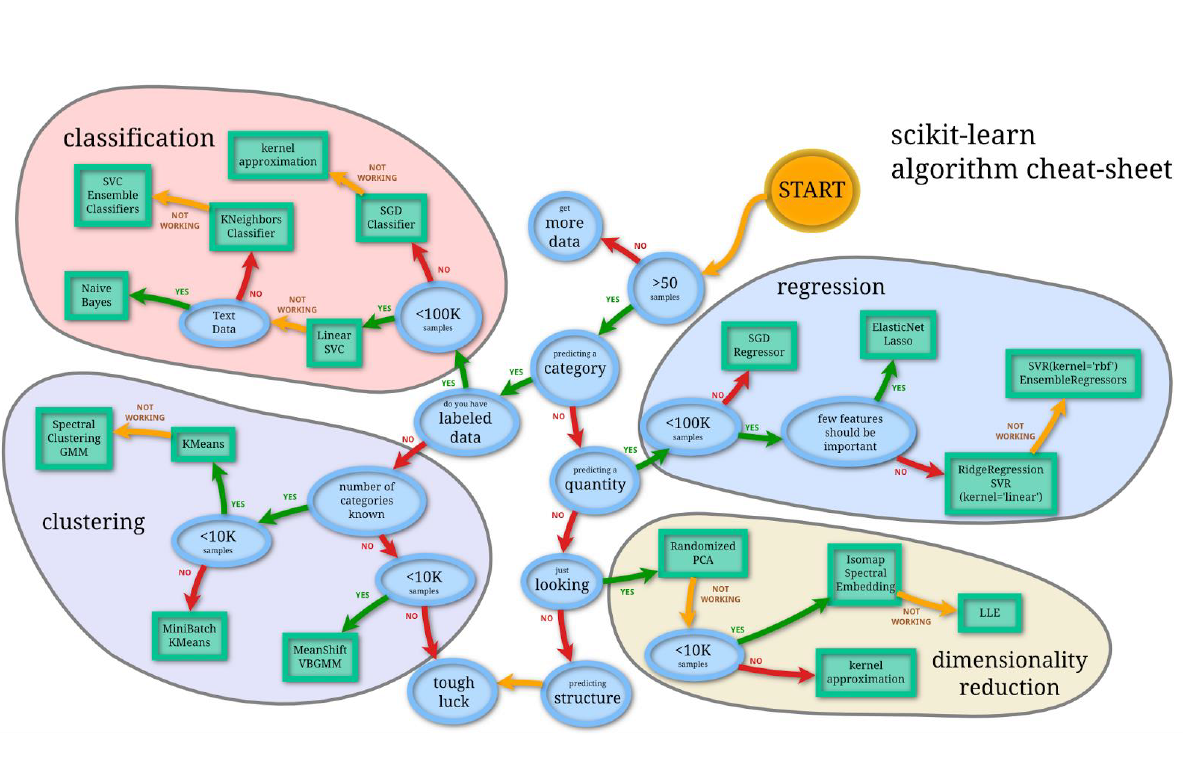
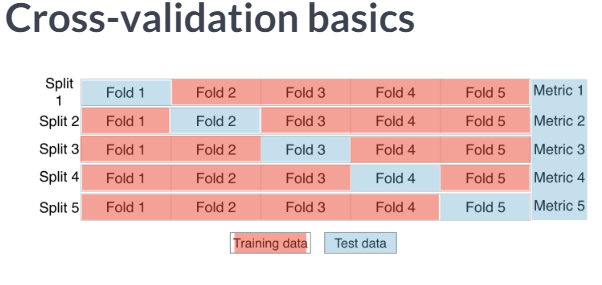
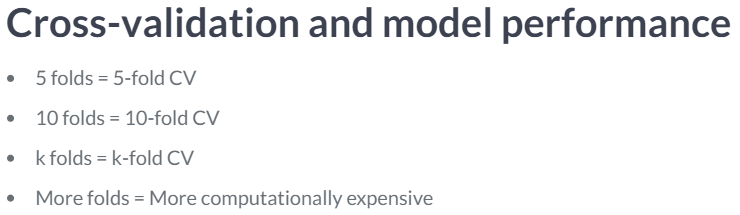
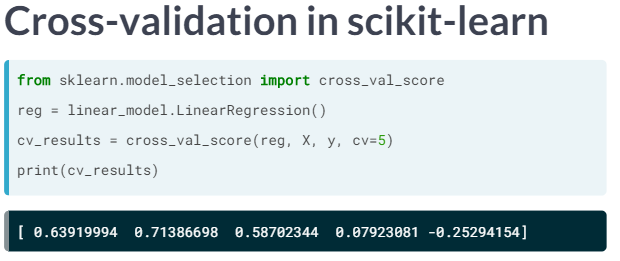
**Machine Learning Algorithms decision flow:**



**Cross Validation**

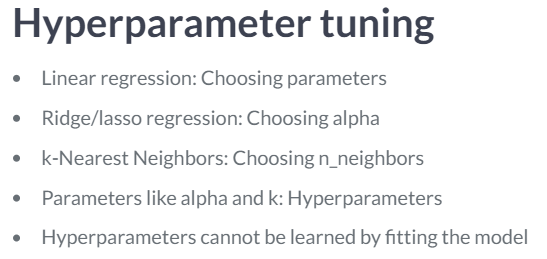






For linear regression results are R-squared values.

**Hyperparameter Tuning**



**Use cross validation with hyperparameter tuning**

**GridSearch will split training data** further into train and test to tune the hyper-parameters passed to it. And finally fit the model on the whole train data with best found parameters. Next test the model on the test data you kept aside in the beginning

#sample code

# parameters that cannot be learned from training

params = {

'colsample\_bytree': [0.4,0.6,0.8],

'learning\_rate':[0.1,0.2,0.3],

'n\_estimators': [100,200,300],

'max\_depth':[2,3,4],

'alpha':[1,10],

'subsamples':[0.5,0.6,0.7]

}

xgb\_cls = XGBClassifier()

# CV is the number of cross validation splits

gs = GridSearchCV(estimator=xgb\_cls,

param\_grid=params,

cv=3,

n\_jobs=-1,

verbose=2

)

gs.fit(X\_train,y\_train)

y\_pred = gs.predict(X\_test)

print(gs.best\_params\_)

print(gs.best\_score\_)

classification\_metrics(X\_test, y\_test, y\_pred, gs,'red','XGB', 'Male/Female Classification',True)

Cross Validation Concepts:

**Metrics for Regressions:**

RSquared:

TSS = ESS + RSS where ESS is the explained variance attributed to error and RSS (Residual Sum of Squares) is one way of quantifying how much error existing in the fitted model.

TSS = ESS + RSS

RSquared = 1 – (RSS/TSS)

ESS = TSS - RSS

RSS = ∑(actual value – predicted value) \*\* 2

TSS = ∑(actual value – actual average value) \*\* 2

Note: RSquared increases artificially as a side effect of increasing the number of independent variables. Use Adjusted RSquared for high number of features.

Adjusted RSquared:

Adjusted RSquared = RSquared – ( (1 – Rsquared) \* (K / (N – K – 1) )

K = number of variables, N is the sample size.

Mean Squared Error (MSE) and “Square” Root Mean Squared Error (RMSE)

MSE = (∑(actual value – predicted value) \*\* 2) / N or RSS/N

RMSE = √MSE or SQRT(MSE) or SQRT(RSS/N)

F-Statistics:

F-Statistic = (RSquared / (1 – Rsquared)) \* ( (N – K – 1) / K )

(F-Statistics is not the same as the F1-measure

Plot Residuals vs Actual y values.

Plot Actual y vs. Predicted y

Plot histogram of residuals

Plot feature Xi vs Xj to look for dependencies.

Sample function to calculate regression statistics:

def calc\_regression\_stats(X,y,yp):

y = np.array(y)

yp = np.array(yp)

n = len(y)

k = len(X.columns)

yavg = sum(y)/n

TSS = sum((y - yavg) \*\* 2)

RSS = sum((y - yp) \*\* 2)

Rsquared = 1 - (RSS/TSS)

AdjRsquared = Rsquared - ((1-Rsquared) \* ( k / ( n - k - 1 ) ) )

MSE = RSS / n

RMSE = np.sqrt(MSE)

Fstatistic = ( Rsquared / (1 - Rsquared) ) \* ( (n - k - 1 ) / k )

error = ( (y - yp) / y ) \* 100

AbsErrorSum = sum(abs(error))

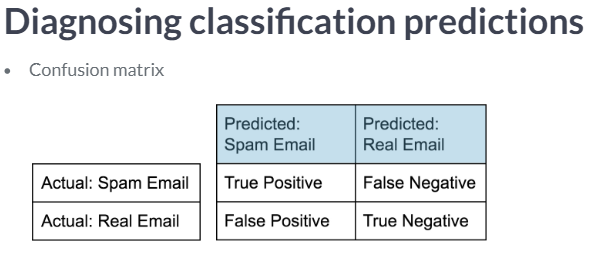
MeanOfError = np.mean(error)

StdOfError = np.std(error)

return Rsquared, AdjRsquared, MSE, RMSE, Fstatistic, MeanOfError, StdOfError, AbsErrorSum

**Metrics for Classification:**

Confusion Matrix



From Confusion Matrix Calculate or run Classification Report

Precision = true positive / (true positive + false positive)

Recall = true positive / (true positive + false negative)

(also calledsensitivity, hit rate, true positive rate)

Sensitivity = true positive / (true positive + false negative)

Specificity = true negative / (true negative + false positive)

Accuracy = (true positive + true negative) /

(true positive + true negative + false positive + false negative)

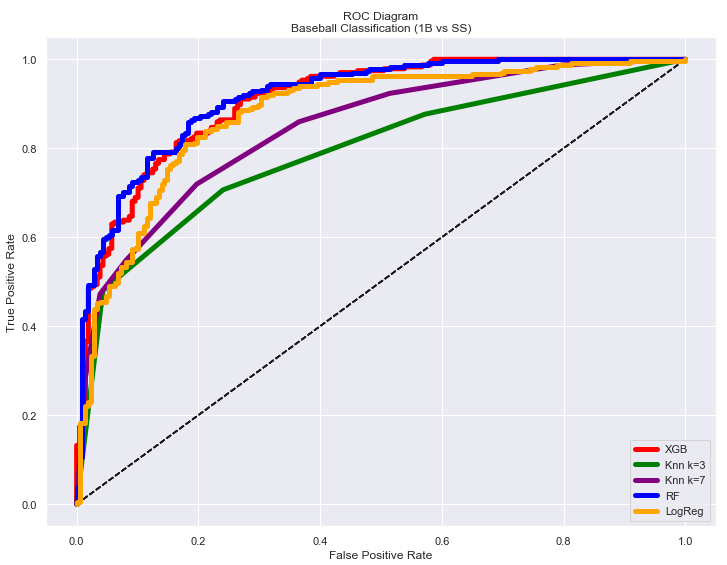
F1score = 2 \* (Precision \* Recall) / (Precision + Recall)

The **support** is the number of samples of the true response that lie in that class

High Precision (not many false positives)

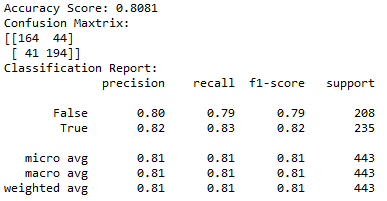
High Recall (predicted most true positives successfully)

Plot ROC (Receiver Operating Characteristic) Curve



y axis is sensitivity. x axis is (1-specificity) Chart is for various threshold settings (cutoff probability values for logistic regression)

Other metrics created by function.



Sample Code for Classifier Statistics:

def classification\_metrics(X\_test, y\_test, y\_pred, classifier,clr,lbl, roctitle,showflag):

print('Accuracy Score: %1.4f' % accuracy\_score(y\_pred,y\_test))

print('Confusion Maxtrix: ')

print(confusion\_matrix(y\_test,y\_pred))

print('Classification Report: ')

print(classification\_report(y\_test,y\_pred))

y\_pred\_prob = classifier.predict\_proba(X\_test)[:,1]

fpr, tpr, thesholds = roc\_curve(y\_test, y\_pred\_prob)

plt.plot([0,1],[0,1], 'k--')

plt.plot(fpr, tpr, linewidth=5,color=clr,label=lbl)

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title(roctitle)

#plot many plots of ROC curves until showflag = True

if showflag == True:

leg = plt.legend(loc='lower right')

plt.show()

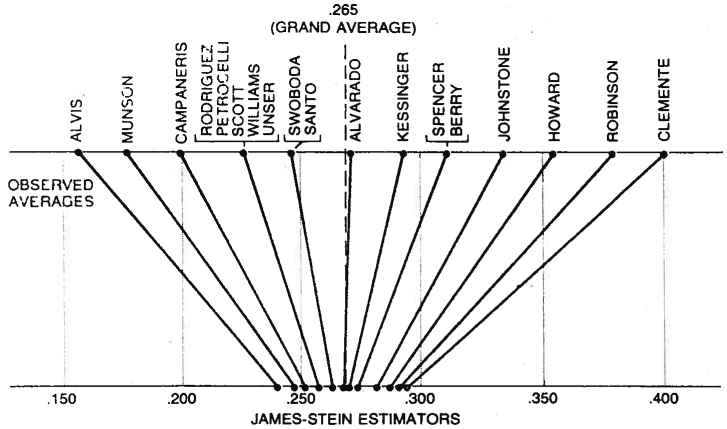
return True

**Algorithm: Linear Regression**

Technical Information:

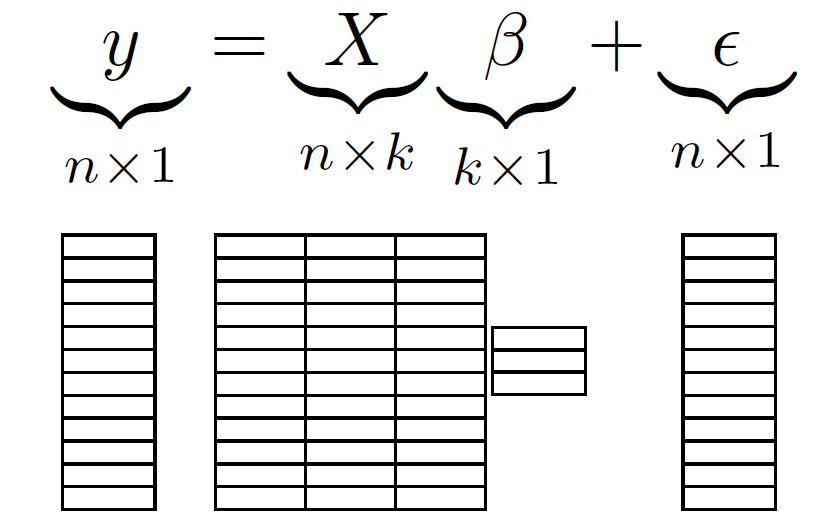
RTTM – Regression Toward the Mean. If you do excellent in first year sports, it could be a combination of skill and luck.

Stein’s Paradox: Shrinking all individual averages toward the grand average (average of averages)



MTTM is a statistical phenomenon that occurs when unusually large or unusually small measurement values are followed by values that are closer to the population mean. This is due to random measurement error or, put another way, non-systematic fluctuations around the true mean.

OLS – ordinary least squares is the criterion for calculating the linear relationship.



k is the number of predictor variables (independent variables for features)

n is the sample size

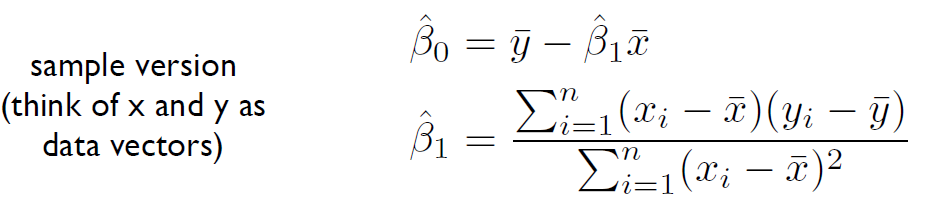
beta’s are the coefficients that you are trying to predict for determining y.

epsilon is the error

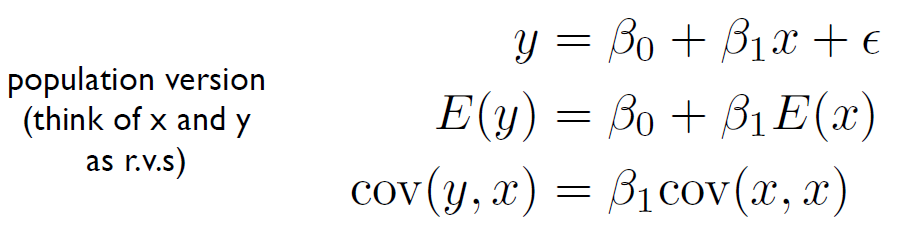
y is the dependent variable or target variable you are trying to predict.

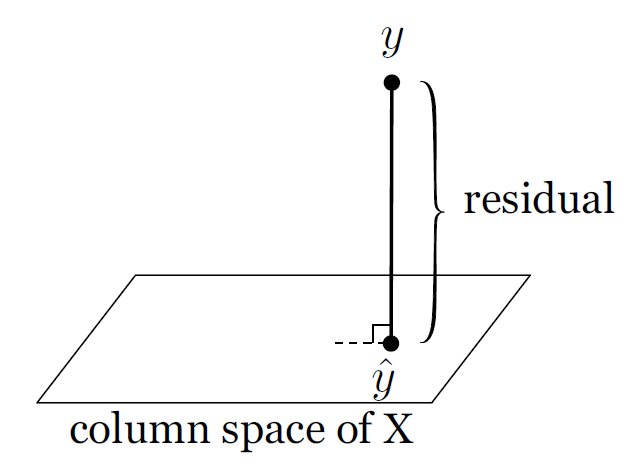
NOTE: X’s don’t have to be at power of 1. X’s can be of higher degree. The beta’s must be degree one.

Calculating OLS:



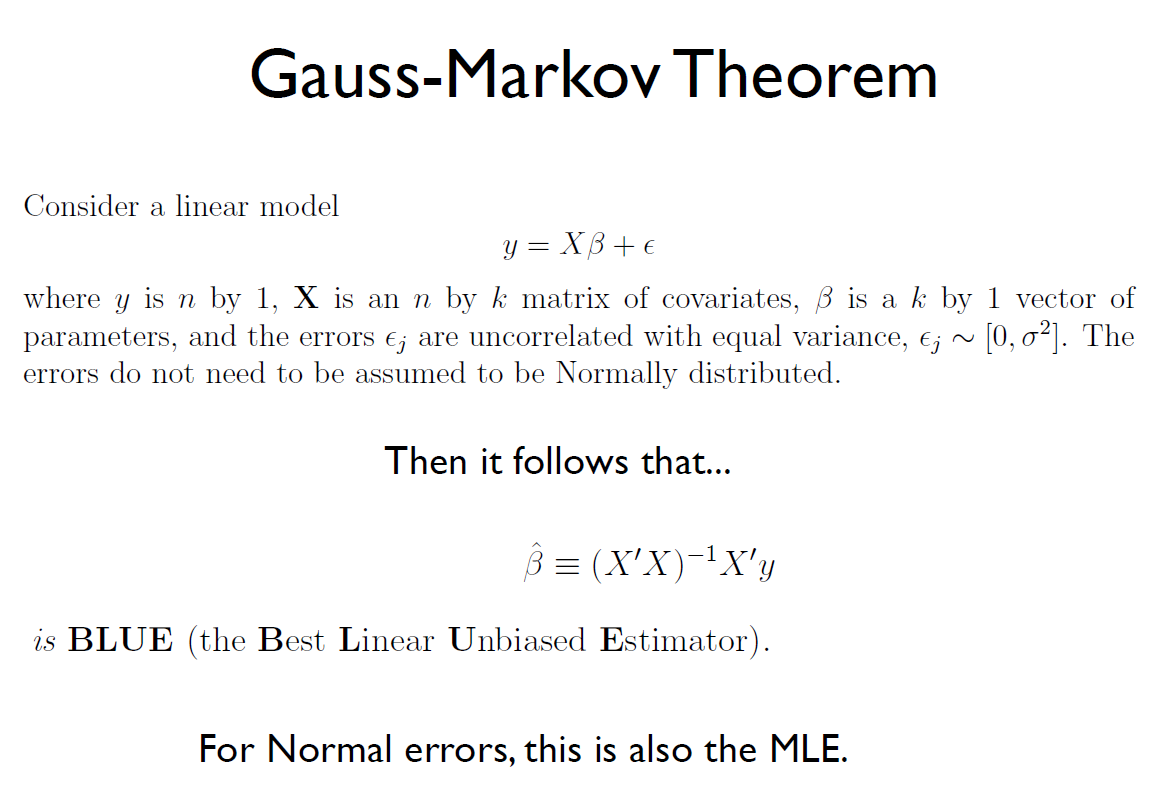
Another way to think about it is:





Regression is a projection. Find the closest point in the column space of X. What combination of X can you get that is the closest to y.

If you are working with normal distributions then the best predictor of y is linear. Theorem as follow states this.



Seven OLS Assumptions for Linear Regression:

1. The regression model is linear in the coefficients and the error term. a regression model is linear when all terms in the model are either the constant or a parameter multiplied by an independent variable.
2. The error term (epsilon) has a population mean of zero. The error term accounts for the variation in the dependent variable that the independent variables do not explain.
3. All independent variables are uncorrelated with the error term. If an independent variable is correlated with the error term, we can use the independent variable to predict the error term, which violates the notion that the error term represents unpredictable random error.
4. Observations of the error term are uncorrelated with each other. One observation of the error term should not predict the next observation. Plot Residuals vs. Observed Values…you should see no pattern.
5. The error term has a constant variance (no heteroscedasticity). The variance of the errors should be consistent for all observations. Again, look at the Residuals vs. Observed Values.
6. No independent variable is a perfect linear function of other explanatory variables. Perfect correlation suggests that two variables are different forms of the same variable using Pearson’s correlation coefficient. EG… SLG + OBP = OPS. If SLG and OBP are X’s then you get a perfect linear relationship…SLG and OBP are just another way of saying OPS.
7. The error term is normally distributed. OLS does not require that the error term follows a [normal distribution](https://statisticsbyjim.com/basics/normal-distribution/) to produce unbiased estimates with the minimum variance. However, satisfying this assumption allows you to perform statistical hypothesis testing and generate reliable confidence intervals and prediction intervals. Use a QQPlot to plot residuals or test them using a normal test.

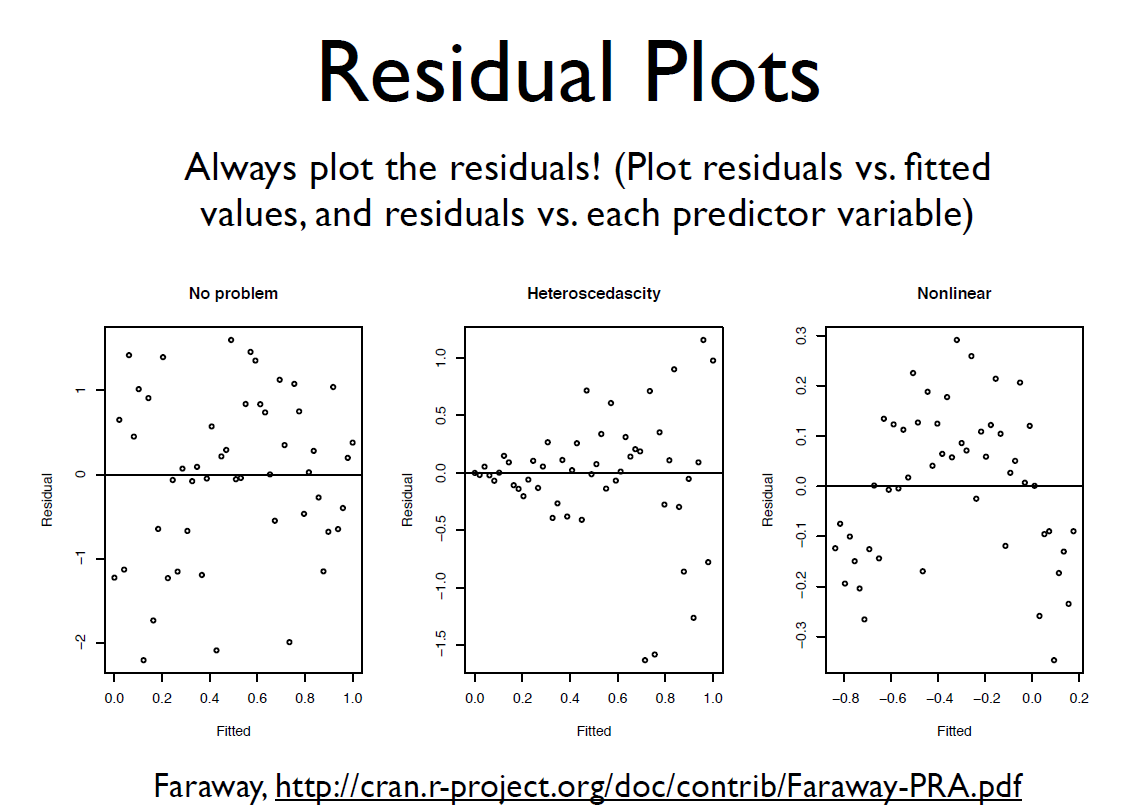
In a nutshell, your linear model should produce residuals that have a mean of zero, have a constant variance, and are not correlated with themselves or other variables. If these assumptions hold true, the OLS procedure creates the best possible estimates.

The best estimates are those that are unbiased and have the minimum variance. When your model satisfies the assumptions, the Gauss-Markov theorem states that the OLS procedure produces unbiased estimates that have the minimum variance.

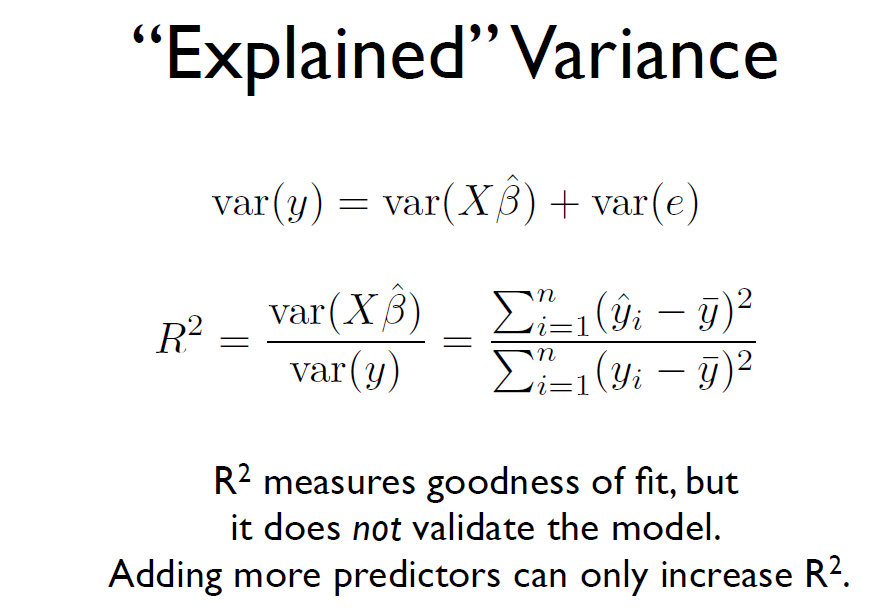
Gauss-Markov Theorem states that OLS is BLUE (Best Linear Unbiased Estimator) – minimum variance or narrowest sampling distribution. That is,  when your model satisfies the assumptions, OLS coefficient estimates follow the tightest possible sampling distribution of unbiased estimates compared to other linear estimation methods.

Should avoid having predictor variables that are highly correlated with each other (collinearity

results in instability, high variances in estimates, and worse interpretability)



How good is the model for the business problem:



Sample Code:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y, test\_size = 0.4, random\_state=61)

reg = LinearRegression()

reg.fit(X\_train, y\_train)

y\_pred = reg.predict(X\_test)

# function that calculates RSquared, MSE, RMSE, and presents plots of y and predicted y, residuals and y # and residual histograms.

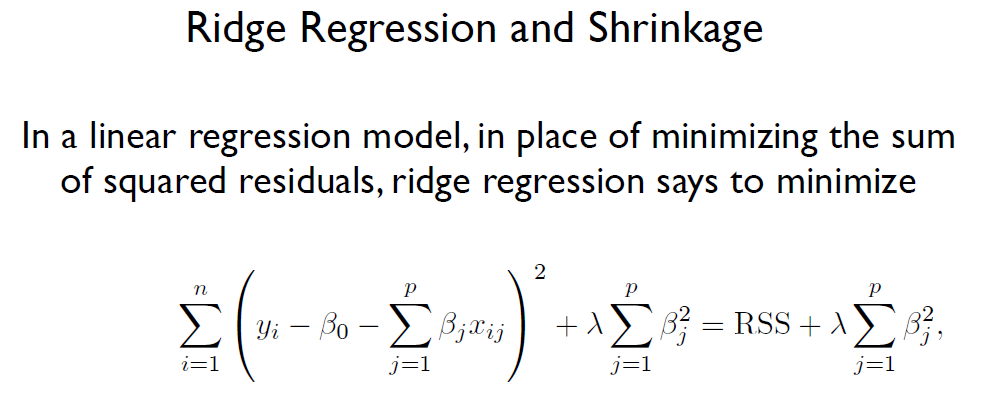
lr\_results(df,X\_test,y\_test,y\_pred,path,'OPSpredictions.csv',stats\_list,reg)

**NOTE: some predictors or features need to be rescaled. High values and relatively low feature values can affect the model in erroneous ways. You can normalize data by zero-mean normalization (Xi – mean (Xi)) / standard deviation (Xi). Or by min/max normalization. (Xi – min(Xi)) / (max(Xi) – min(Xi))**

Variations of Linear Regression: Takes into account Shrinkage and Curse of Dimensionality.

**Regularization: Penalizes for large coefficients. Larger coefficients tends to lead to overfitting**. **Regularization also deals with multicollinearity.**

**Ridge Regression (extension of linear regression):**



Lamda is a tuning **hyperparameter**…use cross validation to choose it. This is a class of penalty imposing regression models. Don’t want sum of squared betas to be too large which leads to overfitting. Takes care of multicollinearity as well.

Ridge regression uses a type of [shrinkage estimator](https://www.statisticshowto.datasciencecentral.com/shrinkage-estimator/) called a ridge estimator. Shrinkage estimators theoretically produce new [estimators](https://www.statisticshowto.datasciencecentral.com/estimator/)that are shrunk closer to the “true” population parameters. The ridge estimator is especially good at improving the least-squares estimate when multicollinearity is present. Deals with situations where variables (k) are more than observations (n).

**Multicollinearity can be briefly described as the phenomenon in which two or more identified predictor variables are linearly related, or codependent. Regularization techniques can be used to combat multicollinearity : L1 (Lasso), L2 (Ridge Regression), and Elastic Nets. A Pearson Coefficient of 0.8 or higher may indicate severe multicollinearity. Possible for individual regression coefficients to be insignificant but for the overall fit of the equation to be high (overfitting)**

L2 Regularization: Ridge regression penalizes sum of squared coefficients. Makes coefficients small where there is collinearity.

# Ridge Regression sample code

ridge = Ridge(alpha=0.001, normalize=True)

ridge.fit(X\_train, y\_train)

y\_pred = ridge.predict(X\_test)

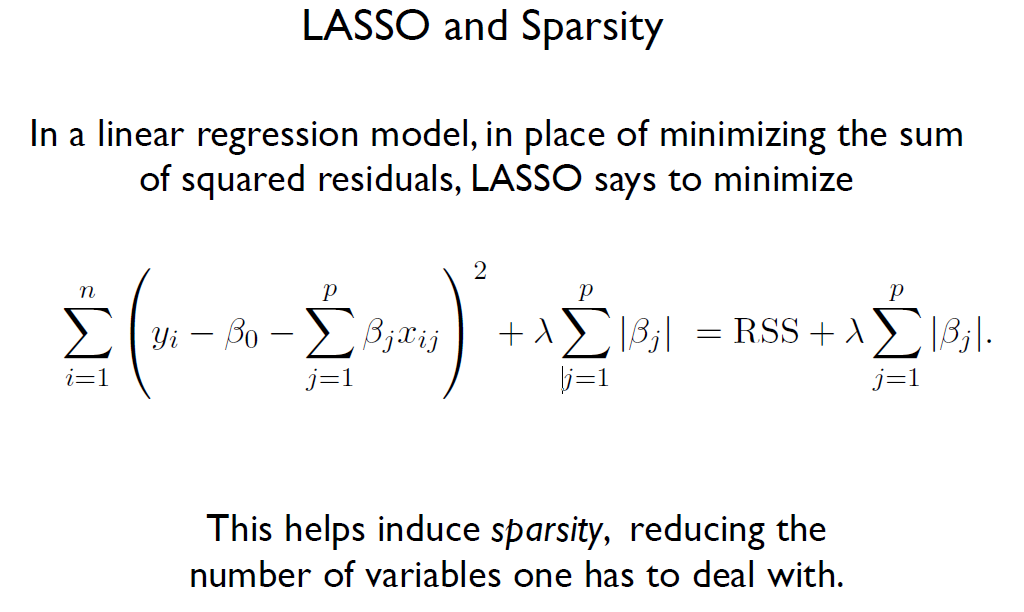
print('\n')

print('Ridge ...')

print('\n')

lr\_results(df,X\_test,y\_test,y\_pred,path,'OPSpredictionsRidge.csv',stats\_list,ridge) # custom function

**Lasso Regression (extension of linear regression) – shrinking the features to features of most impact.**



Takes variables which don’t do much to zero.

L1 Regularization: penalizes the sum of their absolute values. Takes variables to zero what don’t do much for the model. (ones that have collinearity)

Lamda is a tuning **hyperparameter**…use cross validation to choose it. . Don’t want sum of squared betas to be too large which leads to overfitting. Takes care of multicollinearity as well.

lasso = Lasso(alpha=0.0001)

lasso\_coef = lasso.fit(X\_train, y\_train).coef\_

y\_pred = lasso.predict(X\_test)

lr\_results(df,X\_test,y\_test,y\_pred,path,'OPSpredictionsLasso.csv',stats\_list,lasso)

cols = feature\_list

plt.plot(range(len(cols)), lasso\_coef)

plt.xticks(range(len(cols)), cols, rotation=45)

plt.ylabel('Coefficients')

plt.show()

**Elastic Net Regression – combination of Ridge and Lasso regression.**

Elastic Net is a mix of both L1 and L2 Regularizations. A penalty is applied to the sum of the absolute values and to the sum of the squared values.

#Elasticnet with GridSearchCV

# Create the hyperparameter grid

l1\_space = np.linspace(0, 1, 30)

param\_grid = {'l1\_ratio': l1\_space}

# Instantiate the ElasticNet regressor: elastic\_net

elastic\_net = ElasticNet()

# Setup the GridSearchCV object: gm\_cv

gm\_cv = GridSearchCV(elastic\_net, param\_grid, cv=5)

# Fit it to the training data

gm\_cv.fit(X\_train, y\_train)

# Predict on the test set and compute metrics

y\_pred = gm\_cv.predict(X\_test)

lr\_results(df,X\_test,y\_test,y\_pred,path,'OPSpredictionsGSElastic.csv',stats\_list,gm\_cv)

print("Tuned ElasticNet l1 ratio: {}".format(gm\_cv.best\_params\_))

**Decision Trees and Random Forests**

**Pros**

•Fast training

•Can use categories

•Don’t have to scale values

•Fast prediction

•Easy to understand

•Easy to interpret

•Can do multiple classification (not just binary)

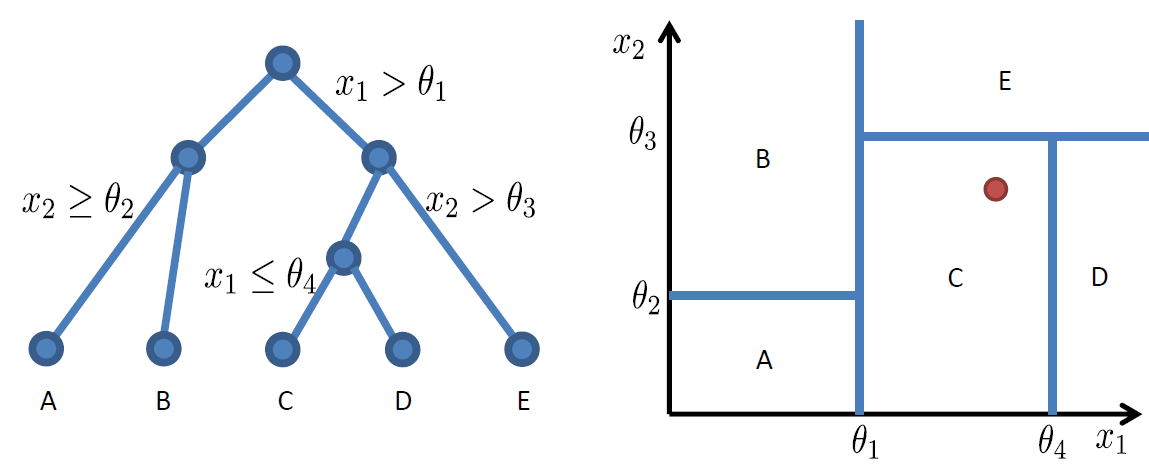
**Cons**

•Sensitive to small changes in the data

•Overfitting (not popular for this reason)

•Only looks at a node at a time

•Only axis aligned splits



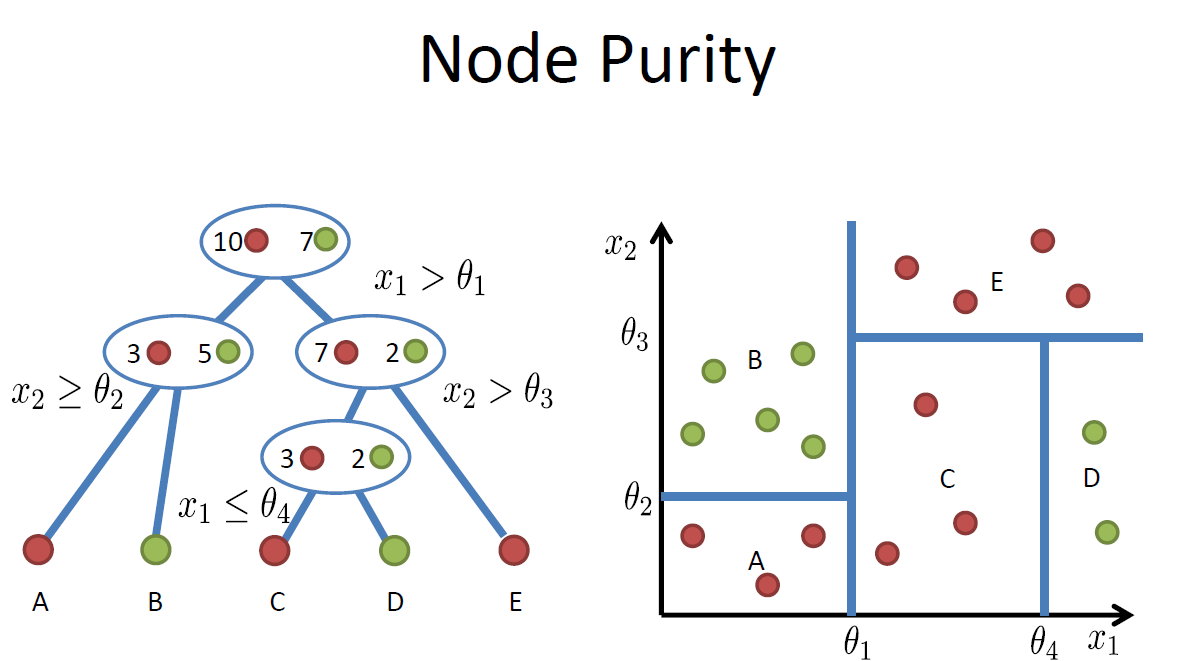
**Only does axis aligned splits (not diagonal)**

**Decisions?**

**Which features to query.**

**What thresholds to choose (how deep do you go).**

**Lowest level tells you the category it belongs to (classification)**

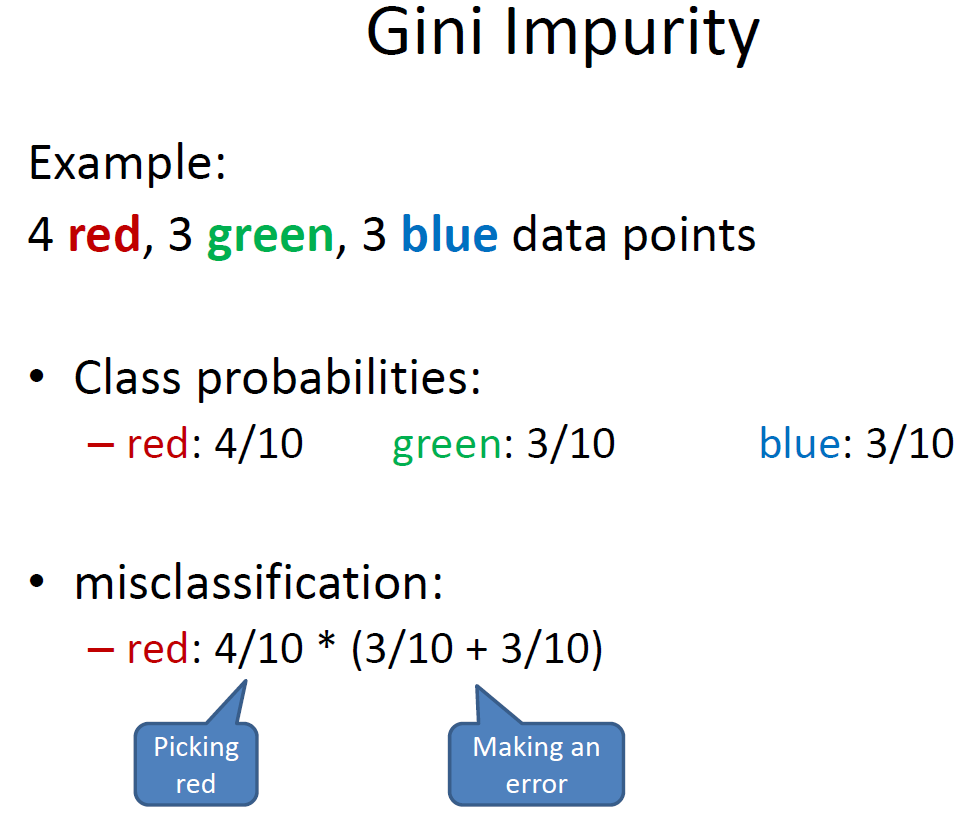


**End nodes only have red or green exclusively (only one class in them). Extreme case would have leaf node for every point (overfitting).**

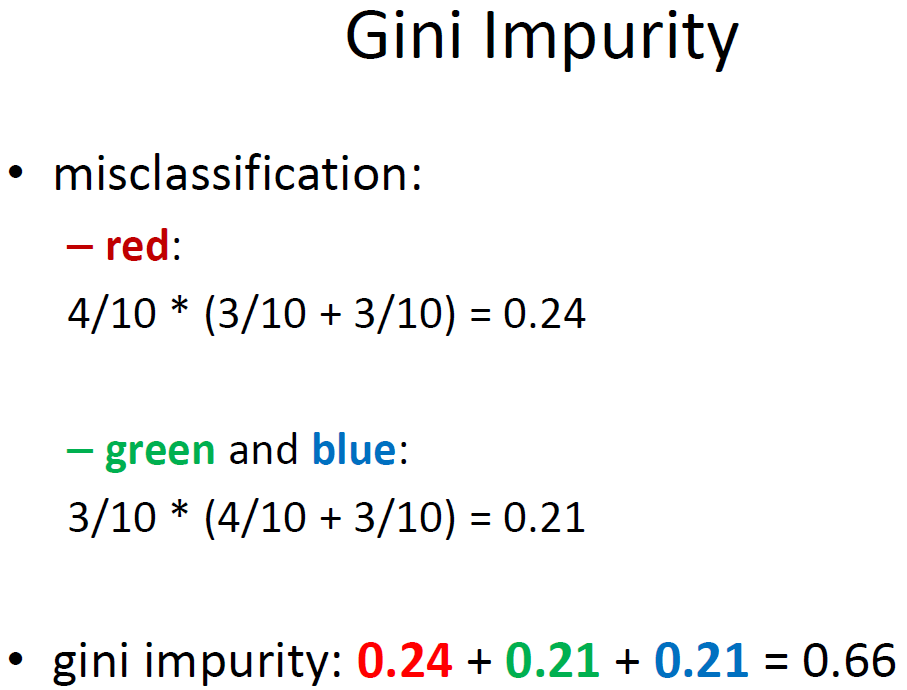
**Gini Impurity is used for node purity. Used in sklearn.**

**Gini Impurity:**

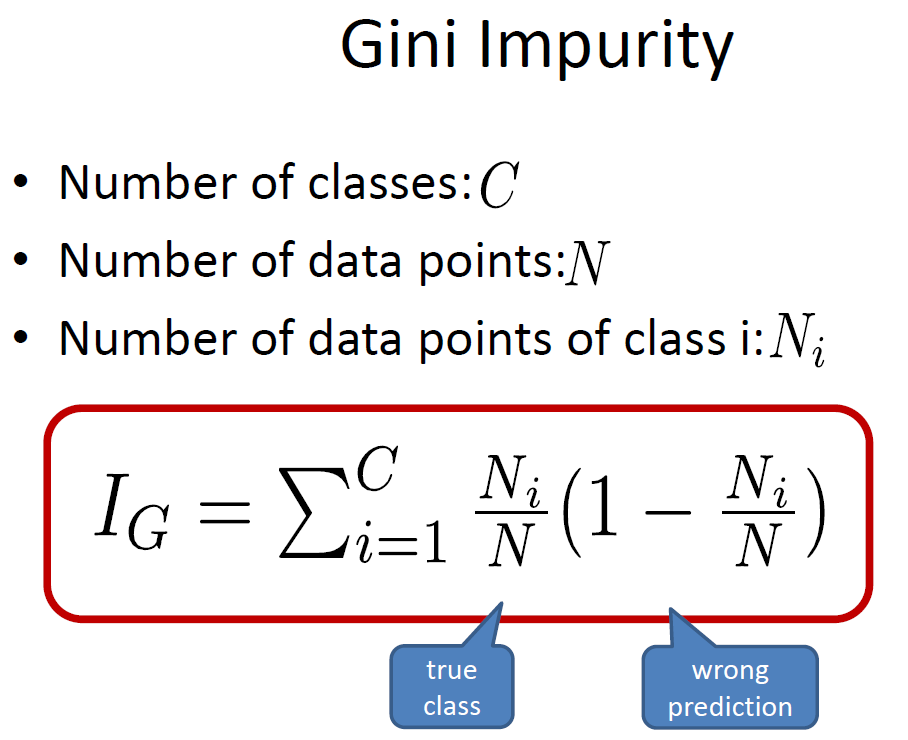
* **Expected Error**
* **If you randomly choose a sample**
* **And predict the class of the entire node based upon it.**

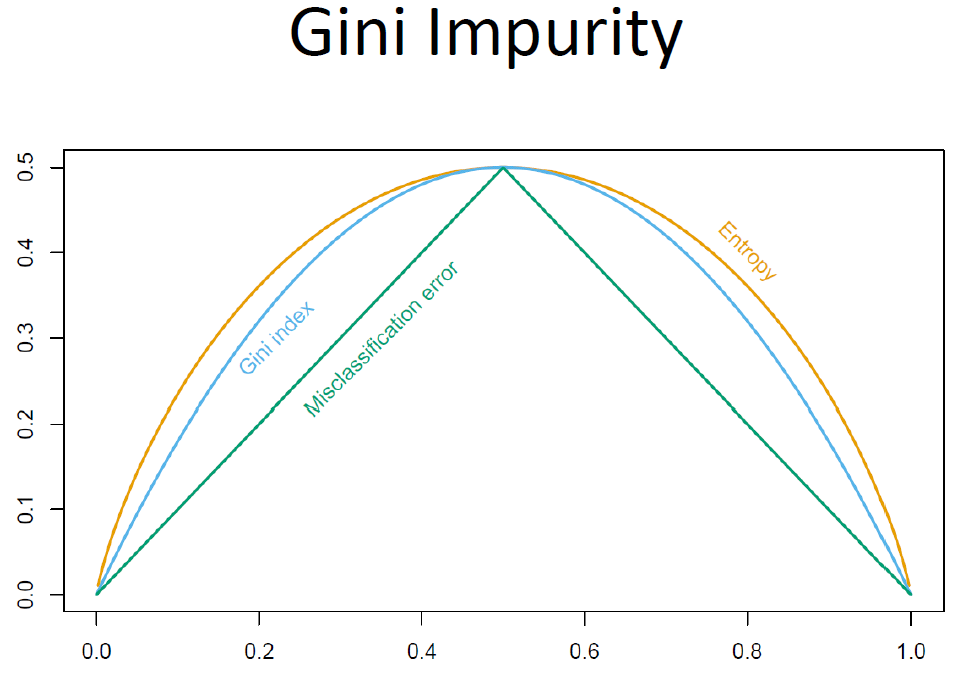


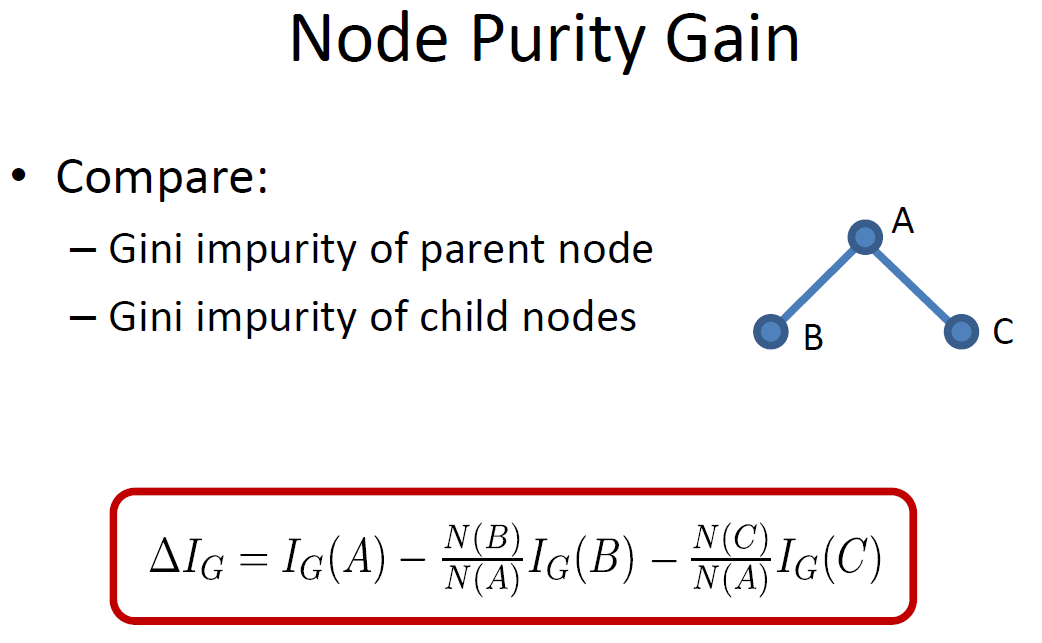
**Probability picking a red ball and chance of making an error.**

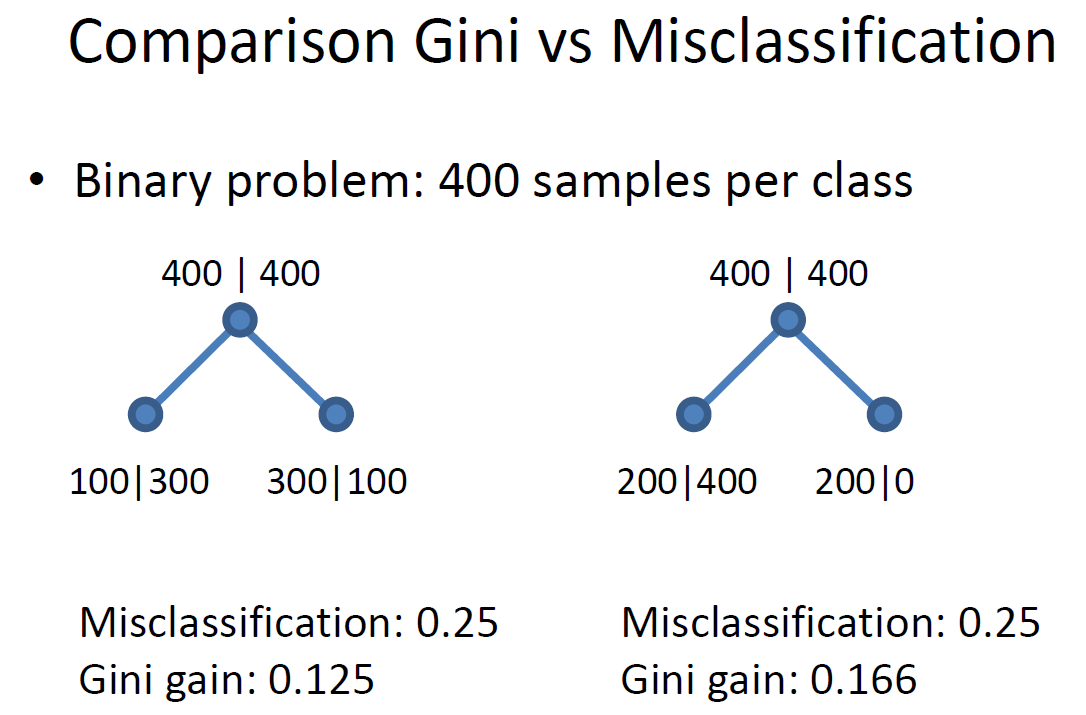


**Gives gini impurity for a specific node.**









Pseudocode:

Check if already finished

For each feature Xi

Calculate the gain from splitting on Xi

Let Xbest be the feature with the highest gain

Create a decision node that splits on Xbest

Repeat on the sub-nodes

Stop When:

Node contains only one class

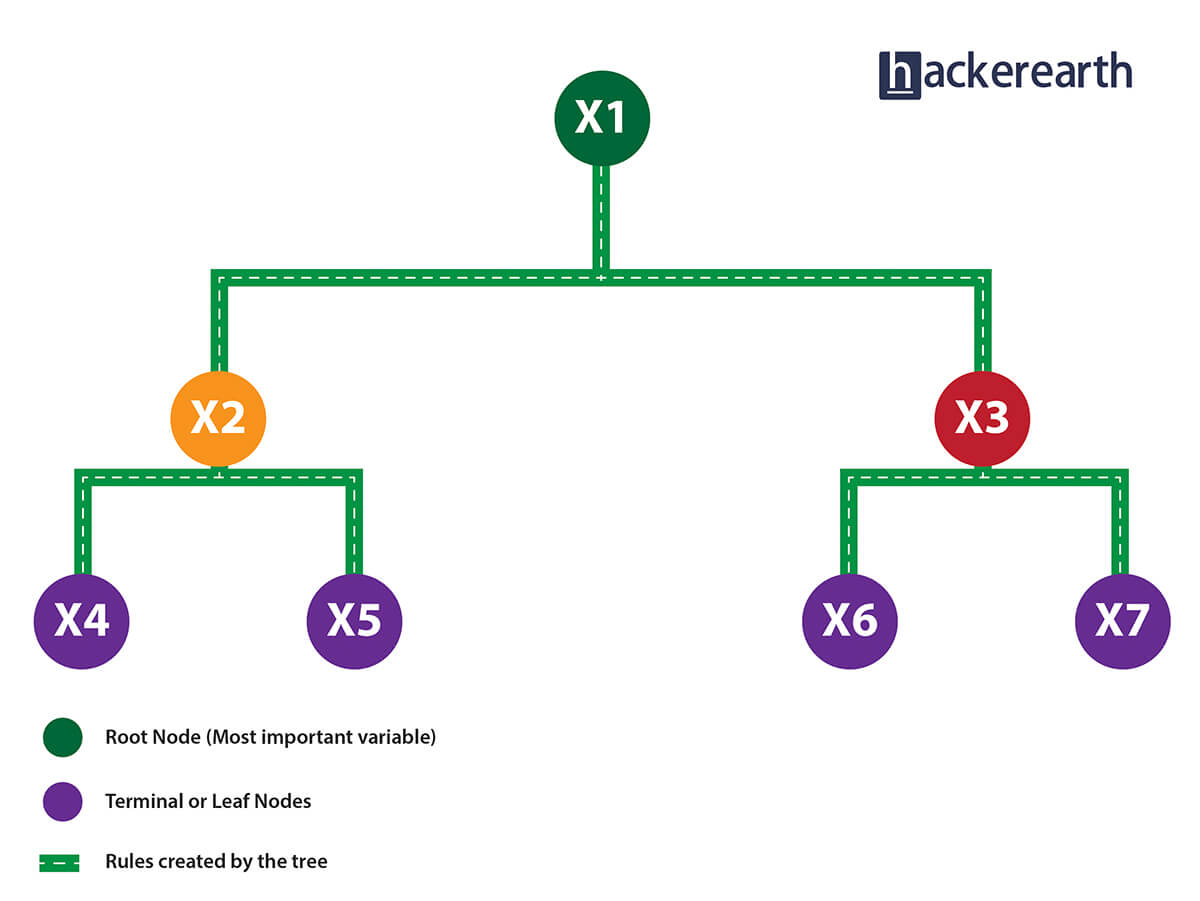
Node contains less than x data points

Max depth is reached

Node purity is sufficient

Overfitting occurs (cross-validation)

**How does it work? (Decision Tree, Random Forest)**

To understand the working of a random forest, it's crucial that you understand a **tree**. A tree works in the following way:[](http://blog.hackerearth.com/wp-content/uploads/2016/12/root-01.jpg)

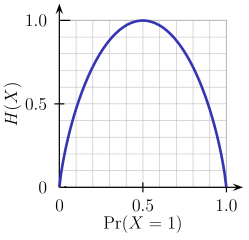
1. Given a data frame (n x p), a tree stratifies or partitions the data based on rules (if-else). Yes, a tree creates rules. These rules divide the data set into distinct and non-overlapping regions. These rules are determined by a variable's contribution to the homogenity or pureness of the resultant child nodes (X2,X3).

2. In the image above, the variable X1 resulted in highest homogeneity in child nodes, hence it became the root node. A variable at root node is also seen as the most important variable in the data set.

3, But how is this homogeneity or pureness determined? In other words, how does the tree decide at which variable to split?

* In **regression trees** (where the output is predicted using the mean of observations in the terminal nodes), the splitting decision is based on minimizing RSS. The variable which leads to the greatest possible reduction in RSS is chosen as the root node. The tree splitting takes a **top-down greedy** approach, also known as *recursive binary splitting*. We call it "greedy" because the algorithm cares to make the best split at the current step rather than saving a split for better results on future nodes.
* In **classification trees** (where the output is predicted using mode of observations in the terminal nodes), the splitting decision is based on the following methods:
  + **Gini Index** - It's a measure of node purity. If the Gini index takes on a smaller value, it suggests that the node is pure. For a split to take place, the Gini index for a child node should be less than that for the parent node.
  + **Entropy** - Entropy is a measure of node impurity. For a binary class (a,b), the formula to calculate it is shown below. Entropy is maximum at p = 0.5. For p(X=a)=0.5 or p(X=b)=0.5 means, a new observation has a 50%-50% chance of getting classified in either classes. The entropy is minimum when the probability is 0 or 1.

Entropy = - p(a)\*log(p(a)) - p(b)\*log(p(b))



In a nutshell, every tree attempts to create rules in such a way that the resultant terminal nodes could be as pure as possible. Higher the purity, lesser the uncertainty to make the decision.

But a decision tree suffers from high variance. "High Variance" means getting high prediction error on unseen data. We can overcome the variance problem by using more data for training. But since the data set available is limited to us, we can use resampling techniques like bagging and random forest to generate more data.

Building many **decision** **trees** results in a **forest**. A random forest works the following way:

1. First, it uses the Bagging (Bootstrap Aggregating) algorithm to create random samples. Given a data set D1 (n rows and p columns), it creates a new dataset (D2) by sampling n cases at random with replacement from the original data. About 1/3 of the rows from D1 are left out, known as Out of Bag(OOB) samples.
2. Then, the model trains on D2. OOB sample is used to determine unbiased estimate of the error.
3. Out of p columns, P << p columns are selected at each node in the data set. The P columns are selected at random. Usually, the default choice of P is p/3 for regression tree and P is sqrt(p) for classification tree.
4. Unlike a tree, no pruning takes place in random forest; i.e, each tree is grown fully. In decision trees, pruning is a method to avoid overfitting. Pruning means selecting a subtree that leads to the lowest test errror rate. We can use cross validation to determine the test error rate of a subtree.
5. Several trees are grown and the final prediction is obtained by averaging or voting.

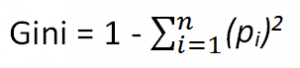
Each tree is grown on a different sample of original data. Since random forest has the feature to calculate OOB error internally, cross validation doesn't make much sense in random forest.

**Calculating Gini Index Example:**

### **What is Gini Index?**

Gini index or Gini impurity measures the degree or probability of a particular variable being wrongly classified when it is randomly chosen. But what is actually meant by ‘impurity’? If all the elements belong to a single class, then it can be called pure. The degree of Gini index varies between 0 and 1, where 0 denotes that all elements belong to a certain class or if there exists only one class, and 1 denotes that the elements are randomly distributed across various classes. A Gini Index of 0.5 denotes equally distributed elements into some classes.

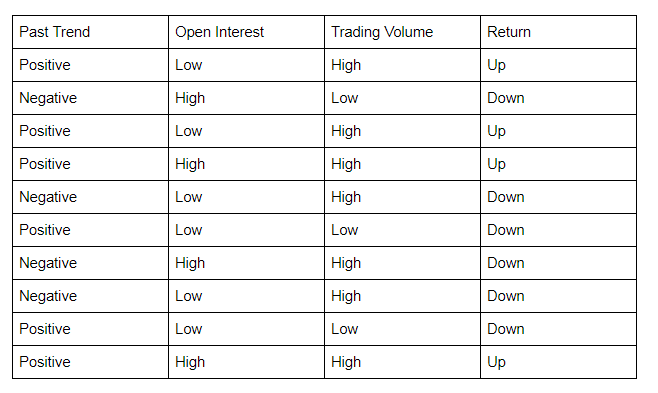
**Formula for Gini Index**



where piis the probability of an object being classified to a particular class.

While building the [decision tree](https://quantra.quantinsti.com/course/decision-trees-analysis-trading-ernest-chan), we would prefer choosing the attribute/feature with the least Gini index as the root node.

### **Let’s understand with a simple example of how the Gini Index works.**



#### **Let’s start by calculating the Gini Index for ‘Past Trend’.**

P(Past Trend=Positive): 6/10

P(Past Trend=Negative): 4/10

If (Past Trend = Positive & Return = Up), probability = 4/6

If (Past Trend = Positive & Return = Down), probability = 2/6

Gini index = 1 - ((4/6)^2 + (2/6)^2) = 0.45

If (Past Trend = Negative & Return = Up), probability = 0

If (Past Trend = Negative & Return = Down), probability = 4/4

Gini index = 1 - ((0)^2 + (4/4)^2) = 0

Weighted sum of the Gini Indices can be calculated as follows:

Gini Index for Past Trend = (6/10)0.45 + (4/10)0 = 0.27

#### **Calculation of Gini Index for Open Interest**

P(Open Interest=High): 4/10

P(Open Interest=Low): 6/10

If (Open Interest = High & Return = Up), probability = 2/4

If (Open Interest = High & Return = Down), probability = 2/4

Gini index = 1 - ((2/4)^2 + (2/4)^2) = 0.5

If (Open Interest = Low & Return = Up), probability = 2/6

If (Open Interest = Low & Return = Down), probability = 4/6

Gini index = 1 - ((2/6)^2 + (4/6)^2) = 0.45

Weighted sum of the Gini Indices can be calculated as follows:

Gini Index for Open Interest = (4/10)0.5 + (6/10)0.45 = 0.47

#### **Calculation of Gini Index for Trading Volume**

P(Trading Volume=High): 7/10

P(Trading Volume=Low): 3/10

If (Trading Volume = High & Return = Up), probability = 4/7

If (Trading Volume = High & Return = Down), probability = 3/7

Gini index = 1 - ((4/7)^2 + (3/7)^2) = 0.49

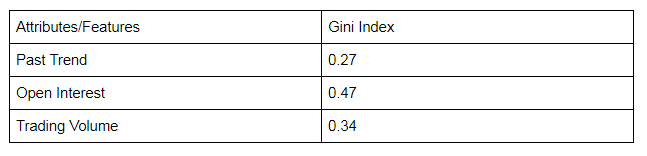
If (Trading Volume = Low & Return = Up), probability = 0

If (Trading Volume = Low & Return = Down), probability = 3/3

Gini index = 1 - ((0)^2 + (1)^2) = 0

Weighted sum of the Gini Indices can be calculated as follows:

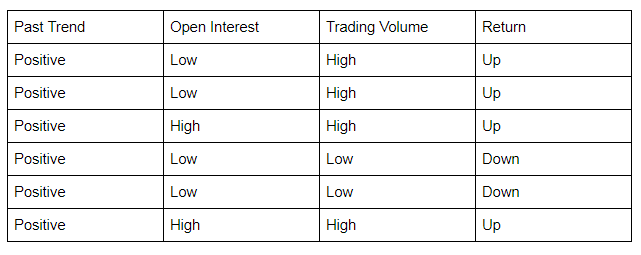
Gini Index for Trading Volume = (7/10)0.49 + (3/10)0 = 0.34



From the above table, we observe that ‘Past Trend’ has the lowest Gini Index and hence it will be chosen as the root node for [how decision tree works](https://quantra.quantinsti.com/course/decision-trees-analysis-trading-ernest-chan).

We will repeat the same procedure to determine the sub-nodes or branches of the decision tree.

We will calculate the Gini Index for the ‘Positive’ branch of Past Trend as follows:



#### **Calculation of Gini Index of Open Interest for Positive Past Trend**

P(Open Interest=High): 2/6

P(Open Interest=Low): 4/6

If (Open Interest = High & Return = Up), probability = 2/2

If (Open Interest = High & Return = Down), probability = 0

Gini index = 1 - (sq(2/2) + sq(0)) = 0

If (Open Interest = Low & Return = Up), probability = 2/4

If (Open Interest = Low & Return = Down), probability = 2/4

Gini index = 1 - (sq(0) + sq(2/4)) = 0.50

Weighted sum of the Gini Indices can be calculated as follows:

Gini Index for Open Interest = (2/6)0 + (4/6)0.50 = 0.33

#### **Calculation of Gini Index for Trading Volume**

P(Trading Volume=High): 4/6

P(Trading Volume=Low): 2/6

If (Trading Volume = High & Return = Up), probability = 4/4

If (Trading Volume = High & Return = Down), probability = 0

Gini index = 1 - (sq(4/4) + sq(0)) = 0

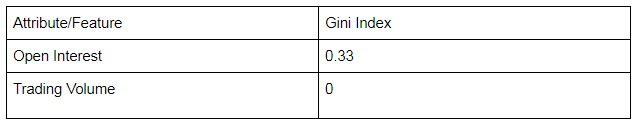
If (Trading Volume = Low & Return = Up), probability = 0

If (Trading Volume = Low & Return = Down), probability = 2/2

Gini index = 1 - (sq(0) + sq(2/2)) = 0

Weighted sum of the Gini Indices can be calculated as follows:

Gini Index for Trading Volume = (4/6)0 + (2/6)0 = 0

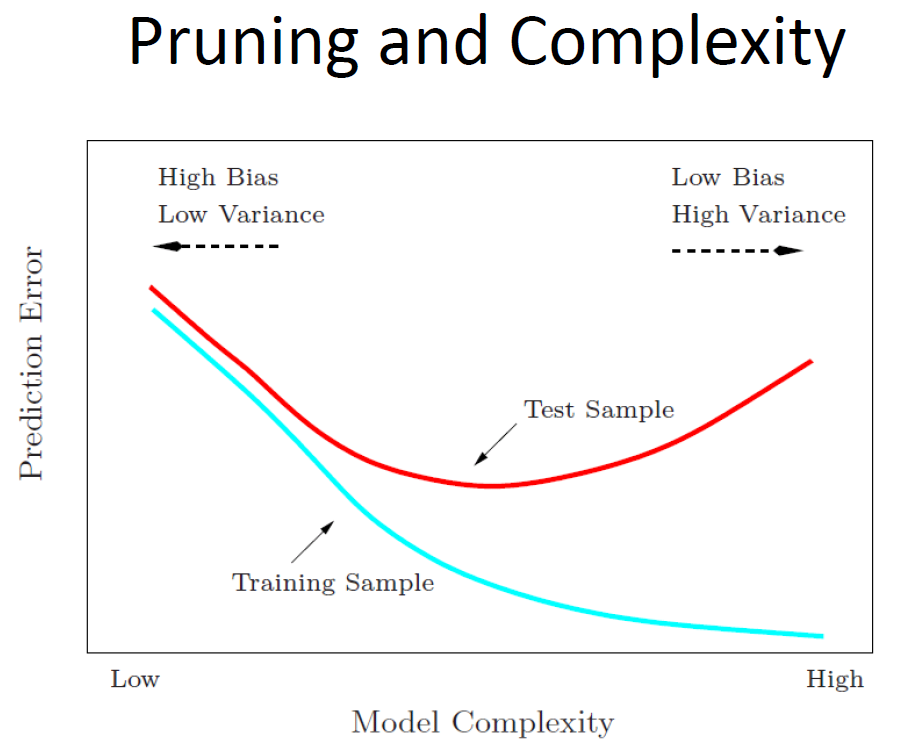


We will split the node further using the ‘Trading Volume’ feature, as it has the minimum Gini index.  
  
Learn [how to make a decision tree](https://quantra.quantinsti.com/course/decision-trees-analysis-trading-ernest-chan) to predict the markets and find trading opportunities using AI techniques with our Quantra course.

### **Conclusion**

Gini Index, unlike information gain, isn’t computationally intensive as it doesn’t involve the logarithm function used to calculate entropy in information gain, which is why Gini Index is preferred over Information gain.

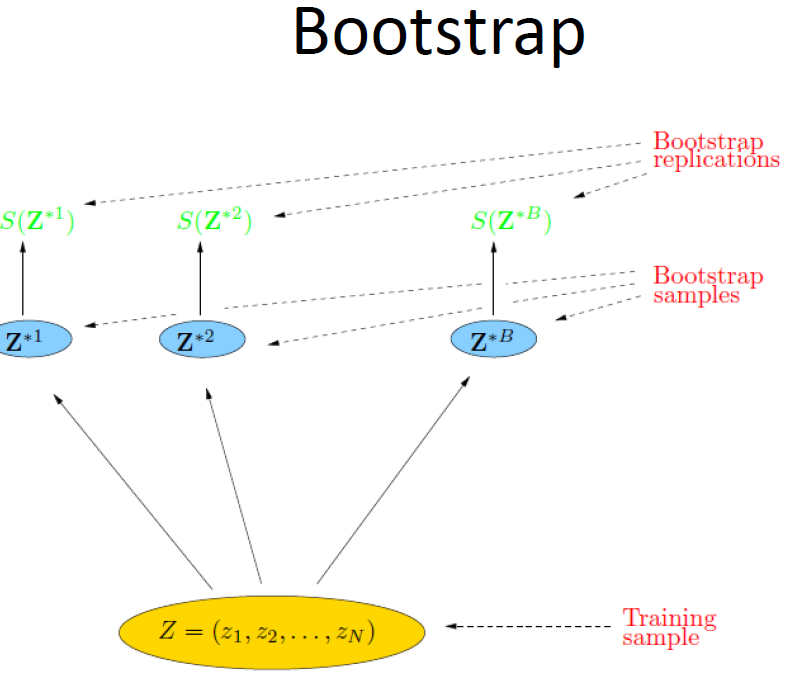
You can learn more about different splitting measures including Gini Index, information gain, etc. in the course on Decision Trees.



**Random Forest (Ensemble Methods) – learning multiple trees with bagging.**

**Bootstrapping**

* **Resampling method from statistics**
* **Useful to get error bars on estimates**
* **Take N data points**
* **Draw N times with replacement**
* **Get estimate for each bootstrap sampled**



**Probability of choosing observation n = 1/N**

**Probability of not choosing n = 1 – (1/N)**

**Probability of not choosing n in N draws = (1 – (1/N)) \*\* N**

**Probability of choosing n in N draws = 1 – (1 – (1/N)) \*\* N**

**p(n in bootstrap) = 1 – e \*\* -1 or approximately 0.632 or the bootstrap sample omits about 36.8% of data**

**BAGGING (Bootstrap AGGregatING)**

•Sample with replacement from your data set

•Learn a classifier for each bootstrap sample

•Average the results

**Bagging Results**

•Reduces overfitting (variance) without introducing too much bias

•Normally uses one type of classifier

•Decision trees are popular

•Not helping with linear models (if you average a linear model, you end up with a line)

•Easy to parallelize

**Random Forests extends Bagging of Decision Trees**

•Builds upon the idea of bagging (randomizing bootstrap)

•Each tree built from bootstrap sample

•Node splits calculated from random feature subsets

**Extension of bagging of decision trees:**

Generate bootstrap training samples, and a tree based on each, but also randomize set of predictors allowed each time a split is considered.

• Typically, # of random predictors chosen to be of order the square root of the total number of predictors.

• All trees are fully grown

• No pruning

• Two parameters (hyperparameters)

– Number of trees

– Number of features to consider when splitting node. Square root of k where k is number of features is recommended.

# sample code random forest regressor

rf = RandomForestRegressor(n\_estimators=1000,random\_state=61)

rf.fit(X\_train, y\_train)

pred = rf.predict(X\_test)

errors = abs(pred - y\_test)

df\_results = df.loc[X\_test.index, :]

df\_results['predOPS'] = pred

#function that prints all regression metrics

lr\_results(df,X\_test,y\_test,pred,path,'OPSpredictionsRF.csv',stats\_list,rf) # function for metrics

# sample code random classifier

rf\_cls = RandomForestClassifier(bootstrap=True, class\_weight=None, criterion='gini',

max\_depth=7, max\_features='auto', max\_leaf\_nodes=None,

min\_impurity\_decrease=0.0, min\_impurity\_split=None,

min\_samples\_leaf=1, min\_samples\_split=2,

min\_weight\_fraction\_leaf=0.0, n\_estimators=1000, n\_jobs=None,

oob\_score=False, random\_state=61, verbose=0, warm\_start=False)

rf\_cls.fit(X\_train,y\_train)

y\_pred = rf\_cls.predict(X\_test)

print(accuracy\_score(y\_pred,y\_test))

print(confusion\_matrix(y\_test,y\_pred))

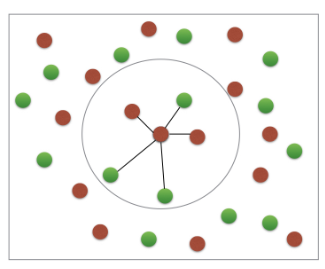
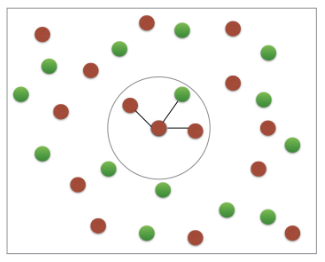
print(classification\_report(y\_test,y\_pred))

**Knn – for classification**

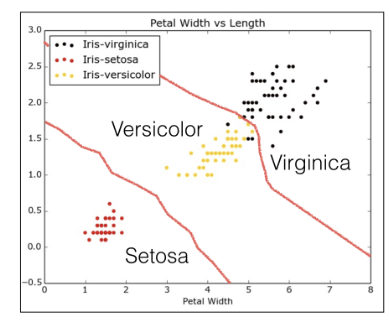
Have a set of labeled data and a set of unlabeled data. We want to predict what the labels will be in the unlabeled data. Labeled data is the training set. Unlabeled data is the test set.

k-Nearest Neighbors:

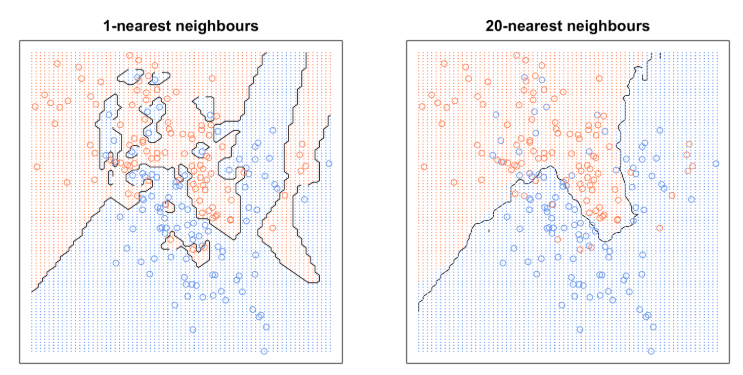
* Basic Idea: Predict the label of a data point by
  + - Looking at the ‘k’ closets labeled data point
    - Taking majority vote
    - k is a **hyperparameter**



k=3 k=5



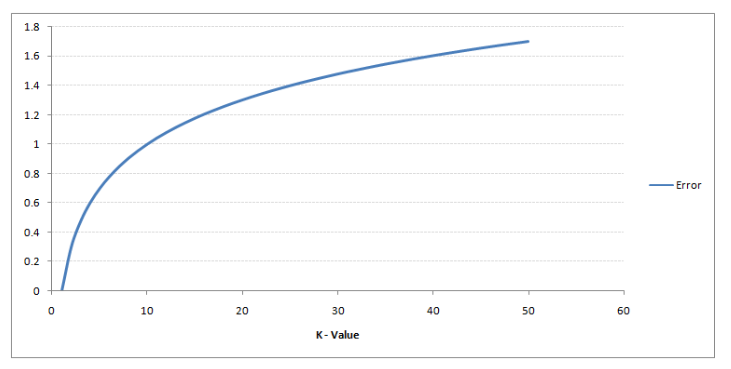
Creates decision boundaries. If data point is in lower left then Setosa, middle is Versicolor and upper right is Virinica.



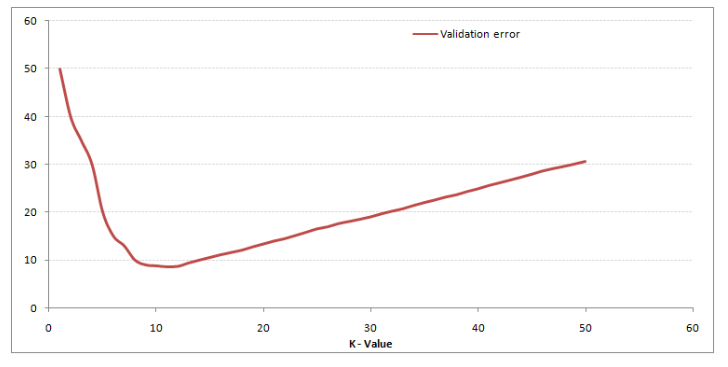
Low k results in higher variance and low bias. Higher k results in lower variance and higher bias. These are just the results of using knn where k=1 and k=20. Large k results in overfitting.

The training phase of the algorithm consists only of storing the feature vectors and class labels of the training samples. That’s it…it is very fast.

Predictions are made for a new instance (x) by searching through the entire training set for the K most similar instances (the neighbors) and summarizing the output variable for those K instances. For regression this might be the mean output variable, in classification this might be the mode (or most common – majority vote) class value.



During training, the lower k the better the error rate (smaller)



During testing, there is an optimal value for k that you have to find.

#sample code for knn

k=3

knn\_cls = KNeighborsClassifier(n\_neighbors=k)

knn\_cls.fit(X\_train,y\_train)

y\_pred = knn\_cls.predict(X\_test)

#function for producing metrics.

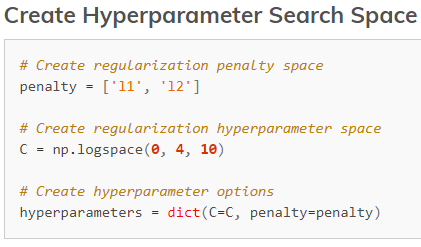
classification\_metrics(X\_test, y\_test, y\_pred, knn\_cls,'green','Knn k='+str(k), 'Knn ROC Curve\nBaseball Classification (1B vs SS)',False)

**Logistic Regression for binary classification**

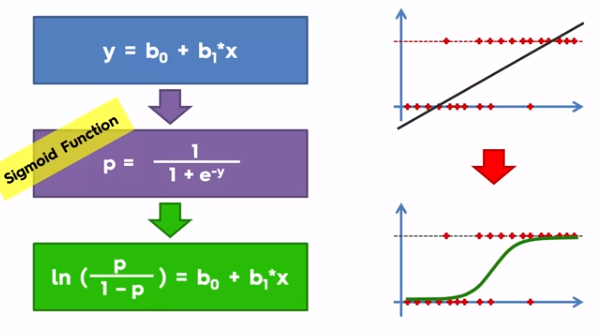
* Logistic Regression outputs probabilities
* If the probability p is greater than 0.5, then data is labeled 1
* If probability p is less than 0.5, then it is labeled 0
* Logistic Reg. produces a linear decision boundary.
* Can vary threshold p for a value to get Receiver Operating Characteristic curve



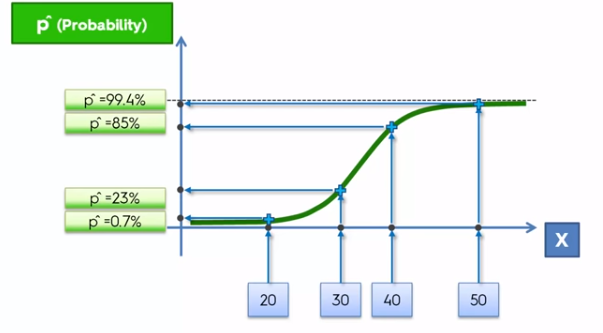
Logistic Regression creates a linear boundary.



Hyperparameters for logistic regression

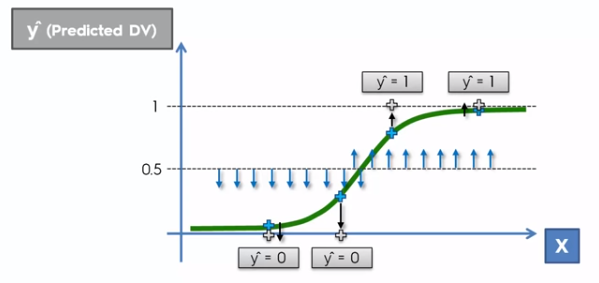


Green box is logistic regression (best fitting line)



Line predicts probability and use probability as a score

To get predictions, start with p = 0.5



# code sample for logistic regression classification

lr\_cls = LogisticRegression()

lr\_cls.fit(X\_train,y\_train)

y\_pred = lr\_cls.predict(X\_test)

classification\_metrics(X\_test, y\_test, y\_pred, lr\_cls,'orange','LogReg', 'ROC Diagram\nBaseball Classification (1B vs SS)',True)

**# Hyperparameter tuning with logistic regression**

from sklearn.linear\_model import LogisticRegression

from sklearn.model\_selection import GridSearchCV

# Setup the hyperparameter grid

c\_space = np.logspace(-5, 8, 15)

param\_grid = {'C': c\_space}

# Instantiate a logistic regression classifier: logreg

logreg = LogisticRegression()

# Instantiate the GridSearchCV object: logreg\_cv

logreg\_cv = GridSearchCV(logreg, param\_grid, cv=5)

# Fit it to the data

logreg\_cv.fit(X,y)

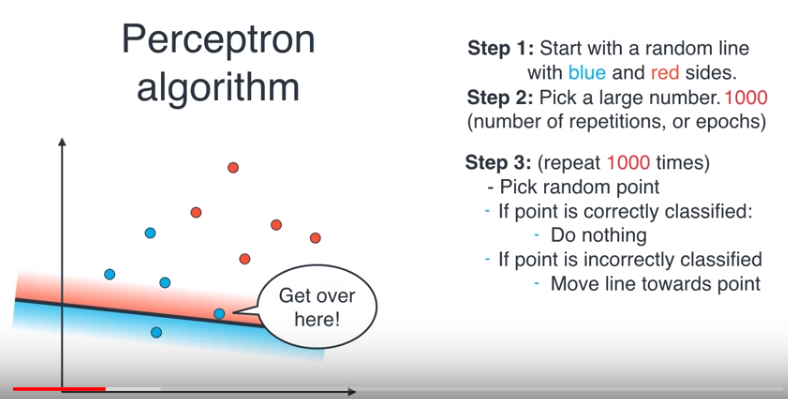
# Print the tuned parameters and score

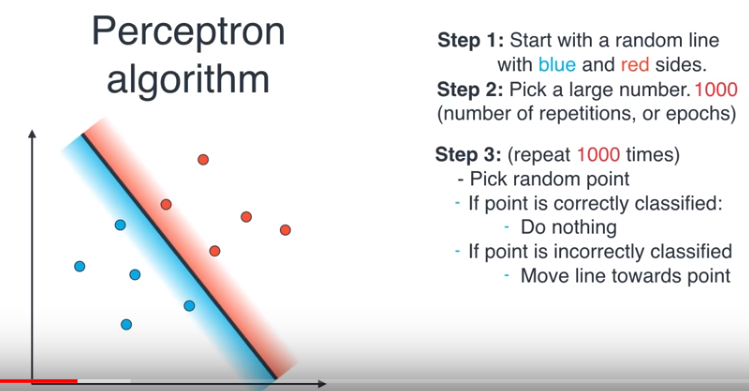
print("Tuned Logistic Regression Parameters: {}".format(logreg\_cv.best\_params\_))

print("Best score is {}".format(logreg\_cv.best\_score\_))

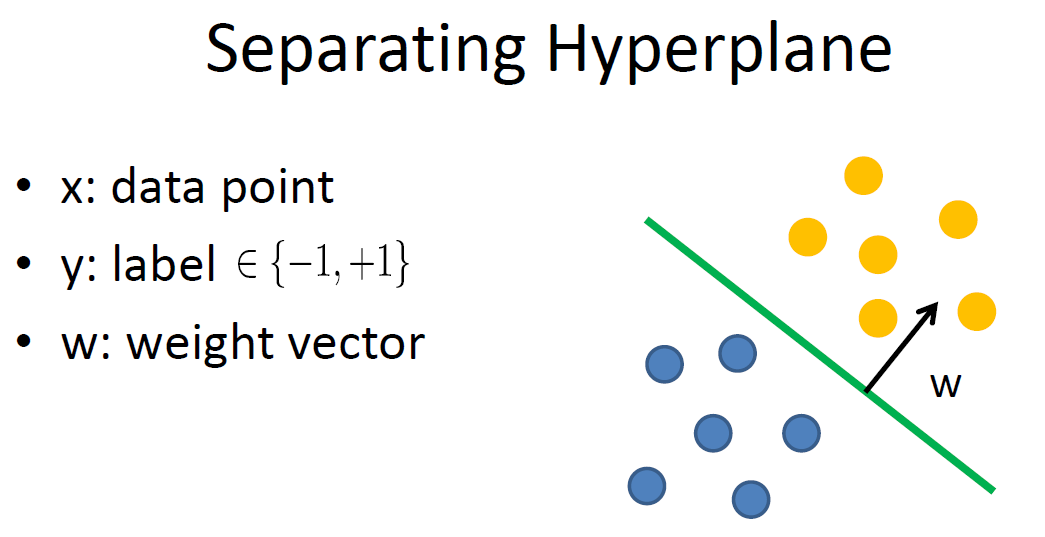
**SVM for classification and regression problems (RBF, linear, non-linear)**

**Perceptron Algorithm:**

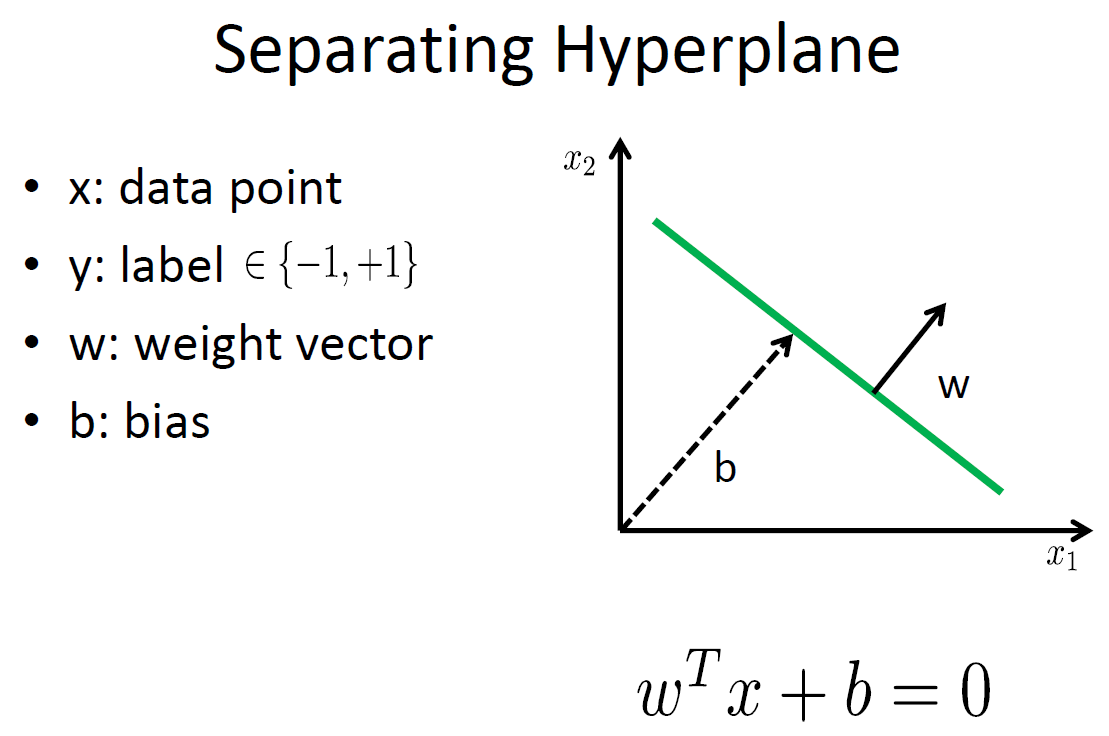
****

****

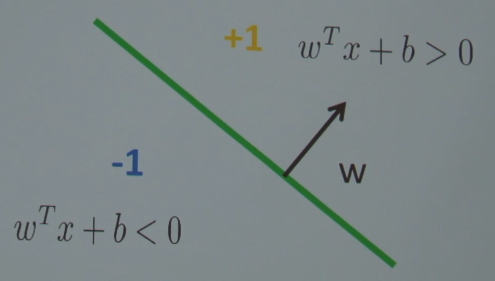
Don’t need to store all your training data (like Knn). All you need to store is vector information.



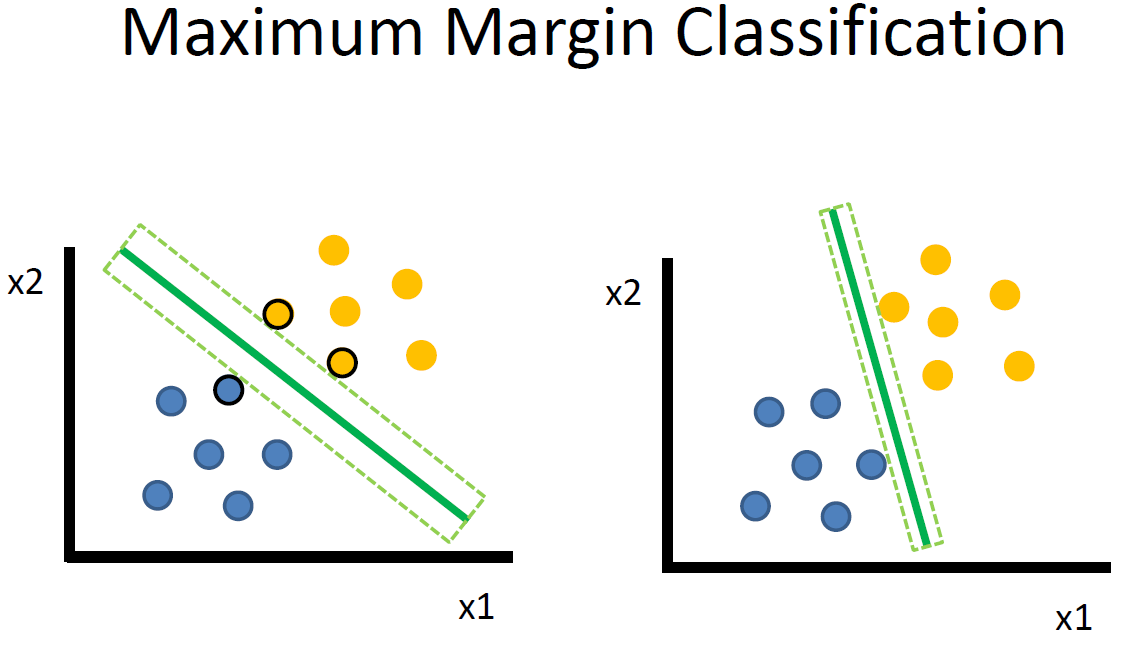
Want to estimate w. w is the orientation of the hyperplane.



Need to know w (weight factor) and b (bias). Optimize randomly and “wiggle” it into optimal hyperplane.

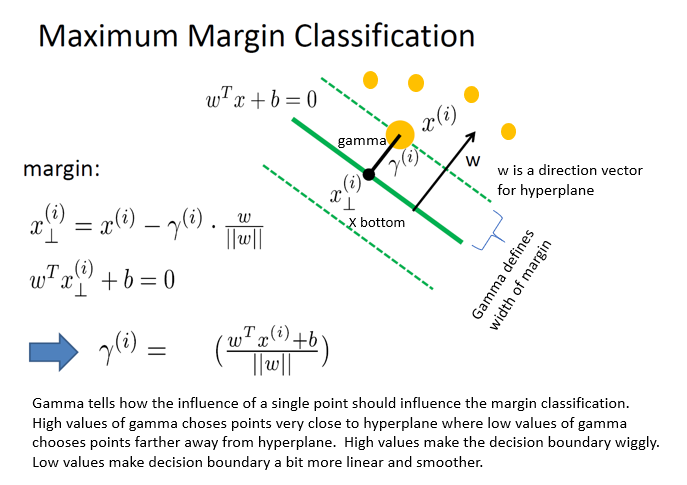


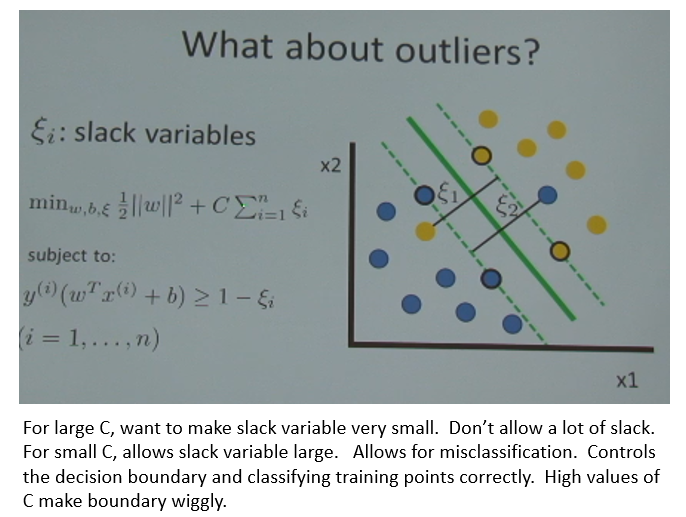
Just need to evaluate this equation (no longer need training data)



Left max margin is better than right. Margins are for the points close to the vector.

C and gamma are **hyperparameters** for SVM classification. The support vector machine only cares about the points that are close the hyperplane within the margins.





# sample code

params = {

'C': [0.1,1,10],

'gamma': [0.001, 0.01, 0.1, 1]

}

svm\_reg1 = SVR(kernel='rbf')

gssvm = GridSearchCV(svm\_reg1, param\_grid=params, cv=3)

gssvm.fit(X, y)

y\_pred = gssvm.predict(X\_test)

lr\_results(df,X\_test,y\_test,y\_pred,path,'OPSpredictionsSVM\_GS.csv',stats\_list,gssvm)

print(gssvm.best\_params\_)

print(gssvm.best\_score\_)

print(np.sqrt(np.abs(gssvm.best\_score\_)))

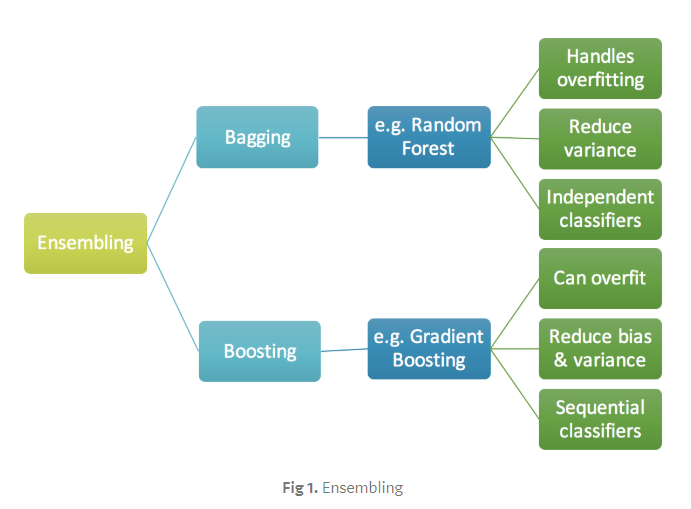
**Boosting**

**“Boosting is one of the most powerful learning ideas introduced in the last twenty years.”** Hastie-Tibshirani-Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*,

Springer (2009)

**Boosting**is an ensemble technique in which the predictors are not made independently, but sequentially.

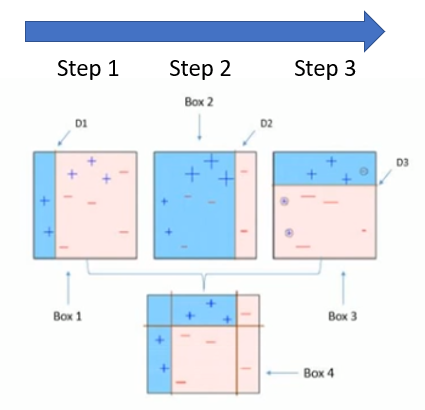
Gradient boosting is a machine learning technique for regression and classification problems, which produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees. (Wikipedia definition)



* Boosting algorithms convert weak leaners into strong learners
* A weak learner is one which is slightly better than random guessing
* Boosting is a sequential process (trees are grown using information from a

Previous grown tree one after another)

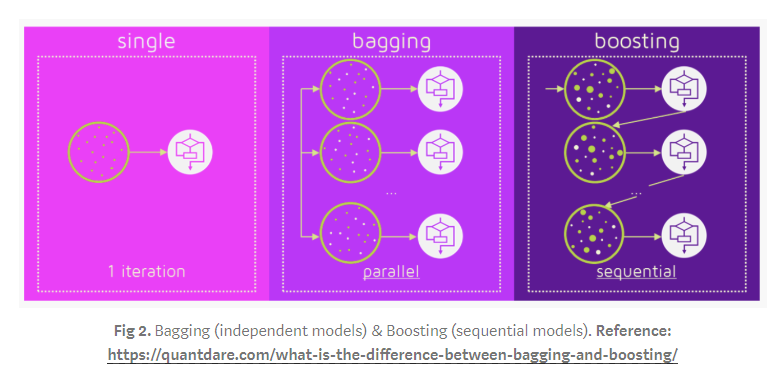
* The process learns from data and tries to improve at each iteration



Takes three weak classifications (box 1, 2, 3) from each tree and uses them to create a strong classification. Classification trees are created in sequence, based upon results from previous one.

Algorithm:

• We first model data with simple models and analyze data for errors.  
• These errors signify data points that are difficult to fit by a simple model.  
• Then for later models, we particularly focus on those hard to fit data to get them right.  
• In the end, we combine all the predictors by giving some weights to each predictor.



#sample XGBoost Regressor

reg\_xgb = xgb.XGBRegressor(objective ='reg:squarederror',

colsample\_bytree=0.6,

learning\_rate=0.2,

max\_depth=3,

n\_estimators=60,

subsamples=0.6,

alpha=1,

gamma=0.001

)

reg\_xgb.fit(X\_train, y\_train)

y\_pred = reg\_xgb.predict(X\_test)

lr\_results(df,X\_test,y\_test,y\_pred,path,'OPSpredictionsXGB.csv',stats\_list,reg\_xgb)

# XGBoost classifier

xgb\_cls = XGBClassifier()

xgb\_cls.fit(X\_train,y\_train)

y\_pred = xgb\_cls.predict(X\_test)

#customer output metrics function

classification\_metrics(X\_test, y\_test, y\_pred, xgb\_cls,'red','XGB', 'XGBoost ROC Curve',False)