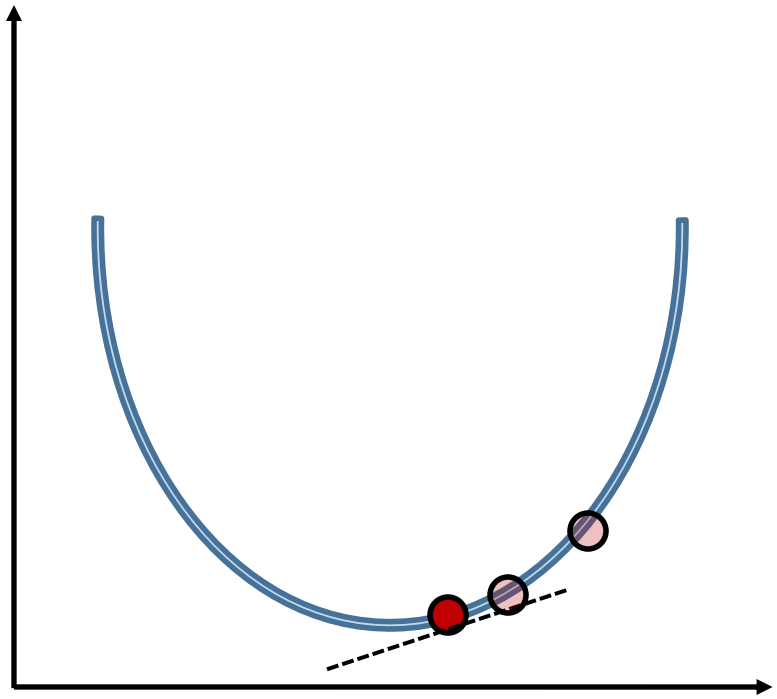
Gradient Descent

- Minimizes the loss function by taking iteratively smaller steps until it finds the minimum.



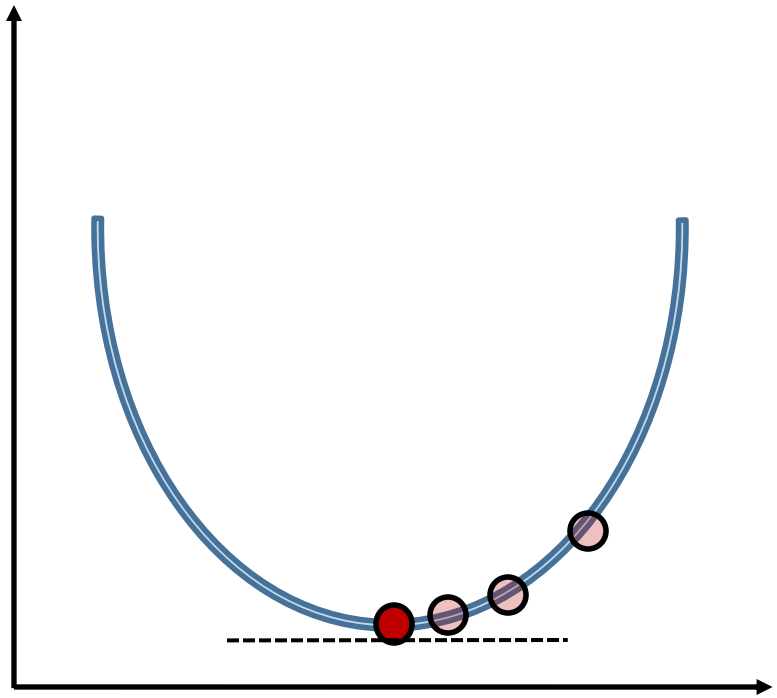
Gradient Descent

- Minimizes the loss function by taking iteratively smaller steps until it finds the minimum.



Gradient Descent

- The step size is updated at each step to help with the minimization – **learning rate**.
- Without the learning rate, we might take steps too big (miss the minimum) or too small (too long to optimize).



Stochastic Gradient Descent

- Not all loss functions are convex → some have local minima or plateaus that make finding the global minimum difficult.
- **Stochastic gradient descent** attempts to solve this by randomly sampling a fraction of the training observations for each tree in the ensemble.
- This makes the algorithm faster and more reliable, but may not always find the true overall minimum.

Training a Gradient Boosted Machine

- Grid search is VERY time consuming because of the time it takes to build these models.
- Tune parameters one at a time:
  1. Start with relatively high learning rate (default of 0.1 is typically good).
  2. Determine optimal number of trees for this learning rate.
  3. Fix tree tuning parameters (number of trees, depth, etc.) and tune learning rate.
  4. Set learning rate again at this new value and re-tune tree parameters.
  5. Try lowering learning rate again to see if any improvements.

More than One Way to Gradient Boost

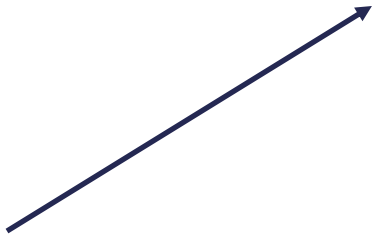
- Many different adaptations to the gradient boosting approach
  - XGBoost
  - LightGBM
  - CatBoost
  - AdaBoost (already covered)
  - Etc.

Extreme Gradient Boosting (XGBoost)

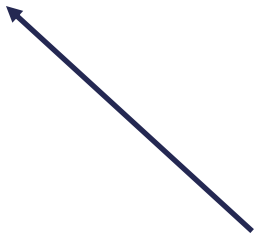
- Advantages over traditional GBM:
  1. Additional regularization parameters to prevent overfitting (but that means more tuning ☹).
  2. Settings to stop model assessment when adding more trees isn't helpful.
  3. Supports parallel processing, but still must be trained sequentially.
  4. Variety of loss functions.
  5. Allows generalized linear models as well as tree-based models (all still weak learners though).
  6. Implemented in R, Python, Julia, Scala, Java, C++.

XGBoost

```
train_x <- model.matrix(Sale_Price ~ ., data = training)[, -1]
train_y <- training$Sale_Price

set.seed(12345)
xgb.ames <- xgboost(data = train_x, label = train_y, subsample = 0.5, nrounds = 100)
```



Stochastic – doesn't use full sample (here random 50% of the training for each tree)



Number of trees in the boosting

Tuning the XGBoost

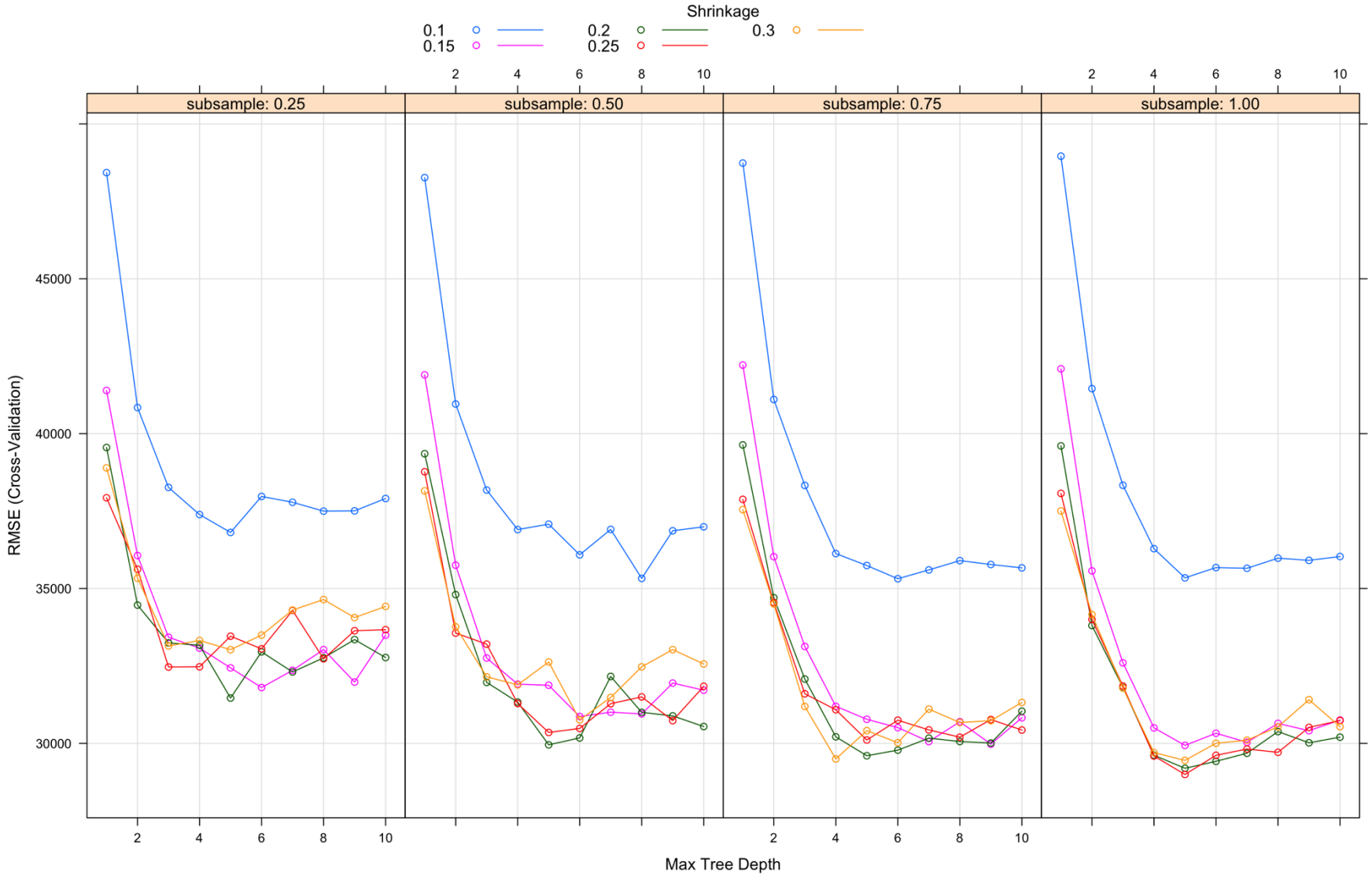
```
xgbcv.ames <- xgb.cv(data = train_x, label = train_y, subsample = 0.5, nrounds = 100, nfold = 10)
```

```
## [1]  train-rmse:141922.035938+724.636846 test-rmse:142049.065625+6419.837344
## [2]  train-rmse:103394.505469+758.538993 test-rmse:103990.367969+5821.524755
...
## [24] train-rmse:17579.951758+747.728455  test-rmse:31767.950000+4192.899902
...
## [99] train-rmse:5963.429492+376.358716   test-rmse:32549.597461+4221.951431
## [100]    train-rmse:5898.152881+370.517169   test-rmse:32570.780469+4237.752213
```

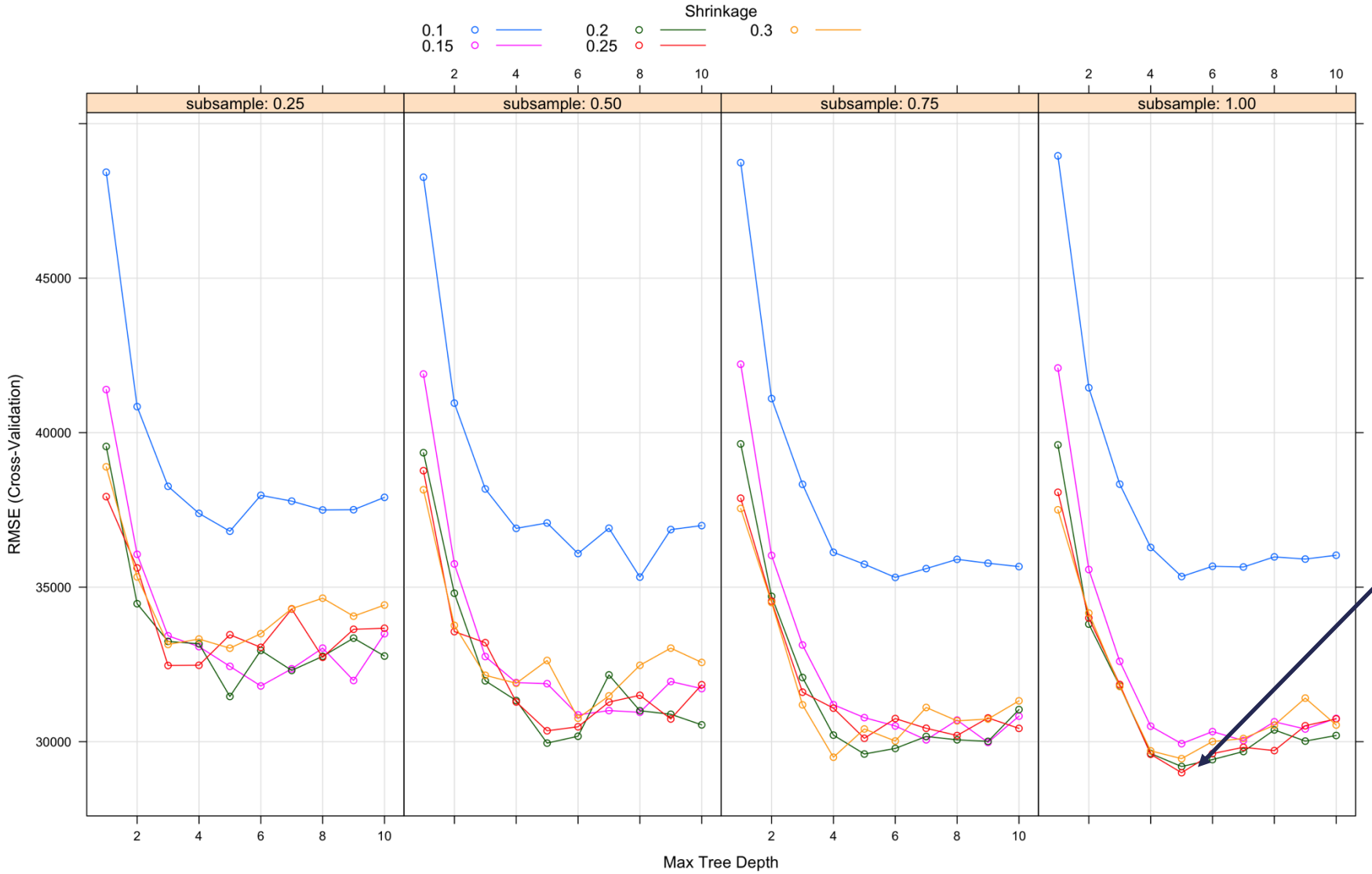
Tuning the XGBoost – Grid Search with caret

```
tune_grid <- expand.grid(
  nrounds = 24,
  eta = c(0.1, 0.15, 0.2, 0.25, 0.3),
  max_depth = c(1:10),
  gamma = c(0),
  colsample_bytree = 1,
  min_child_weight = 1,
  subsample = c(0.25, 0.5, 0.75, 1)
)

xgb.ames.caret <- train(x = train_x, y = train_y,
      method = "xgbTree",
      tuneGrid = tune_grid,
      trControl = trainControl(method = 'cv', number = 10))

plot(xgb.ames.caret)
```

Tuning the XGBoost – Grid Search with `caret`

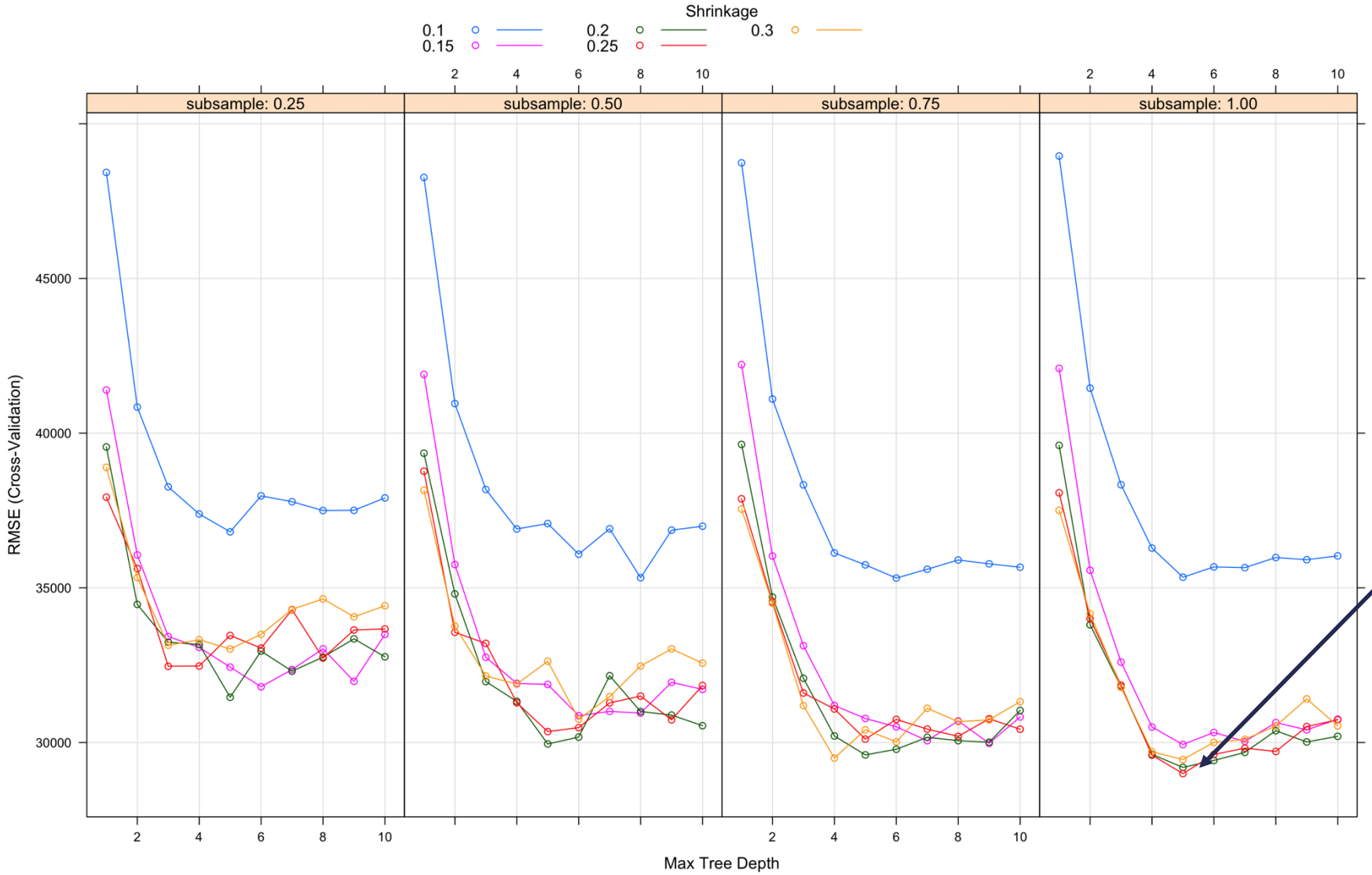


Tuning the XGBoost – Grid Search with `caret`



Seems to be optimized here!

Tuning the XGBoost – Grid Search with `caret`



Seems to be optimized here!
Max tree depth = 5
Subsample = 100%
η = 0.25

Variable Importance in XGBoost

- XGBoost provides 3 built-in measures of variable importance:
  1. **Gain** – improvement in accuracy before and after a variable is used on a branch.
  2. **Coverage** – measures the relative number of observations influenced by the variable.
  3. **Frequency** – percentage of splits in the whole ensemble that use this variable.

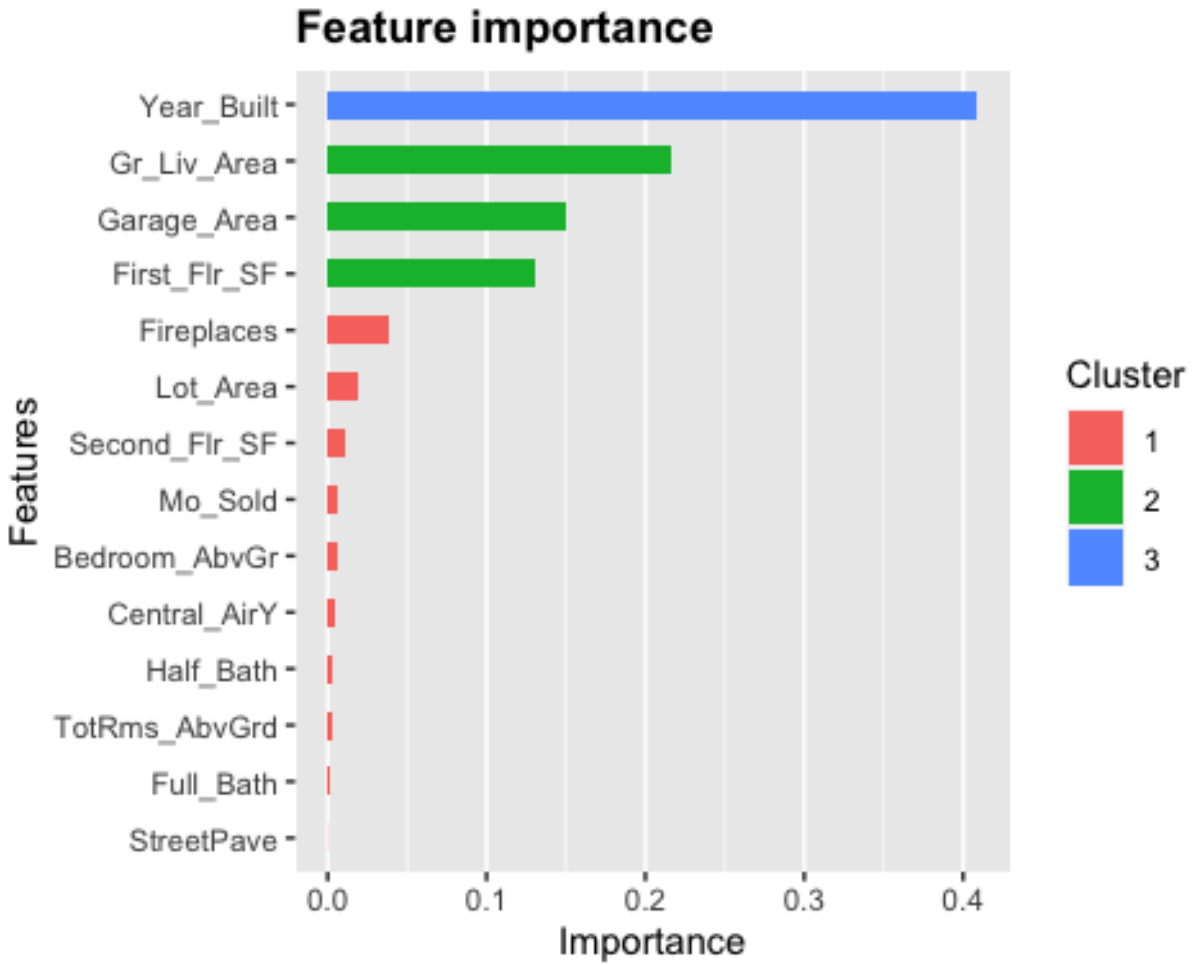
Variable Importance in XGBoost

```
set.seed(12345)
xgb.ames <- xgboost(data = train_x, label = train_y, subsample = 1, nrounds = 24, eta = 0.25, max_depth = 5)

xgb.importance(feature_names = colnames(train_x), model = xgb.ames)
```

```
##           Feature         Gain       Cover   Frequency
##  1:    Year_Built 0.4067195058 0.183708803 0.140819964
##  2:   Gr_Liv_Area 0.2179077573 0.192822988 0.121212121
##  3:   Garage_Area 0.1492481431 0.094004320 0.110516934
##  4:  First_Flr_SF 0.1288022057 0.143082897 0.149732620
##  5:    Fireplaces 0.0380211449 0.046918450 0.033868093
##  6:      Lot_Area 0.0187551683 0.146431299 0.128342246
##  7: Second_Flr_SF 0.0097506768 0.043431213 0.055258467
##  8:        random 0.0069491958 0.008125998 0.069518717
##  9:       Mo_Sold 0.0054516126 0.008260751 0.051693405
## 10: Bedroom_AbvGr 0.0052478584 0.049740091 0.053475936
## 11:  Central_AirY 0.0046903747 0.018367205 0.012477718
## 12: TotRms_AbvGrd 0.0037997263 0.015582316 0.030303030
## 13:     Half_Bath 0.0034109121 0.025202844 0.026737968
## 14:     Full_Bath 0.0009226260 0.005692282 0.010695187
## 15:    StreetPave 0.0003230922 0.018628544 0.005347594
```

Variable Importance in XGBoost

```
xgb.ggplot.importance(xgb.importance(feature_names = colnames(train_x), model = xgb.ames))
```



Variable Selection

- XGBoost uses all the variables since they are averaged across all the trees used to build the model.
- Variable selection can be performed by a variety of methods.
  - Many permutations of including/excluding variables → time consuming!
  - Compare variables to random variable → much easier!

Variable Selection – Random Variable Comparison

```
training$random <- rnorm(2051)

train_x <- model.matrix(Sale_Price ~ ., data = training)[, -1]
train_y <- training$Sale_Price

set.seed(12345)
xgb.ames <- xgboost(data = train_x, label = train_y, subsample = 1, nrounds = 24, eta = 0.25, max_depth = 5)
```

Completely random variable that shouldn't be related to target

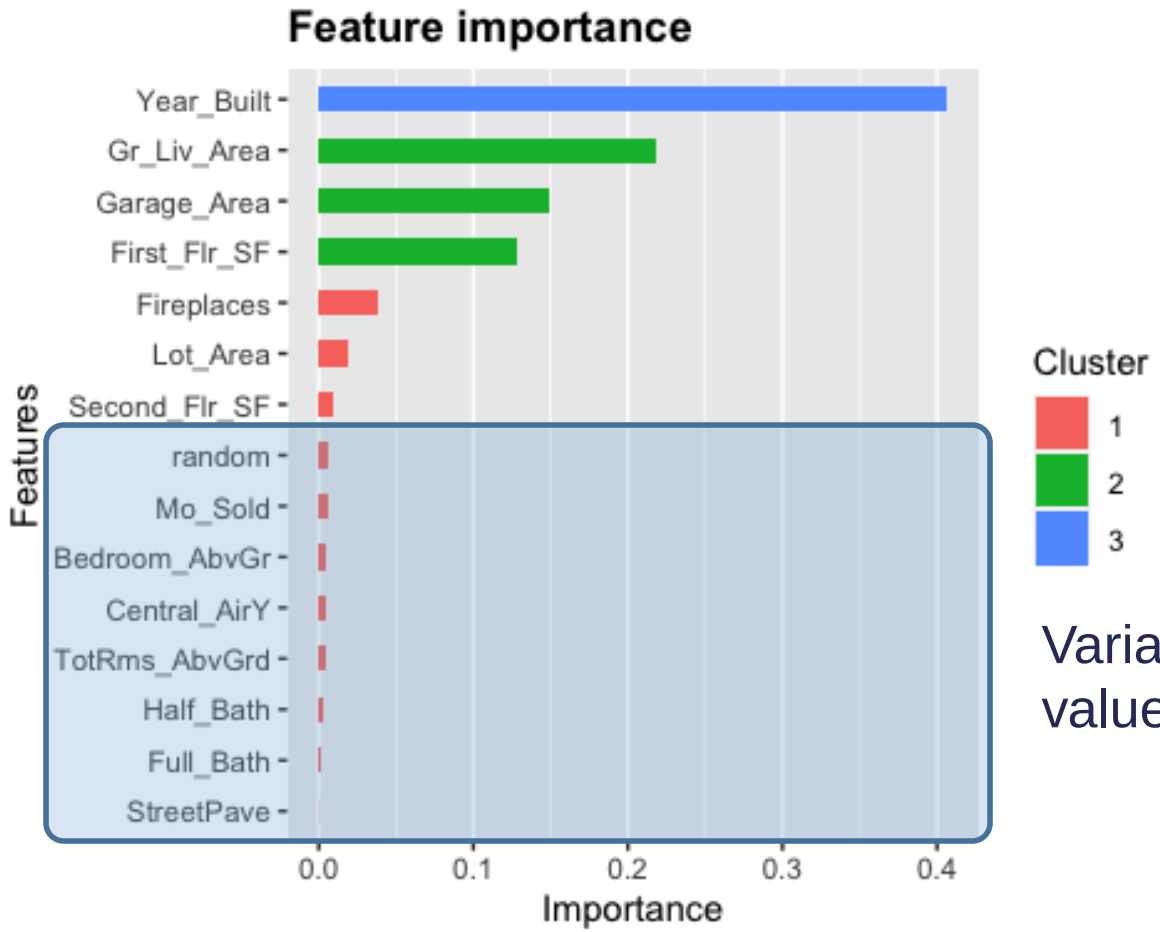
Variable Selection – Random Variable Comparison

```
xgb.importance(feature_names = colnames(train_x), model = xgb.ames)
```

```
##           Feature         Gain       Cover   Frequency
##  1:    Year_Built 0.4067195058 0.183708803 0.140819964
##  2:   Gr_Liv_Area 0.2179077573 0.192822988 0.121212121
##  3:   Garage_Area 0.1492481431 0.094004320 0.110516934
##  4:  First_Flr_SF 0.1288022057 0.143082897 0.149732620
##  5:    Fireplaces 0.0380211449 0.046918450 0.033868093
##  6:      Lot_Area 0.0187551683 0.146431299 0.128342246
##  7: Second_Flr_SF 0.0097506768 0.043431213 0.055258467
##  8:        random 0.0069491958 0.008125998 0.069518717
##  9:       Mo_Sold 0.0054516126 0.008260751 0.051693405
## 10: Bedroom_AbvGr 0.0052478584 0.049740091 0.053475936
## 11:  Central_AirY 0.0046903747 0.018367205 0.012477718
## 12: TotRms_AbvGrd 0.0037997263 0.015582316 0.030303030
## 13:     Half_Bath 0.0034109121 0.025202844 0.026737968
## 14:     Full_Bath 0.0009226260 0.005692282 0.010695187
## 15:    StreetPave 0.0003230922 0.018628544 0.005347594
```

Variable Selection – Random Variable Comparison

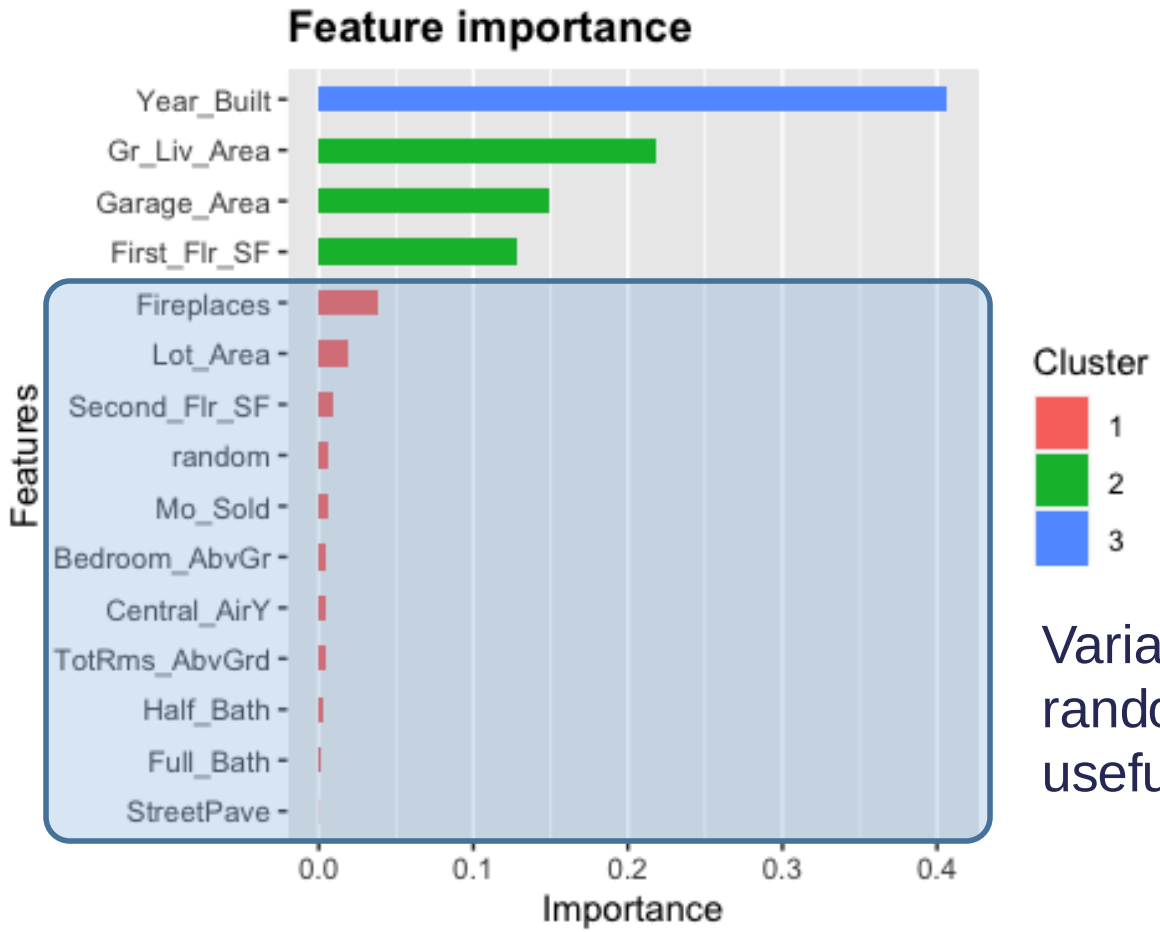
```
xgb.ggplot.importance(xgb.importance(feature_names = colnames(train_x), model = xgb.ames))
```



Variables **below** random in value might not be useful.

Variable Selection – Random Variable Comparison

```
xgb.ggplot.importance(xgb.importance(feature_names = colnames(train_x), model = xgb.ames))
```



Variables **clustered with** random in value might not be useful.

Gradient Boosting Summary

Advantages

- Very accurate.
- Tend to outperform random forests when properly trained and tuned.
- Variable importance provided.

Disadvantages

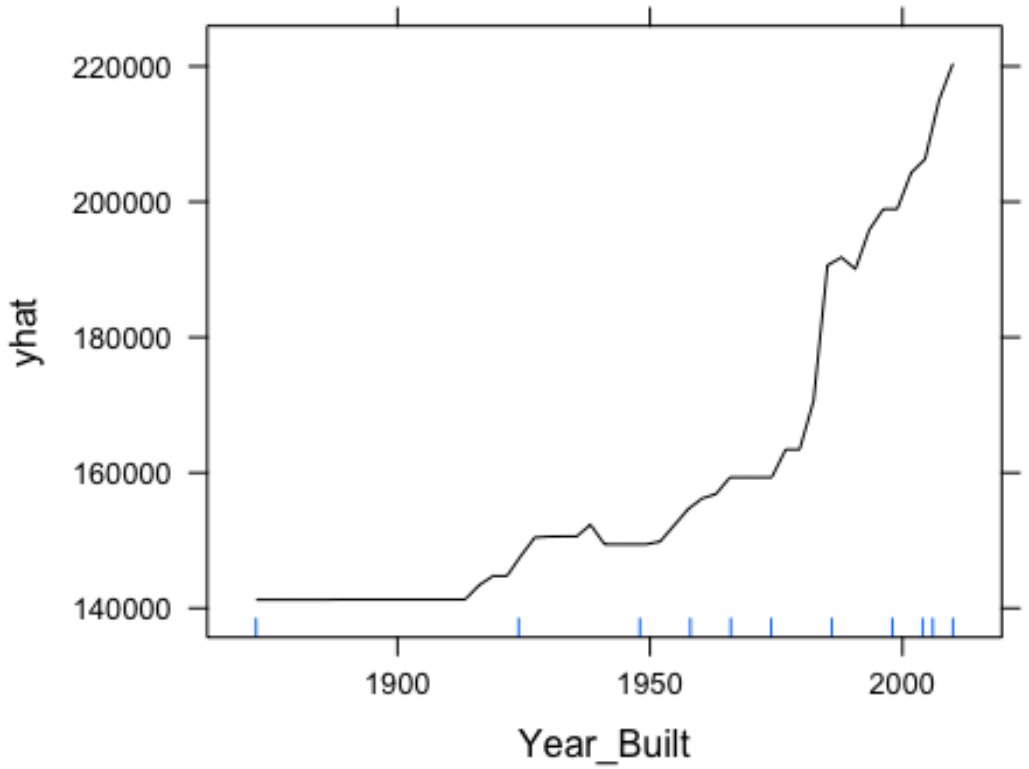
- Lacks “interpretability” beyond variable importance.
- Computationally **slower** than random forests → building sequentially.
- More tuning parameters than random forest.
- Harder to optimize.
- More sensitive to tuning parameters.

"Interpretable"

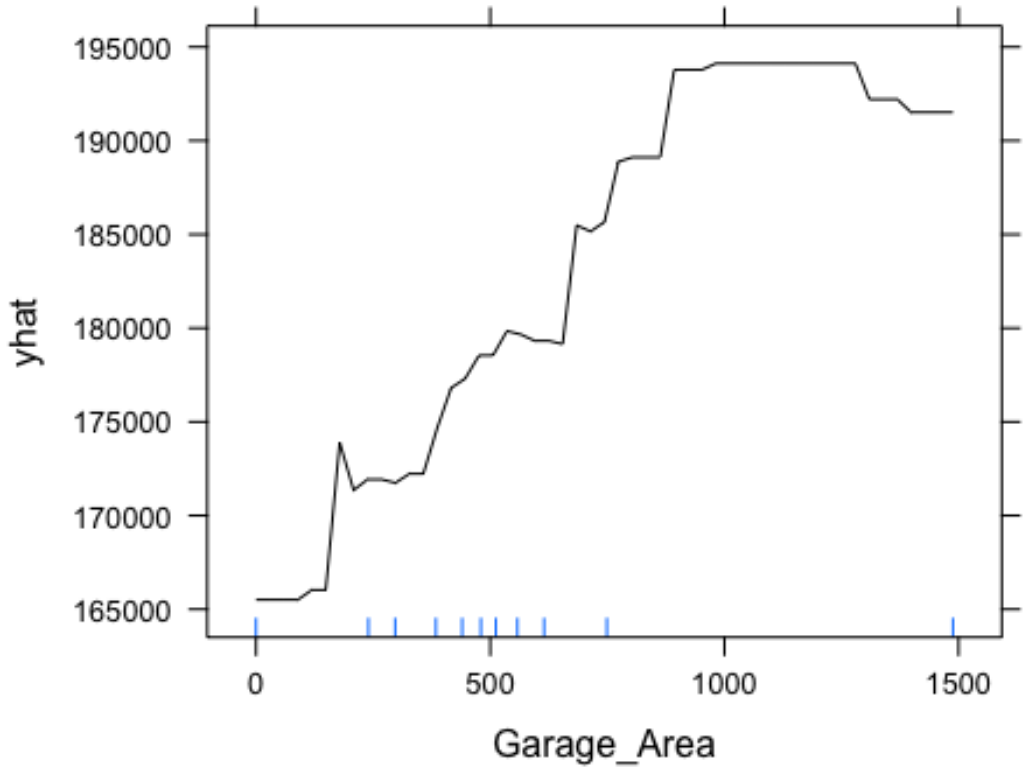
- Most machine learning models are **not** interpretable in the classical sense – as one predictor variable increase, the target variable always does…BLAH.
- This is because the relationships are **not linear**.
- The relationships are more complicated than a linear relationship, so the interpretations are as well.
- Similar to GAM's we can get a general idea of overall pattern for a predictor variable compared to a target variable – **partial dependence plots**.

Partial Dependence Plots

```
partial(xgb.ames, pred.var = "Year_Built",
        plot = TRUE, rug = TRUE, alpha = 0.1,
        plot.engine = "lattice", train = train_x)
```




```
partial(xgb.ames, pred.var = "Garage_Area",
        plot = TRUE, rug = TRUE, alpha = 0.1,
        plot.engine = "lattice", train = train_x)
```



EXPLAINABLE BOOSTING MACHINE (EBM)

Generalized Additive Models (GAMs)

- Provides **general** framework for **adding** of non-linear functions together instead of the typical linear structure.

$$y = \beta_0 + f_1(x_1) + f_2(x_2) + \cdots + f_p(x_p) + \varepsilon$$

- Can be used for regression or classification problems.

GAMs into EBMs

- Provides **general** framework for **adding** of non-linear functions together instead of the typical linear structure.

$$y = \beta_0 + f_1(x_1) + f_2(x_2) + \cdots + f_p(x_p) + \varepsilon$$

- Can be used for regression or classification problems.

EBMs use machine learning algorithms (like Random Forests and Boosting Trees) to build these individual pieces before adding together.

GBM – One Variable at a Time

- Build a simple model **using only one variable** to predict target (VERY SIMPLE, not trying to be anything close to perfect):

$$y = g_1(x_1) + \varepsilon_1$$

- Model has error, ε_1. What if we tried to predict that error with another simple model **using only one variable that wasn't previously used** (AGAIN, SIMPLE)?

$$\varepsilon_1 = g_2(x_2) + \varepsilon_2$$

- This model has error too…

GBM – One Variable at a Time

- Continue to add model after model, each one predicting the error (residuals) from the previous round…

$$y = g_1(x_1) + g_2(x_2) + g_3(x_3) + \cdots + g_k(x_k) + \varepsilon_k$$

Original model using x_1 only

Predict error from model 1 using x_2 only

Predict error from model 2 using x_3 only

REPEAT – Round Robin Approach to Variables

- Continue to add model after model, each one predicting the error (residuals) from the previous round…

$$y = g_1(x_1) + g_2(x_2) + g_3(x_3) + \cdots + g_k(x_k) + \varepsilon_k$$

$$y = g_2(x_2) + g_1(x_1) + g_3(x_3) + \cdots + g_k(x_k) + \varepsilon_k$$

$$y = g_2(x_2) + g_3(x_3) + g_1(x_1) + \cdots + g_k(x_k) + \varepsilon_k$$

$$\vdots$$

Thousands of iterations (like random forest approach)

REPEAT – Round Robin Approach to Variables

- Continue to add model after model, each one predicting the error (residuals) from the previous round…

$$y = g_1(x_1) + g_2(x_2) + g_3(x_3) + \cdots + g_k(x_k) + \varepsilon_k$$

$$y = g_2(x_2) + g_1(x_1) + g_3(x_3) + \cdots + g_k(x_k) + \varepsilon_k$$

$$y = g_2(x_2) + g_3(x_3) + g_1(x_1) + \cdots + g_k(x_k) + \varepsilon_k$$

$$\vdots$$

Aggregate all models for variable x_1 to estimate relationship between y and x_1 – GAM $f_1(x_1)$

GAMs into EBMs

- Provides **general** framework for **adding** of non-linear functions together instead of the typical linear structure.

$$y = \beta_0 + f_1(x_1) + f_2(x_2) + \cdots + f_p(x_p) + \varepsilon$$

- Can be used for regression or classification problems.

Aggregate all models for variable x_1 to estimate relationship between y and x_1 – GAM $f_1(x_1)$

REPEAT – Round Robin Approach to Variables

- Continue to add model after model, each one predicting the error (residuals) from the previous round…

$$y = g_1(x_1) + g_2(x_2) + g_3(x_3) + \cdots + g_k(x_k) + \varepsilon_k$$

$$y = g_2(x_2) + g_1(x_1) + g_3(x_3) + \cdots + g_k(x_k) + \varepsilon_k$$

$$y = g_2(x_2) + g_3(x_3) + g_1(x_1) + \cdots + g_k(x_k) + \varepsilon_k$$

$$\vdots$$

Aggregate all models for variable x_2 to estimate relationship between y and x_2 – GAM $f_2(x_2)$

GAMs into EBMs

- Provides **general** framework for **adding** of non-linear functions together instead of the typical linear structure.

$$y = \beta_0 + f_1(x_1) + f_2(x_2) + \cdots + f_p(x_p) + \varepsilon$$

- Can be used for regression or classification problems.

Aggregate all models for variable x_2 to estimate relationship between y and x_2 – GAM $f_2(x_2)$

GAMs into EBMs

- Provides **general** framework for **adding** of non-linear functions together instead of the typical linear structure.

$$y = \beta_0 + f_1(x_1) + f_2(x_2) + \cdots + f_p(x_p) + \varepsilon$$

- Can be used for regression or classification problems.

EBMs use machine learning algorithms (like Random Forests and Boosting Trees) to build these individual pieces before adding together.

Explainable Boosting Machine Summary

Advantages

- Very accurate.
- Early results show they are powerful at predicting (almost to level of XGBoost).
- Interpretation available due to GAM nature (individually estimated relationships – think PDP).

Disadvantages

- Computationally **slower** than random forests → building sequentially.
- More tuning parameters than random forest.
- Best implementation is currently only in Python.