Parametric vs Non-Parametric ML Algorithms

Parametric Algorithms

- Parametric algorithms make certain assumptions about the underlying data distribution or relationship between features and target variables. These algorithms have a fixed number of parameters that need to be learned from the training data. Once these parameters are learned, the model structure is determined, and the training data is no longer needed.
- Parametric algorithms are generally computationally efficient, especially when dealing with large datasets, but they might not capture complex relationships in the data well if the underlying assumptions do not hold.

Non-Parametric Algorithms

- Non-parametric algorithms make fewer assumptions about the underlying data distribution. These algorithms do not have a fixed number of parameters and can adjust their model complexity based on the amount of training data. They can capture more complex relationships but might require a larger amount of data to generalize effectively.
- Non-parametric algorithms can better capture intricate patterns in the data, but they might be computationally expensive and prone to overfitting if not properly regularized.

In summary, the choice between parametric and non-parametric algorithms depends on the nature of the data, the complexity of the underlying relationships, the amount of available data, and computational considerations. Parametric models are simpler and more interpretable but might not capture complex relationships well. Non-parametric models can capture complexity but require more data and can be computationally intensive.

Regression vs Classification

Regression is a type of supervised learning task where the goal is to predict a continuous numerical value or quantity. In other words, the output of a regression model is a real number that can be any value within a certain range. The main objective is to model the relationship between the input features (also known as independent variables or predictors) and the continuous target variable.

Examples of regression tasks include:

- Predicting house prices based on features like square footage, number of bedrooms, etc.
- Forecasting stock prices over time.
- Estimating a person's age based on various biometric measurements.

Common regression algorithms include linear regression, polynomial regression, support vector regression, and neural networks.

Classification is another type of supervised learning task where the goal is to assign input data to a specific category or class from a predefined set of classes. The output of a classification model is a discrete class label, indicating which category the input data belongs to.

Examples of classification tasks include:

- Email spam detection (categorizing emails as spam or not spam).
- Image classification (identifying whether an image contains a cat or a dog).
- Medical diagnosis (determining whether a patient has a certain disease based on symptoms).

Common classification algorithms include decision trees, random forests, support vector machines, k-nearest neighbors, and deep learning models like convolutional neural networks (CNNs) for image classification.

Supervised Learning

Supervised learning is a type of machine learning where the algorithm learns from labeled training data. In this approach, the training data consists of input features and their corresponding target labels. The algorithm's goal is to learn a mapping between the input features and the target labels so that it can make accurate predictions on new, unseen data.

Examples of supervised learning tasks include:

- Regression: Predicting a continuous numerical value (e.g., predicting house prices).
- Classification: Assigning inputs to predefined categories (e.g., email spam detection, image classification).

In supervised learning, the algorithm is "supervised" by having access to the correct answers during training, which enables it to learn patterns and relationships in the data.

Unsupervised Learning

Unsupervised learning involves training an algorithm on unlabeled data, where the algorithm's objective is to find patterns or structures in the data without explicit target labels. The algorithm seeks to discover inherent relationships, groupings, or distributions within the data.

Examples of unsupervised learning tasks include:

- Clustering: Grouping similar data points together (e.g., customer segmentation based on purchasing behavior).
- Dimensionality Reduction: Reducing the number of input features while retaining meaningful information (e.g., Principal Component Analysis).
- Anomaly Detection: Identifying unusual or rare instances in the data.

Unsupervised learning is used when there is no predefined notion of correct answers, and the algorithm explores the data's intrinsic properties to create meaningful representations.

Semi-Supervised Learning

Semi-supervised learning is a hybrid approach that combines elements of both supervised and unsupervised learning. In this scenario, the training data contains a mix of labeled and unlabeled examples. The algorithm uses the labeled data to learn patterns and relationships just like in supervised learning, but it also leverages the unlabeled data to improve its understanding of the data distribution and uncover additional patterns.

Semi-supervised learning is especially useful when obtaining large amounts of labeled data is expensive or time-consuming. By using a small amount of labeled data in combination with a larger amount of unlabeled data, the algorithm can potentially achieve better performance than pure supervised learning.

In summary, supervised learning involves learning from labeled data, unsupervised learning focuses on discovering patterns in unlabeled data, and semi-supervised learning combines both labeled and unlabeled data to improve learning outcomes. The choice of which approach to use depends on the problem at hand, the availability of labeled data, and the desired outcomes.

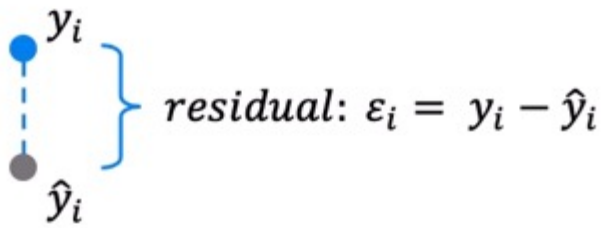
Linear Regression

Linear regression is a fundamental statistical and machine learning technique used for modeling the relationship between a dependent variable (also called the target or response variable) and one or more independent variables (also called predictors or features). It assumes a linear relationship between the independent variables and the dependent variable. The goal of linear regression is to find the best-fitting linear equation that describes this relationship.

A **causal relationship** is a connection between two or more variables where changes in one variable directly cause changes in another variable. In other words, a causal relationship implies that changes in the independent variable lead to changes in the dependent variable, and there is a cause-and-effect mechanism at play.

Simple Linear Regression-

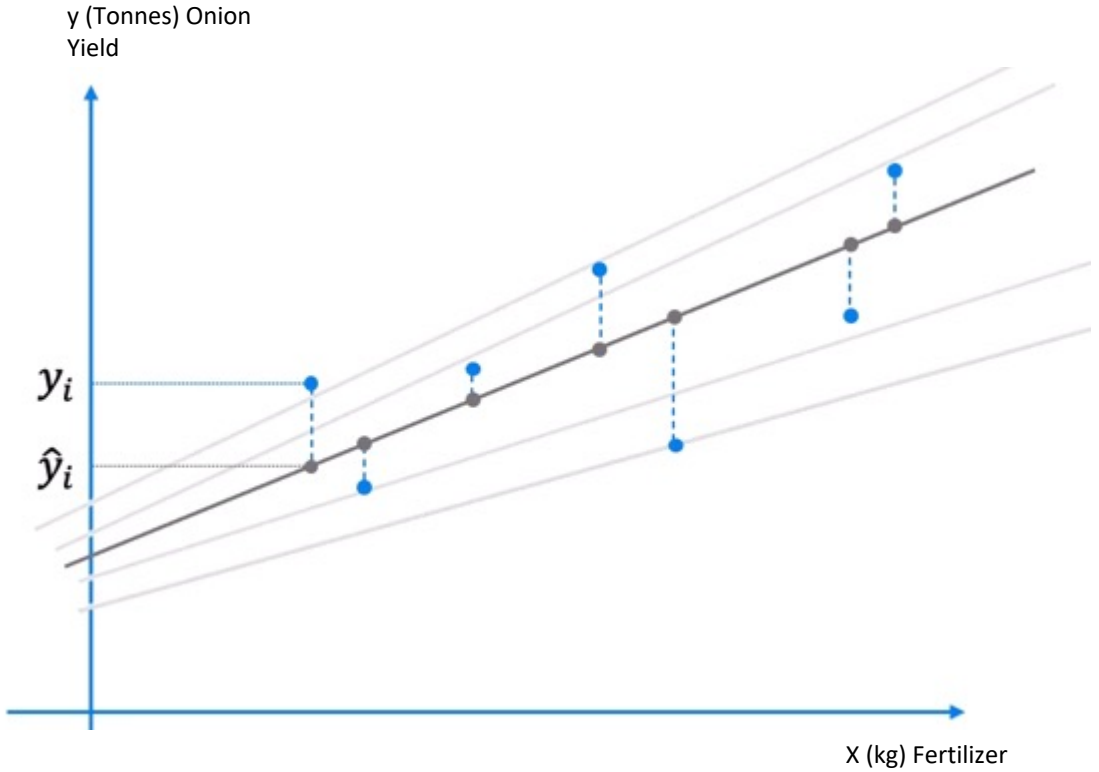
Ordinary Least Squares:



$\hat{y} = b_0 + b_1 X_1$

b_0, b_1 such that:

$SUM(y_i - \hat{y}_i)^2$ is minimized



Least Squares stands for minimum squares error or SSE

Least Square Linear Equation

The least squares linear equation is the result of fitting a linear model to data in such a way that it minimizes the sum of the squared differences between the observed values and the values predicted by the linear equation. This method is known as least squares regression and is commonly used to find the best-fitting line through a set of data points.

The equation of a straight line in its general form is:

$$y = mx + b$$

Where:

- y is the dependent variable (the variable you're trying to predict),
- x is the independent variable (the variable you're using to make predictions),
- m is the slope of the line,
- b is the y-intercept (the value of y when x is 0).

In the context of least squares regression, the goal is to find the values of m and b that minimize the sum of the squared differences between the observed y values and the predicted y values (based on the linear equation). Mathematically, the least squares method aims to minimize the following sum:

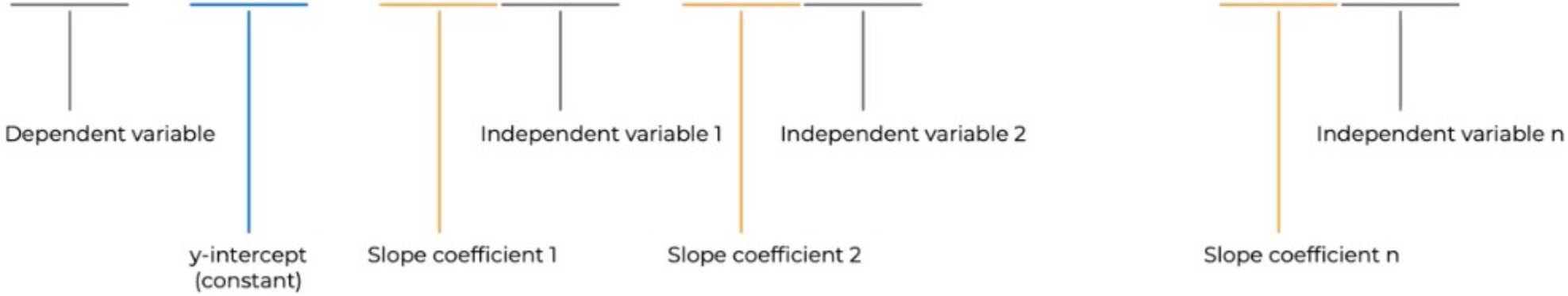
$$\sum_{i=1}^{n} (y_i - (mx_i + b))^2$$

Where n is the number of data points, (x_i, y_i) are individual data points, and $(mx_i + b)$ is the predicted y value for a given x_i.

The values of m and b that minimize the sum of squared differences can be calculated using various methods, including calculus and matrix algebra. The result is the equation of the least squares regression line that best fits the data.

Multiple Linear Regression

$$\hat{y} = b_0 + b_1 X_1 + b_2 X_2 + \cdots + b_n X_n$$

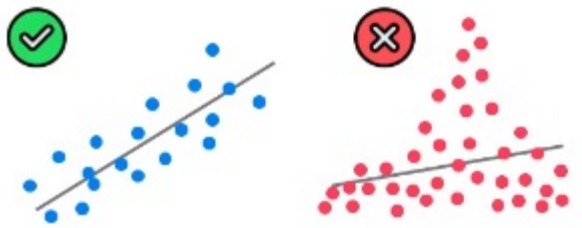


$$\text{Onion}\,[t] = 8t + 3\,\frac{t}{kg} \times Fertilizer[kg] - 0.54\,\frac{t}{°C} \times AvgTemp[°C] + 0.04\,\frac{t}{mm} \times Rain[mm]$$

Assumptions of Linear Regression

Linearity-linear relationship between Y and each X

Linear regression needs the relationship between the independent and dependent variables to be linear. It is also important to check for outliers since linear regression is sensitive to outlier effects. The linearity assumption can best be tested with scatter plots, the following two examples depict two cases, where no and little linearity is present.

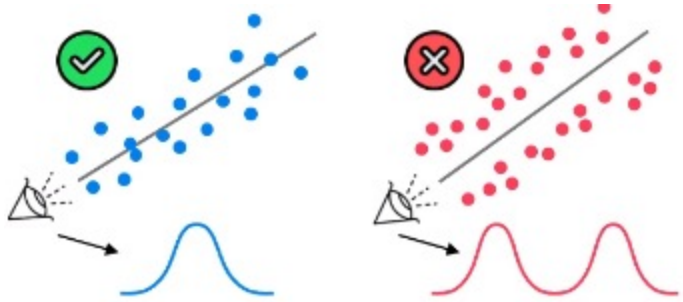


Fixes for Linearity:

- Run a non-linear regression -Classification and Regression Trees, Naive Bayes, K-Nearest Neighbors, Learning Vector Quantization and Support Vector Machines.
- Transform the relationship
  - Exponential transformation
  - Logarithmic transformation

Multivariate Normality-

linear regression analysis requires that the errors between observed and predicted values (i.e., the residuals of the regression) should be normally distributed. This assumption may be checked by looking at a histogram



Fixes

- Central limit theorem

Assumptions of Linear Regression

No Multicollinearity-

It assumes that the independent variables are not highly correlated with each other.

Multicollinearity may be checked multiple ways:

1) Correlation matrix – When computing a matrix of Pearson's bivariate correlations among all independent variables, the magnitude of the correlation coefficients should be less than .80.

2) Variance Inflation Factor (VIF) – The VIFs of the linear regression indicate the degree that the variances in the regression estimates are increased due to multicollinearity. VIF values higher than 5 or 10 indicate that multicollinearity is a problem.

If multicollinearity is found in the data, one possible solution is to center the data. To center the data, subtract the mean score from each observation for each independent variable. However, the simplest solution is to identify the variables causing multicollinearity issues (i.e., through correlations or VIF values) and removing those variables from the regression.

No Endogeneity-

Endogeneity refers to a situation in which there is a correlation between the independent variables and the error term in a regression model. This can lead to biased and inconsistent coefficient estimates, which undermines the reliability of the results. In other words, endogeneity violates the assumption that the independent variables are not affected by the error term. If they are correlated, it can lead to omitted variable bias

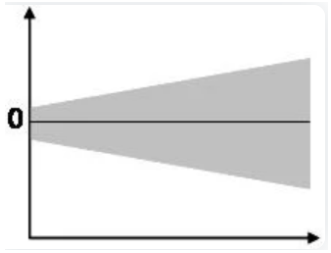
Omitted variable bias is a common issue in regression analysis that occurs when an important variable is left out of a regression model. This omission can lead to biased and unreliable coefficient estimates for the included variables, as well as incorrect conclusions about their relationships with the dependent variable. Omitted variable bias arises because the omitted variable might be correlated with both the included independent variables and the dependent variable.

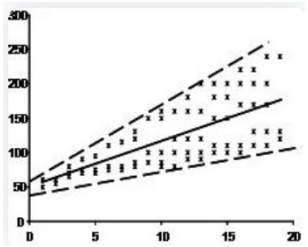
Assumptions of Linear Regression

Homoscedasticity-

This assumption states that the variance of error terms are similar across the values of the independent variables. A plot of standardized residuals versus predicted values can show whether points are equally distributed across all values of the independent variables.

A scatterplot of residuals versus predicted values is good way to check for homoscedasticity. There should be no clear pattern in the distribution; if there is a cone-shaped pattern (as shown below), the data is heteroscedastic.





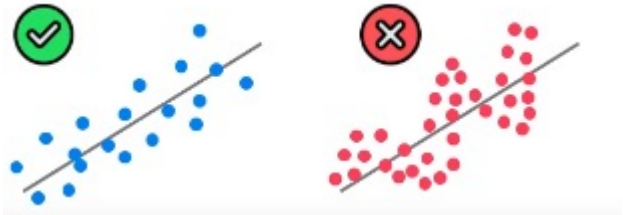
Fixes:

- Check for Omitted variable bias
- Look for outliers
- Log transformation

No Autocorrelation-

Autocorrelation is another assumption of linear regression that needs to be met. Autocorrelation occurs when the residuals are not independent from each other**.** In other words, it is a measure of similarity or correlation between adjacent data points, where data points are affected by the values of points that came before. For instance, this typically occurs in stock prices, where the price is not independent from the previous price. Ideally, model errors should be independent and identically distributed and thus should have no patterns in them**.** Autocorrelation can cause problems like invalid linear regression conclusions or interpretations.

To test the linear regression model for autocorrelation, we can use the **Durbin-Watson test**. Durbin-Watson's d tests the null hypothesis that the residuals are not linearly autocorrelated. Values around 2 indicate no autocorrelation, and as a rule of thumb, values of $1.5 < d < 2.5$ show that there is no autocorrelation in the data**.** If autocorrelation is present, we need to investigate the omission of a key predictor or use methods like autoregressive models to estimate the regression parameters of the Y versus X relationship



Math Behind Linear Regression

Linear regression is a statistical method used to model the relationship between a dependent variable and one or more independent variables by fitting a linear equation to the observed data. The goal is to find the best-fitting line that minimizes the sum of squared differences between the observed values and the values predicted by the linear equation. The mathematical foundation of linear regression involves concepts from linear algebra and calculus.

Here's a step-by-step overview of the mathematical process behind simple linear regression (one independent variable):

1. **Model Assumption:**
   The model assumes a linear relationship between the dependent variable y and the independent variable x:
   $y = mx + b + \varepsilon$
   where m is the slope, b is the intercept, and ε represents the error term (deviation from the true line).

Math Behind Linear Regression

2. **Residuals and Error:**

   The error for each data point is the difference between the observed y value and the predicted y value from the linear equation:

   $$\varepsilon = y - (mx + b)$$

3. **Objective Function - Minimization of Residuals:**

   The goal of linear regression is to find the values of m and b that minimize the sum of squared residuals (also known as the least squares criterion):

   $$\text{Minimize} \sum_{i=1}^{n} \varepsilon_i^2$$

   This can be expanded and rewritten as:

   $$\text{Minimize} \sum_{i=1}^{n} (y_i - (mx_i + b))^2$$

4. **Minimization Using Calculus:**

   To find the values of m and b that minimize the sum of squared residuals, calculus is used. Taking partial derivatives with respect to m and b, setting them to zero, and solving the resulting equations gives the optimal values:

   $$m = \frac{n(\sum_{i-1}^{n} x_i y_i) - (\sum_{i-1}^{n} x_i)(\sum_{i-1}^{n} y_i)}{n(\sum_{i-1}^{n} x_i^2) - (\sum_{i-1}^{n} x_i)^2}$$

   $$b = \frac{(\sum_{i-1}^{n} y_i)(\sum_{i-1}^{n} x_i^2) - (\sum_{i-1}^{n} x_i y_i)(\sum_{i-1}^{n} x_i)}{n(\sum_{i-1}^{n} x_i^2) - (\sum_{i-1}^{n} x_i)^2}$$

Math Behind Linear Regression

5. **Interpretation of Coefficients:**

   The coefficient m represents the change in the dependent variable y for a unit change in the independent variable x. The intercept b represents the value of y when x is zero.

6. **Fitting the Line:**

   Using the calculated m and b, the linear regression line $y = mx + b$ is fitted to the data.

This explanation covers the basics of simple linear regression. For multiple linear regression (when there are more than one independent variable), the process involves matrix notation and vector calculus to extend the concept to higher dimensions.

Decomposition Variability

The decomposition of variability refers to the process of breaking down the total variability in a dataset or a statistical model into its constituent parts or sources. In the context of regression analysis, the variability in the dependent variable can often be decomposed into different components that help us understand the factors contributing to the observed variations.

Let's explore how variability can be decomposed in the context of regression analysis:

1. **Total Variability (Total Sum of Squares, SST):** This represents the total variation in the dependent variable. It measures the differences between the observed values of the dependent variable and their mean.

$$\text{SST} = \sum_{i=1}^{n}(y_i - \bar{y})^2$$

Where y_i is the observed value of the dependent variable for observation i, and $\bar{y}$ is the mean of the observed values.

Explained Variability (Regression Sum of Squares, SSR): This represents the variation in the dependent variable that is explained by the regression model. It measures how well the model fits the data.

$$\text{SSR} = \sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2$$

Where $\hat{y}_i$ is the predicted value of the dependent variable for observation i.

Unexplained Variability (Residual Sum of Squares, SSE): This represents the variation in the dependent variable that is not explained by the regression model. It captures the errors or residuals of the model.

$$\text{SSE} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Decomposition of Variability Identity: The decomposition of variability identity shows the relationship between the total variability, explained variability, and unexplained variability:

$$\text{SST} = \text{SSR} + \text{SSE}$$

In other words, the total variability can be partitioned into the variability explained by the model (regression) and the variability that remains unexplained (residuals).

$$\sum_{i=1}^{n}(y_i - \bar{y})^2 = \sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2 + \sum_{i=1}^{n} e_i^2$$

Decomposition Variability-AIC

The **Akaike information criterion (AIC)** is a mathematical method for evaluating how well a model fits the data it was generated from. In statistics, AIC is used to compare different possible models and determine which one is the best fit for the data. Once you've created several possible models, you can use AIC to compare them. Lower AIC scores are better, and AIC penalizes models that use more parameters. So if two models explain the same amount of variation, the one with fewer parameters will have a lower AIC score and will be the better-fit model.

How to compare models using AIC

AIC determines the relative information value of the model using the maximum likelihood estimate and the number of parameters (independent variables) in the model. The formula for AIC is:

$$AIC=2k+n\log(RSS/n) \quad (k=d+2)$$

K is the number of independent variables used and L is the log-likelihood estimate (a.k.a. the likelihood that the model could have produced your observed y-values). The default K is always 2, so if your model uses one independent variable your K will be 3, if it uses two independent variables your K will be 4, and so on.

To compare models using AIC, you need to calculate the AIC of each model. If a model is more than 2 AIC units lower than another, then it is considered significantly better than that model.

Decomposition Variability-BIC

The Bayesian Information Criterion (BIC), also known as the Schwarz criterion, is another model selection criterion that, like the Akaike Information Criterion (AIC), helps you choose the most appropriate model among a set of candidates. BIC is particularly useful in situations where you want to avoid overfitting and select a model that is not only well-fitting but also parsimonious.

The BIC formula is given by:

$$\mathrm{BIC} = -2\ln(L) + k\ln(n)$$

Where:

- L is the likelihood of the data given the model.
- k is the number of estimated parameters in the model.
- n is the sample size.

Like AIC, the BIC penalizes models for having more parameters. However, the penalty term in BIC is larger because it includes the natural logarithm of the sample size. This means that BIC tends to favor simpler models more strongly than AIC does.

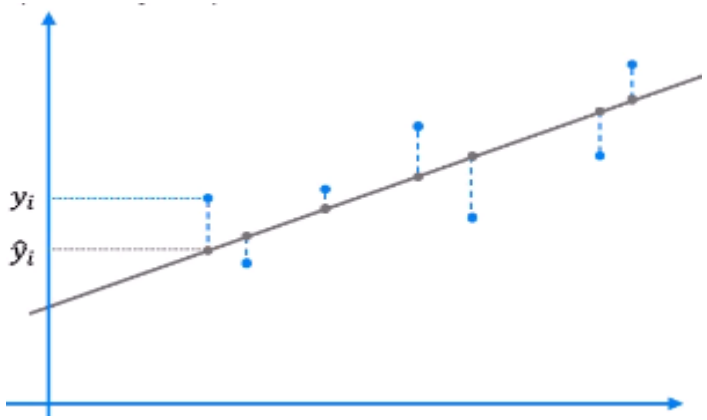
Here's how to use BIC for model selection:

1. **Fit Models:** Fit different models to your data, just like with AIC.
2. **Calculate BIC:** Calculate the BIC value for each model using the formula.
3. **Compare BIC:** Compare the BIC values of the models. The model with the lowest BIC is preferred, as it indicates a better trade-off between model fit and complexity.
4. **Interpretation:** Similar to AIC, a difference in BIC values of around 2 or more between models is generally considered meaningful. Lower BIC values indicate a better fit.
5. **Sample Size Effect:** BIC is more sensitive to sample size than AIC. As the sample size increases, the penalty for model complexity in BIC becomes more significant, encouraging the selection of simpler models.
6. **Model Complexity:** Because of the stronger penalty for complexity in BIC, it is often more conservative in model selection, favoring simpler models even more.
7. **Similarity to AIC:** BIC and AIC both aim to balance model fit and complexity but apply different penalties. The choice between them depends on the context and the trade-off you wish to make.

R-Squared

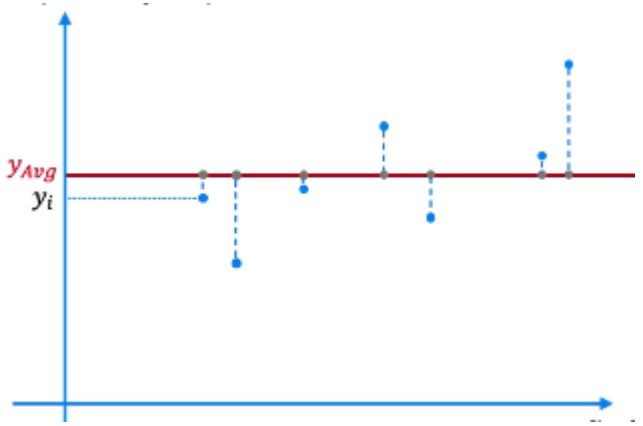
R-squared (Coefficient of Determination) is a statistical measure used to assess the proportion of the variance in the dependent variable that is explained by the independent variables in a regression model. It provides insight into how well the independent variables account for the variability observed in the dependent variable. R-squared is commonly used to evaluate the goodness of fit of a regression model.

Regression



$$SS_{res} = SUM(y_i - \hat{y}_i)^2$$

Average



$$SS_{tot} = SUM(y_i - y_{avg})^2$$

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

Rule of Thumb

1.0 = Perfect fit (suspicious)
~0.9 = Very good
<0.7 = Not great
<0.4 = Terrible
<0 = Model makes no sense for this data

Adjusted R-Squared

Adjusted R-squared penalizes excessive use of variables so its value will be less than R-squared

Adjusted R-squared is a modification of the regular R-squared (coefficient of determination) that takes into account the number of independent variables in a regression model. It addresses a limitation of the standard R-squared, which tends to increase as more independent variables are added to the model, even if those variables do not significantly improve the model's explanatory power.

The formula for the adjusted R-squared is:

$$\text{Adjusted } \mathrm{R}^2 = 1 - \frac{(1-R^2)\cdot(n-1)}{n-k-1}$$

Where:

- R^2 is the regular R-squared value.
- n is the number of observations in the dataset.
- k is the number of independent variables in the model.

Feature Scaling

Feature Scaling is a technique to standardize the independent features present in the data in a fixed range. It is performed during the data pre-processing to handle highly varying magnitudes or values or units.
If feature scaling is not done, then a machine learning algorithm tends to weigh greater values, higher and consider smalller values as the lower values, regardless of the unit of the values.

Why use Feature Scaling?

- Scaling guarantees that all features are on a comparable scale and have comparable ranges.
- Algorithm performance improvement: When the features are scaled, several machine learning methods, including gradient descent-based algorithms, distance-based algorithms (such k-nearest neighbours), and support vector machines, perform better or converge more quickly.
- Preventing numerical instability: Numerical instability can be prevented by avoiding significant scale disparities between features.
- Scaling features makes ensuring that each characteristic is given the same consideration during the learning process. Without scaling, bigger scale features could dominate the learning, producing skewed outcomes.

Normalization or Min-Max Scaling is used to transform features to be on a similar scale. The new point is calculated as:
`X_new = (X - X_min)/(X_max - X_min)`
This scales the range to [0, 1] or sometimes [-1, 1]

Standardization or Z-Score Normalization is the transformation of features by subtracting from mean and dividing by standard deviation. This is often called as Z-score.
`X_new = (X - mean)/Std`
The range is not bound but usually between [-3,+3]

Feature Scaling

Normalized Data Vs Standardized Data

- Normalization is used when the data doesn't have Gaussian distribution whereas Standardization is used on data having Gaussian distribution.
- Normalization scales in a range of [0,1] or [-1,1]. Standardization is not bounded by range.
- Normalization is highly affected by outliers. Standardization is slightly affected by outliers.
- Normalization is considered when the algorithms do not make assumptions about the data distribution. Standardization is used when algorithms make assumptions about the data distribution.

If you see a bell-curve in your data then standardization is more preferable. For this, you will have to plot your data. If your dataset has extremely high or low values (outliers) then standardization is more preferred because usually, normalization will compress these values into a small range.

Feature Selection

- Univariate feature selection is a method used to select the most important features in a dataset. The idea behind this method is to evaluate each individual feature's relationship with the target variable and select the ones that have the strongest correlation.

Regularization

- Regularization is one of the ways to improve our model to work on unseen data by ignoring the less important features.
- Regularization minimizes the validation loss and tries to improve the accuracy of the model.
- It avoids overfitting by adding a penalty to the model with high variance, thereby shrinking the beta coefficients to zero.

There are two types of regularization:

1.Lasso Regularization

2.Ridge Regularization

Bias Variance Trade-off

What is Variance?

Variance tells us about the spread of the data points. It calculates how much a data point differs from its mean value and how far it is from the other points in the dataset.

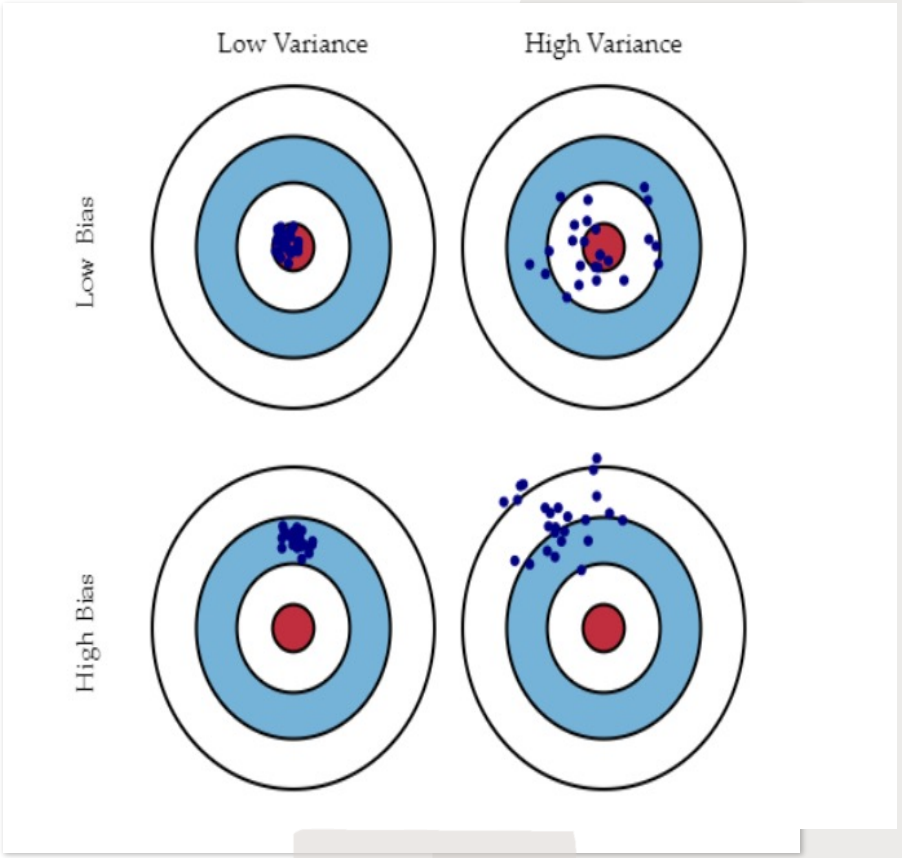
What is Bias?

It is the difference between the average prediction and the target value.

Low bias and low variance will give a balanced model, whereas high bias leads to underfitting, and high variance lead to overfitting.

- **Low Bias:** The average prediction is very close to the target value
- **High Bias:** The predictions differ too much from the actual value
- **Low Variance:** The data points are compact and do not vary much from their mean value
- **High Variance:** Scattered data points with huge variations from the mean value and other data points.

To make a good fit, we need to have a correct balance of bias and variance.



Bias Variance Trade-off

The bias-variance trade-off is a fundamental concept in machine learning and statistical modeling that deals with the relationship between a model's prediction errors due to bias and variance. It helps us understand the sources of errors in a model and guides us in finding the right level of complexity for a model.

Here's how the bias-variance trade-off works:

1. **Bias:**
    1. Bias refers to the error introduced by approximating a real-world problem, which may be complex, by a simplified model. A model with high bias makes strong assumptions about the data and may oversimplify the underlying relationships. It tends to consistently underperform on both the training and test data.
    2. High-bias models are often too simplistic and fail to capture the underlying patterns in the data. They are said to be underfitting the data.
2. **Variance:**
    1. Variance refers to the model's sensitivity to small fluctuations or noise in the training data. A model with high variance is highly flexible and can fit the training data very closely. However, it tends to perform poorly on new, unseen data because it's capturing noise rather than true patterns.
    2. High-variance models can be overly complex and fit the training data too closely. They are said to be overfitting the data.
3. **Trade-off:**
    1. The goal of a machine learning model is to find a balance between bias and variance that minimizes the overall prediction error on new data (test data).
    2. As you increase the complexity of a model (for example, by adding more features or increasing the degree of polynomial regression), the variance tends to increase while bias decreases. This can lead to overfitting.
    3. Conversely, as you decrease the complexity of a model, bias increases while variance decreases. This can lead to underfitting.

The bias-variance trade-off illustrates that there's a "sweet spot" in model complexity where the combined error due to bias and variance is minimized, resulting in better generalization to new data.

To summarize:

- **Underfitting:** High bias, low variance. The model is too simple to capture the underlying patterns in the data.
- **Overfitting:** Low bias, high variance. The model is too complex and fits noise in the data.
- **Balanced Model:** Moderate bias, moderate variance. The model generalizes well to new data.

Finding this balance often involves techniques such as regularization, cross-validation, and careful feature selection. The goal is to choose a model that is complex enough to capture essential patterns in the data but not so complex that it fits noise or idiosyncrasies.

Underfitting vs Overfitting

Underfitting and overfitting are two common issues in machine learning and statistical modeling that relate to how well a model generalizes from the training data to new, unseen data. These issues can affect the performance and reliability of a machine learning model:

1. **Underfitting:**
   - **Definition**: Underfitting occurs when a model is too simple to capture the underlying patterns in the data, resulting in poor performance on both the training data and new, unseen data.
   - **Characteristics**: The model's predictions are overly generalized and do not fit the training data well. It often exhibits high bias and low variance.
   - **Causes:** Underfitting can happen when the model is too basic (e.g., linear regression on nonlinear data) or lacks sufficient complexity to capture the relationships in the data.
   - **Solution:** To address underfitting, you can try increasing the model's complexity (e.g., using a more sophisticated algorithm or adding more features) and ensuring that the model is not too constrained during training.

2. **Overfitting**:
   - **Definition**: Overfitting occurs when a model is excessively complex and learns to memorize the training data, capturing noise and random fluctuations rather than the genuine underlying patterns. As a result, it performs well on the training data but poorly on new data.
   - **Characteristics**: The model fits the training data extremely well, often achieving high accuracy, but it performs poorly on validation or test data. It typically exhibits low bias and high variance.
   - **Causes**: Overfitting can happen when a model is too complex for the available data or when the training dataset is small, leading the model to learn noise in the data.
   - **Solution**: To combat overfitting, you can reduce the model's complexity (e.g., using feature selection, feature engineering, or regularization techniques), increase the size of the training dataset, or employ techniques like cross-validation to assess model performance on unseen data.

Finding the right balance between underfitting and overfitting is crucial for building a model that generalizes well to new data. This is often referred to as achieving the "bias-variance trade-off." Regularization techniques, like L1 and L2 regularization, are commonly used to prevent overfitting, while feature engineering and the use of more complex models can help address underfitting. Proper evaluation techniques, such as cross-validation, are essential for identifying and mitigating these issues in machine learning models.

What is Lasso Regularization (L1)?

- It stands for Least Absolute Shrinkage and Selection Operator
- It adds L1 the penalty
- L1 is the sum of the absolute value of the beta coefficients

Cost function = Loss + λ + Σ ||w||

Here,

Loss = sum of squared residual

λ = penalty

w = slope of the curve

What is Ridge Regularization (L2)

- It adds L2 as the penalty
- L2 is the sum of the square of the magnitude of beta coefficients

Cost function = Loss + λ + Σ $||w||^2$

Here,

Loss = sum of squared residual

λ = penalty

w = slope of the curve

λ is the penalty term for the model. As λ increases cost function increases, the coefficient of the equation decreases and leads to shrinkage.

Comparing Lasso and Ridge Regularization techniques

L1 Regularization	L2 Regularization
Penalty is the absolute value of coefficients	Penalty is the square of the coefficients
Estimate median of the data	Estimate mean of the data
Shrinks coefficients to zero	Shrinks coefficients equally
Can be used for dimension reduction and feature selection	Useful when we have collinear features

Difference between Fit,Transform and Fit_Transform methods

Method	Purpose	Syntax	Example
fit()	Learn and estimate the parameters of the transformation	`estimator.fit(X)`	`estimator.fit(train_data)`
transform()	Apply the learned transformation to new data	`transformed_data = estimator.transform(X)`	`transformed_data = estimator.transform(test_data)`
fit_transform()	Learn the parameters and apply the transformation to new data	`transformed_data = estimator.fit_transform(X)`	`transformed_data = estimator.fit_transform(data)`

Note: In the syntax, `estimator` refers to the specific estimator or transformer object from Scikit-Learn that is being used. `X` represents the input data.

Example: Suppose we have a dataset `train_data` for training and `test_data` for testing. We can use `fit()` to learn the parameters from the training data (`estimator.fit(train_data)`) and then use `transform()` to apply the learned transformation to the test data (`transformed_data = estimator.transform(test_data)`). Alternatively, we can use `fit_transform()` to perform both steps in one (`transformed_data = estimator.fit_transform(data)`).

Fine Tuning Linear Regression model using SGDRegressor

The best hyperparameter values for an `SGDRegressor` model depend on your specific dataset and problem. Hyperparameter tuning is typically done using techniques like grid search or random search in combination with cross-validation to find the optimal hyperparameters for your specific task. However, some guidance on commonly tuned hyperparameters for the `SGDRegressor` model:

1. **Learning Rate (`eta0` or `learning_rate`):** This parameter controls the step size during training. Values typically range from 0.01 to 0.1 or smaller. You may need to experiment to find the best learning rate for your specific dataset.

2. **Number of Epochs (`max_iter`):** This parameter determines the number of iterations the algorithm will run for. You should set it to a sufficiently large value to ensure convergence but not too large to avoid overfitting.

3. **Regularization (`alpha`):** The `alpha` parameter controls the amount of regularization applied to the model. Values are typically set logarithmically, e.g., [0.0001, 0.001, 0.01, 0.1, 1.0]. A smaller `alpha` encourages the model to fit the training data closely, while a larger `alpha` encourages a simpler model.

4. **Penalty (`penalty`):** This parameter specifies the type of regularization. Common choices are 'l2' (ridge) and 'l1' (lasso). You can also use 'elasticnet' to combine both 'l2' and 'l1' penalties.

5. **Loss Function (`loss`):** The `loss` parameter determines the loss function used during training. Common options include 'squared_loss' for ordinary least squares (OLS) regression and 'huber' for robust regression.

6. **Shuffling (`shuffle`):** Setting this to `True` can help with convergence by shuffling the training data at the start of each epoch.

7. **Mini-Batch Size (`batch_size`):** You can specify a mini-batch size if you want to use mini-batch gradient descent. A common value is 32 or 64, but this can depend on your dataset size.

8. **Random Seed (`random_state`):** Set a random seed for reproducibility.

Remember that the best hyperparameters depend on the nature of your dataset, its size, and the specific problem you're trying to solve. It's a good practice to perform hyperparameter tuning using techniques like cross-validation to find the best combination of hyperparameters for your particular case. Additionally, you may want to consider feature engineering and data preprocessing as these can also significantly impact your model's performance.

Cross Validation

Cross-validation is a widely used technique in machine learning for assessing the performance and generalization of a predictive model. It helps to estimate how well a model will perform on unseen data by dividing the dataset into multiple subsets, training the model on different subsets, and evaluating its performance on the remaining data. This process helps in detecting issues like overfitting or underfitting and provides a more reliable estimate of a model's performance compared to a single train-test split.

Here are the key steps involved in cross-validation:

1. **Data Splitting:** The dataset is divided into "folds" or "subsets." Common choices include k-fold cross-validation and stratified k-fold cross-validation. In k-fold cross-validation, the data is divided into k equal-sized folds. In stratified k-fold, the division is done while preserving the class distribution, which is useful for classification tasks.

2. **Model Training and Evaluation**: The model is trained and evaluated k times, with each of the k subsets used as the test set exactly once while the remaining k-1 subsets are used as the training set. This results in k sets of evaluation metrics.

3. **Performance Metrics**: Common performance metrics, such as mean squared error (MSE) for regression tasks or accuracy, precision, recall, and F1-score for classification tasks, are computed for each fold.

4. **Performance Aggregation**: The performance metrics from each fold are aggregated to compute a single performance metric that represents the model's overall performance. For example, in k-fold cross-validation, you might calculate the mean or median of the k performance metrics to obtain a more robust estimate.

5. **Hyperparameter Tuning**: Cross-validation is often used for hyperparameter tuning, where different combinations of hyperparameters are evaluated across multiple folds to find the best set of hyperparameters.

Cross Validation

Benefits of Cross-Validation:

- Provides a more robust estimate of a model's performance by using multiple splits of the data.
- Helps detect issues like overfitting or underfitting that may not be apparent with a single train-test split.
- Utilizes the entire dataset for both training and testing, maximizing data utilization.

Common Variations of Cross-Validation:

- **K-Fold Cross-Validation**: The dataset is divided into k equally sized subsets, and each fold is used as the test set exactly once.
- **Stratified K-Fold Cross-Validation**: Like k-fold, but it preserves the class distribution in each fold, ensuring that each class is represented proportionally.
- **Leave-One-Out Cross-Validation (LOOCV)**: Each data point serves as a separate test set, while the remaining data points are used for training. LOOCV is useful for very small datasets but can be computationally expensive.
- **Time Series Cross-Validation**: Designed for time series data, it ensures that training data comes before test data to mimic real-world scenarios where future data is unknown.

Cross-validation is a crucial tool for model assessment and selection, and it helps ensure that your machine learning models generalize well to unseen data.

K-fold cross-validation

In this technique, the whole dataset is partitioned in k parts of equal size and each partition is called a fold. It's known as k-fold since there are k parts where k can be any integer - 3,4,5, etc.

One fold is used for validation and other K-1 folds are used for training the model. To use every fold as a validation set and other left-outs as a training set, this technique is repeated k times until each fold is used once.

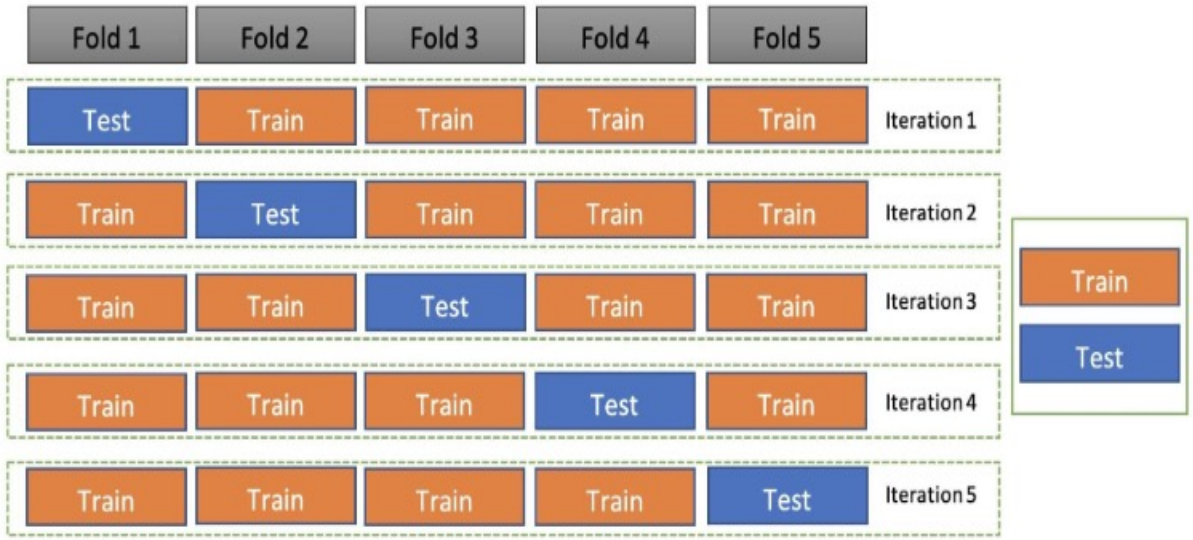


Image source: sqlrelease.com

The image above shows 5 folds and hence, 5 iterations. In each iteration, one fold is the test set/validation set and the other k-1 sets (4 sets) are the train set. To get the final accuracy, you need to take the accuracy of the k-models validation data.

This validation technique is not considered suitable for imbalanced datasets as the model will not get trained properly owing to the proper ratio of each class's data.

Here's an example of how to perform k-fold cross-validation using Python.

Code:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import cross_val_score,KFold
from sklearn.linear_model import LogisticRegression
iris=load_iris()
X=iris.data
Y=iris.target
lr=LogisticRegression()
k_fold=KFold(n_splits=7)
score=cross_val_score(lr,X,Y,cv=k_fold)
print("Cross Validation Scores:{}".format(score))
print("Average Cross Validation score:{}".format(score.mean()))
```

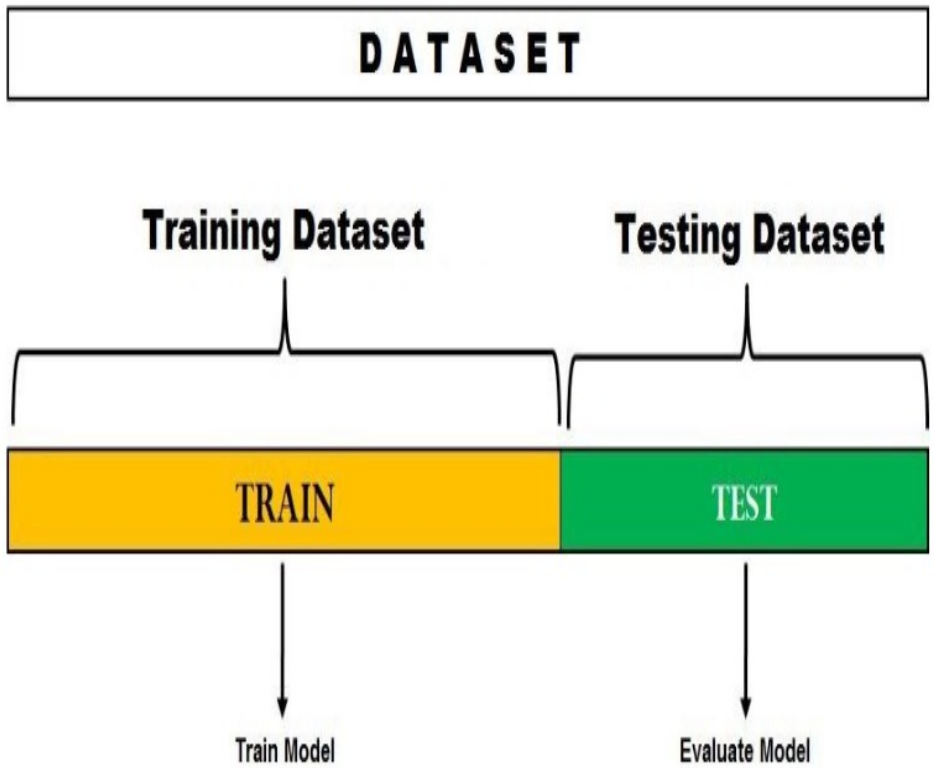
Image Source : Author

Output:

```
Cross Validation Scores:[1.         1.         1.         0.80952381
0.95238095 0.85714286
 0.9047619 ]
Average Cross Validation score:0.9319727891156463
```

Holdout cross-validation

Also called a train-test split, holdout cross-validation has the entire dataset partitioned randomly into a training set and a validation set. A rule of thumb to partition data is that nearly 70% of the whole dataset will be used as a training set and the remaining 30% will be used as a validation set. Since the dataset is split into only two sets, the model is built just one time on the training set and executed faster.



Code:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
iris=load_iris()
X=iris.data
Y=iris.target
lr=LogisticRegression()
x_train,x_test,y_train,y_test=train_test_split(X,Y,test_size=0.25,ran
dom_state=42)
lr.fit(x_train,y_train)
results=lr.predict(x_test)
print("Training
accuracy:{}".format(accuracy_score(lr.predict(x_train),y_train)))
print("Testing accuracy:{}".format(accuracy_score(results,y_test)))
```

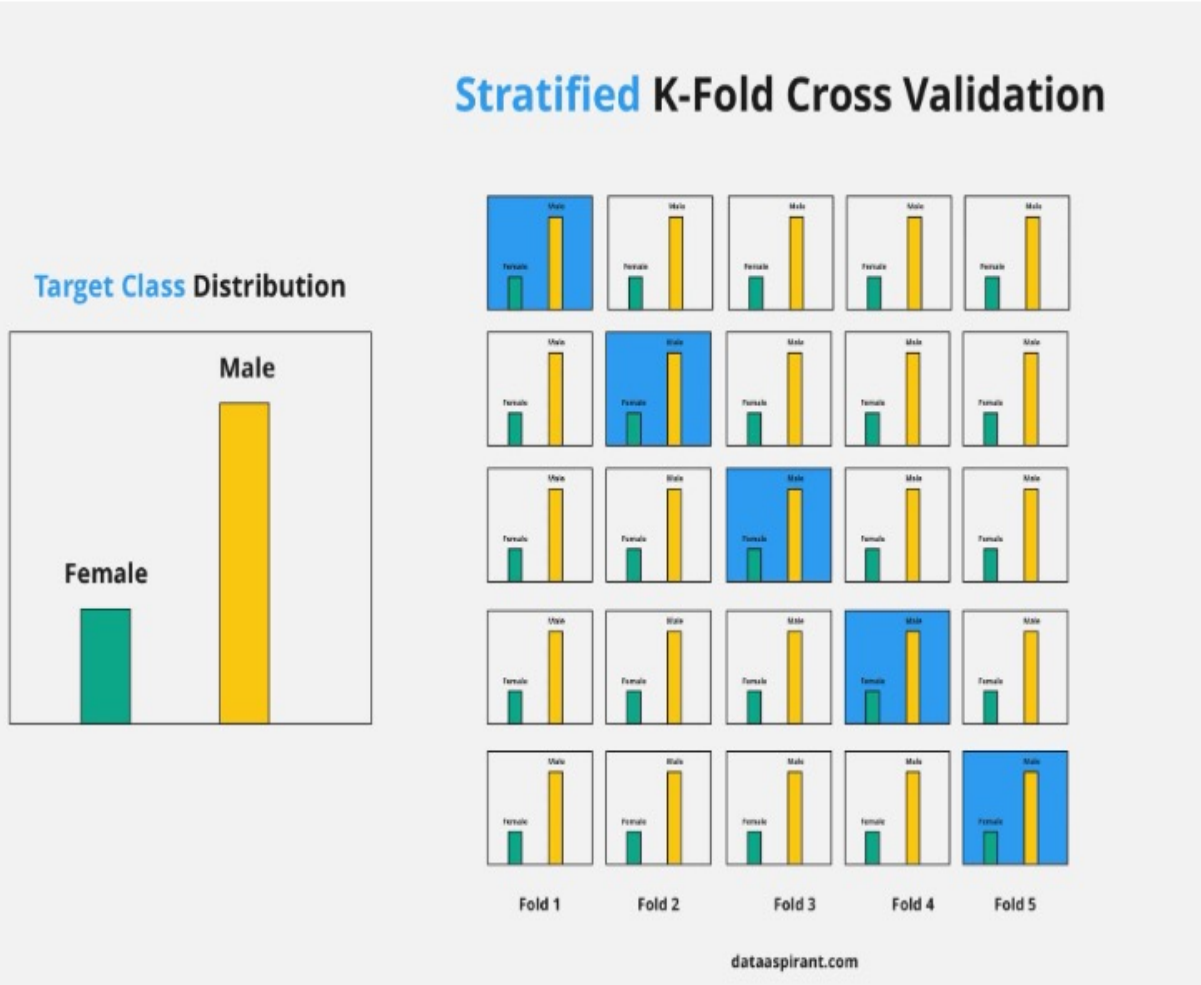
Image Source : Author

Output:

```
Training accuracy:0.9642857142857143
Testing accuracy:1.0
```

Stratified k-fold cross-validation

As seen above, k-fold validation can't be used for imbalanced datasets because data is split into k-folds with a uniform probability distribution. Not so with stratified k-fold, which is an enhanced version of the k-fold cross-validation technique. Although it too splits the dataset into k equal folds, each fold has the same ratio of instances of target variables that are in the complete dataset. This enables it to work perfectly for imbalanced datasets, but not for time-series data.



In the example above, the original dataset contains females that are a lot less than males, so this target variable distribution is imbalanced. In the stratified k-fold cross-validation technique, this ratio of instances of the target variable is maintained in all the folds.

Code:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import cross_val_score,StratifiedKFold
from sklearn.linear_model import LogisticRegression
iris=load_iris()
X=iris.data
Y=iris.target
lr=LogisticRegression()
st_kf=StratifiedKFold(n_splits=3)
score=cross_val_score(lr,X,Y,cv=st_kf)
print("Cross Validation Scores:{}".format(score))
print("Average Cross Validation score:{}".format(score.mean()))
```

Image Source : Author

Output:

```
Cross Validation Scores:[0.98 0.96 0.98]
Average Cross Validation score:0.9733333333333333
```

Leave-p-out cross-validation

An exhaustive cross-validation technique, p samples are used as the validation set and n-p samples are used as the training set if a dataset has n samples. The process is repeated until the entire dataset containing n samples gets divided on the validation set of p samples and the training set of n-p samples. This continues till all samples are used as a validation set.

The technique, which has a high computation time, produces good results. However, it's not considered ideal for an imbalanced dataset and is deemed to be a computationally unfeasible method. This is because if the training set has all samples of one class, the model will not be able to properly generalize and will become biased to either of the classes.

Code:

```
from sklearn.model_selection import LeavePOut,cross_val_score
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
iris=load_iris()
X=iris.data
Y=iris.target
lpo=LeavePOut(p=1)
lpo.get_n_splits(X)
tree=RandomForestClassifier(n_estimators=5,max_depth=3,n_jobs=-1)
score=cross_val_score(tree,X,Y,cv=lpo)
print("Cross Validation Scores-{}".format(score))
print("Average Cross Validation score :{}".format(score.mean()))
```

Leave-one-out cross-validation

In this technique, only 1 sample point is used as a validation set and the remaining n-1 samples are used in the training set. Think of it as a more specific case of the leave-p-out cross-validatio technique with P=1.

To understand this better, consider this example:
There are 1000 instances in your dataset. In each iteration, 1 instance will be used for the validation set and the remaining 999 instances will be used as the training set. The process repeats itself until every instance from the dataset is used as a validation sample.

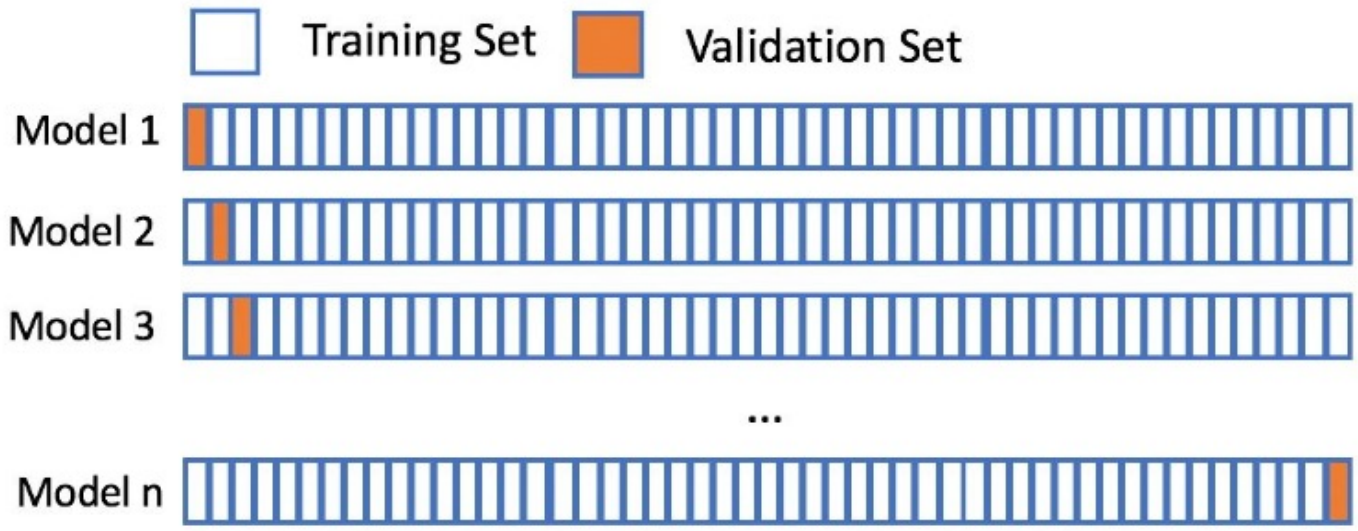


Image source

The leave-one-out cross-validation method is computationally expensive to perform and shouldn't be used with very large datasets. The good news is that the technique is very simple and requires no configuration to specify. It also provides a reliable and unbiased estimate for your model performance.

Code:

```
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import LeaveOneOut,cross_val_score
iris=load_iris()
X=iris.data
Y=iris.target
leave_one_out=LeaveOneOut()
rfc=RandomForestClassifier(n_estimators=7,max_depth=4,n_jobs=-1)
score=cross_val_score(rfc,X,Y,cv=leave_one_out)
print("Cross Validation Scores:{}".format(score))
print("Average Cross Validation score:{}".format(score.mean()))
```

Time series (rolling cross-validation / forward chaining method)

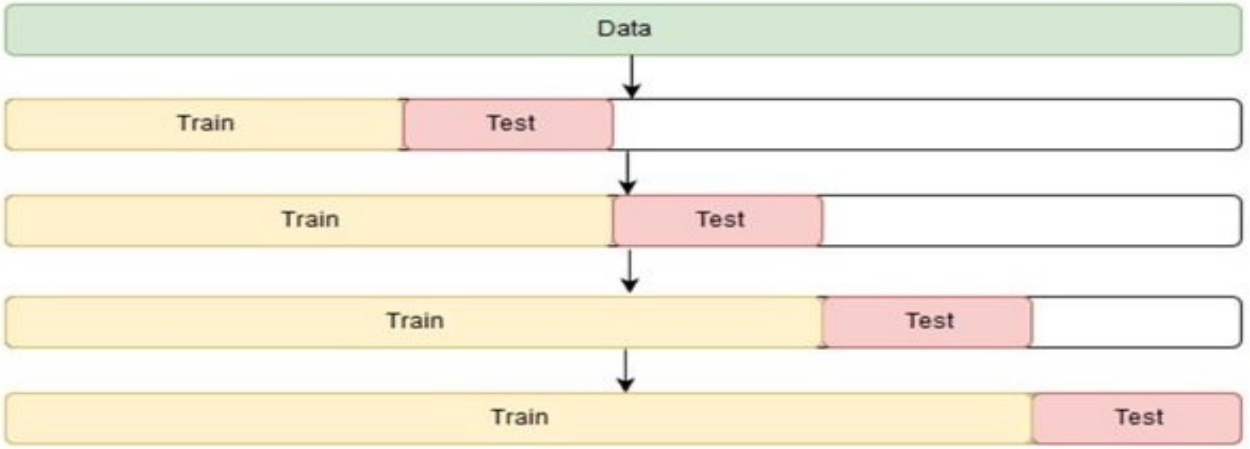
Before going into the details of the rolling cross-validation technique, it's important to understand what time-series data is.

Time series is the type of data collected at different points in time. This kind of data allows one to understand what factors influence certain variables from period to period. Some examples of time series data are weather records, economic indicators, etc.

In the case of time series datasets, the cross-validation is not that trivial. You can't choose data instances randomly and assign them the test set or the train set. Hence, this technique is used to perform cross-validation on time series data with time as the important factor.

Since the order of data is very important for time series-related problems, the dataset is split into training and validation sets according to time. Therefore, it's also called the forward chaining method or rolling cross-validation.

To begin:
Start the training with a small subset of data. Perform forecasting for the later data points and check their accuracy. The forecasted data points are then included as part of the next training dataset and the next data points are forecasted. The process goes on.



Code:

```
import numpy as np
from sklearn.model_selection import TimeSeriesSplit
X = np.array([[1, 2], [3, 4], [1, 2], [3, 4], [1, 2], [3, 4], [1,1],
[2,2]])
y = np.array([1, 2, 3, 4, 5, 6, 7, 8])
tscv = TimeSeriesSplit(n_splits=7)
print(tscv)
i=1
for train_index, test_index in tscv.split(X):
   print(f"TRAIN {i}:", train_index, "TEST:", test_index)
   X_train, X_test = X[train_index], X[test_index]
   y_train, y_test = y[train_index], y[test_index]
   i+=1
```

Image Source : Author

Output:

```
TimeSeriesSplit(gap=0, max_train_size=None, n_splits=7,
test_size=None)
TRAIN 1: [0] TEST: [1]
TRAIN 2: [0 1] TEST: [2]
TRAIN 3: [0 1 2] TEST: [3]
TRAIN 4: [0 1 2 3] TEST: [4]
TRAIN 5: [0 1 2 3 4] TEST: [5]
TRAIN 6: [0 1 2 3 4 5] TEST: [6]
TRAIN 7: [0 1 2 3 4 5 6] TEST: [7]
```

Classification

Classification: a Machine Learning technique to identify the category of new observations based on training data.

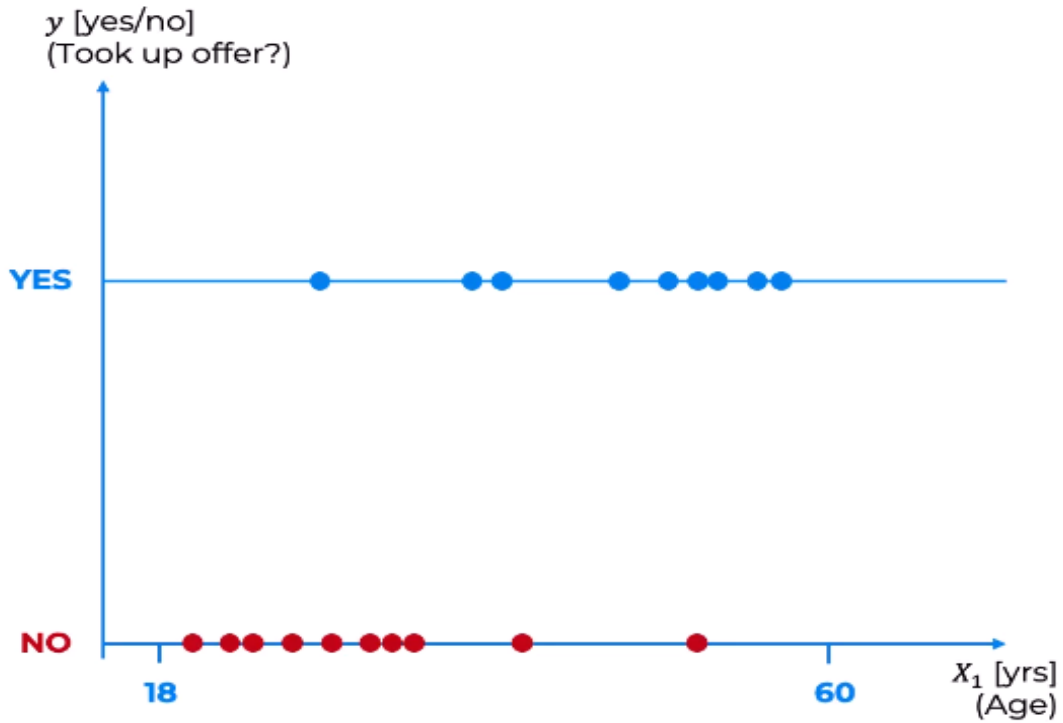
Logistic Regression

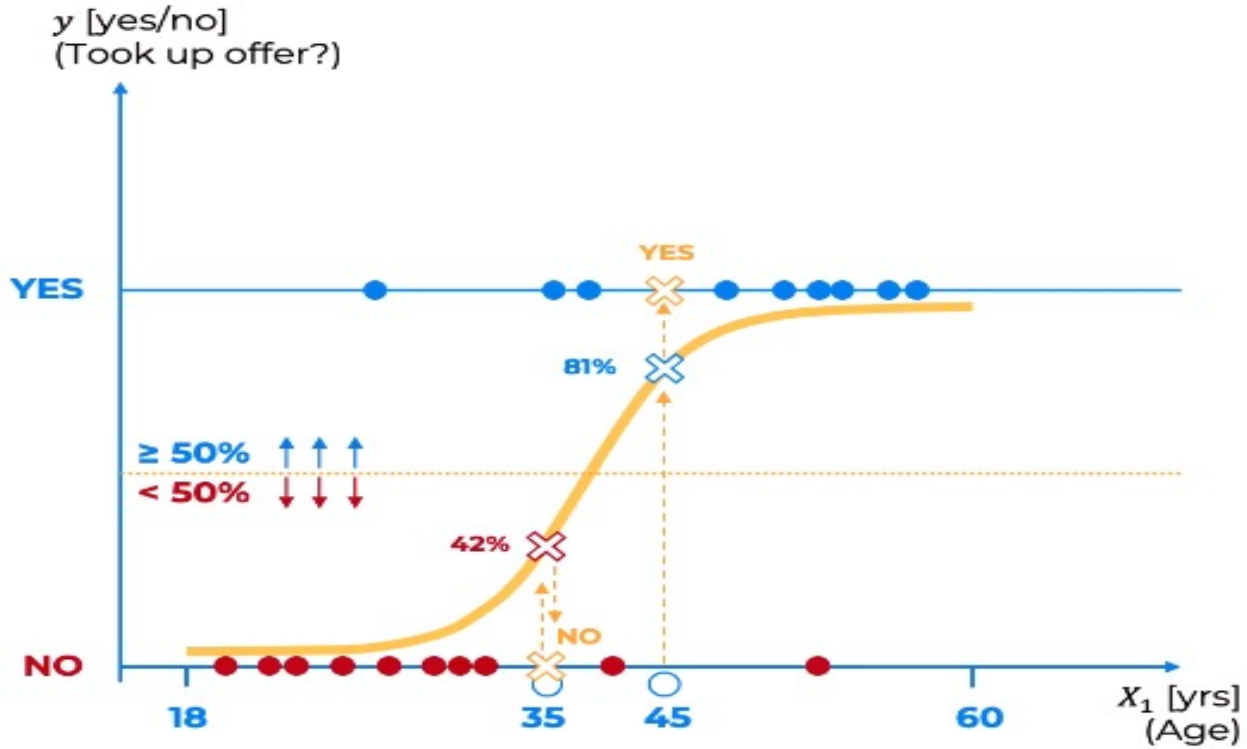
Logistic regression: predict a categorical dependent variable from a number of independent variables.

Will purchase health insurance: Yes / No ~ Age + Income + Level of Education + Family or Single

Will purchase health insurance: Yes / No ~ Age

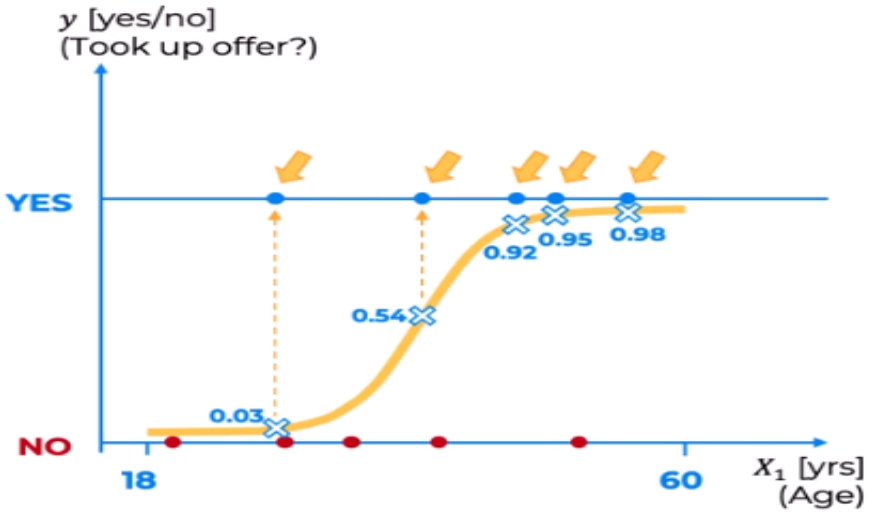
$$\ln \frac{p}{1-p} = b_0 + b_1 X_1$$

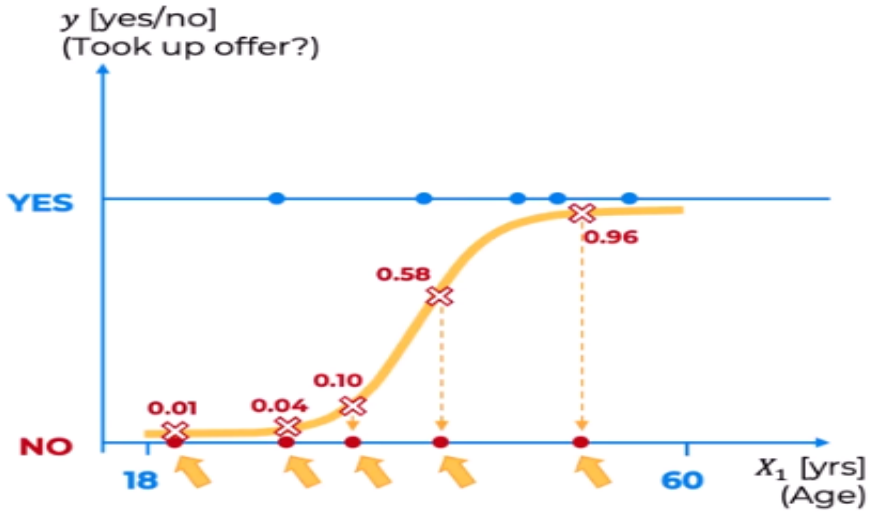




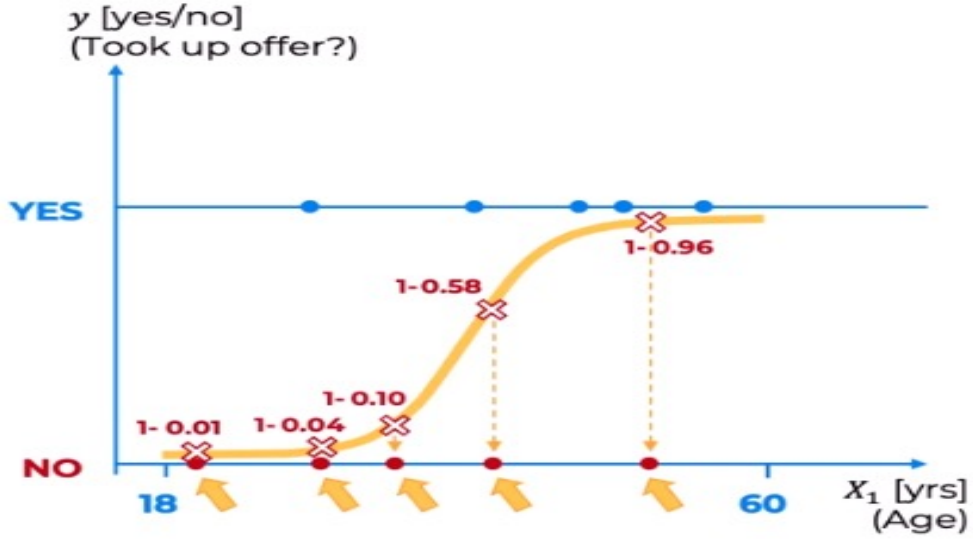
Maximum Likelihood

Probability of saying yes

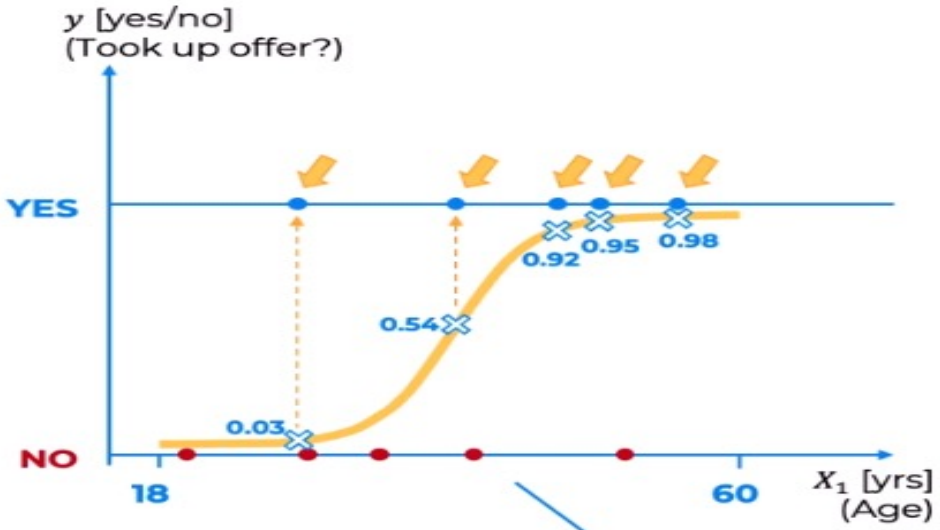


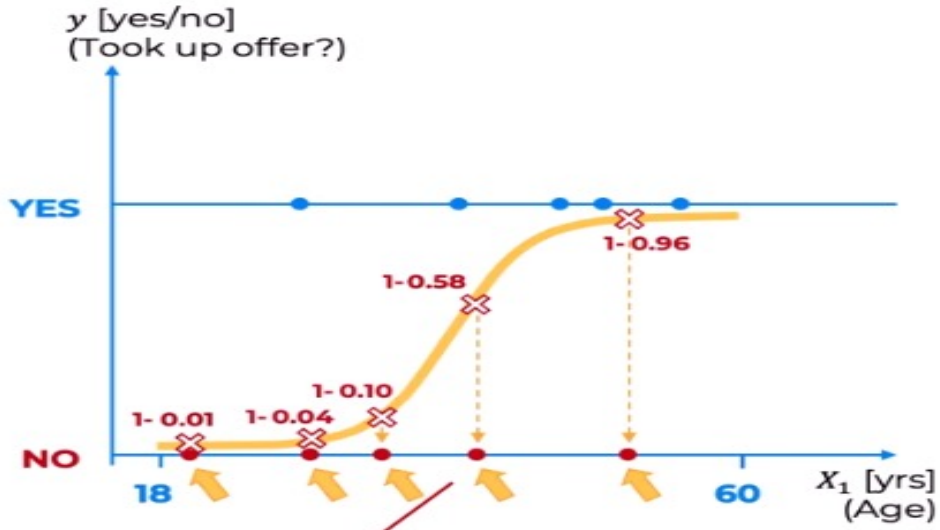


Probability of saying No



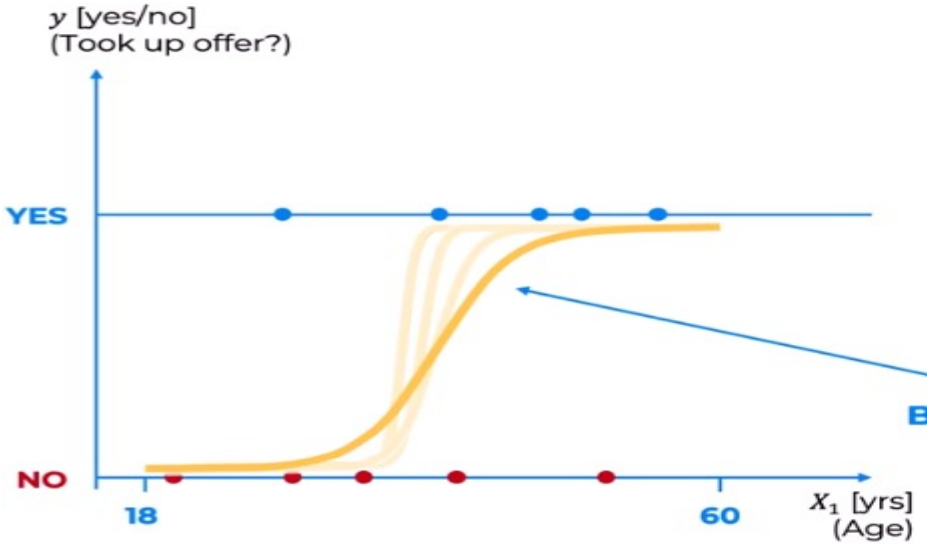
Maximum Likelihood





Likelihood = 0.03 x 0.54 x 0.92 x 0.95 x 0.98 x (1 – 0.01) x (1 – 0.04) x (1 – 0.10) x (1 – 0.58) x (1 – 0.96)

Likelihood = **0.00019939**



Likelihood = 0.00007418

Likelihood = 0.00012845

Likelihood = 0.00016553

Likelihood = 0.00019939

Best Curve <= Maximum Likelihood

Odds Ratio

The odds ratio (OR) is a statistical measure used to quantify the strength and direction of the association between two binary outcomes. It is commonly used in epidemiology, medical research, and social sciences to assess the likelihood of an event occurring in one group compared to another.

The formula for calculating the odds ratio is:

$$OR = \frac{(\text{Odds of the event in group 1})}{(\text{Odds of the event in group 2})}$$

In this formula:

- Group 1 typically represents the group with the outcome of interest or exposure to a certain factor.

- Group 2 represents the reference or comparison group.

The odds of an event occurring in each group are calculated as:

- Odds of the event in group 1 = $\frac{\text{Number of events in group 1}}{\text{Number of non-events in group 1}}$
- Odds of the event in group 2 = $\frac{\text{Number of events in group 2}}{\text{Number of non-events in group 2}}$

The odds ratio can take on values greater than 1, less than 1, or equal to 1, and its interpretation depends on the value:

1. OR = 1: The odds of the event are the same in both groups, indicating no association or difference.

2. OR > 1: Group 1 has higher odds of the event than Group 2, suggesting a positive association or increased risk.

3. OR < 1: Group 2 has higher odds of the event than Group 1, indicating a negative association or decreased risk.

The odds ratio is used to assess the strength and direction of the relationship between two variables and is especially useful when studying the effects of exposures, interventions, or risk factors on outcomes. It is commonly reported in research studies along with confidence intervals to provide a range of values within which the true odds ratio is likely to fall.

Sigmoid Function

ƒ(x) = p(y=1) = e^x / (e^x + 1), is the formula for the logistic function, also known as the sigmoid function.

This function is commonly used in logistic regression and machine learning for binary classification problems.

Here's a breakdown of the components:

- ƒ(x): This represents the logistic function, and it is a function of a real-valued input variable x.
- p(y=1): This represents the probability that a binary outcome variable y equals 1.
- e: This is the base of the natural logarithm, approximately equal to 2.71828.

The logistic function takes a real-valued input x and transforms it into a probability value between 0 and 1. It does so by applying the exponential function (e^x) to x, which makes the output positive, and then dividing it by the sum of that exponential value and 1. This ensures that the output is a valid probability, as probabilities must be between 0 and 1.

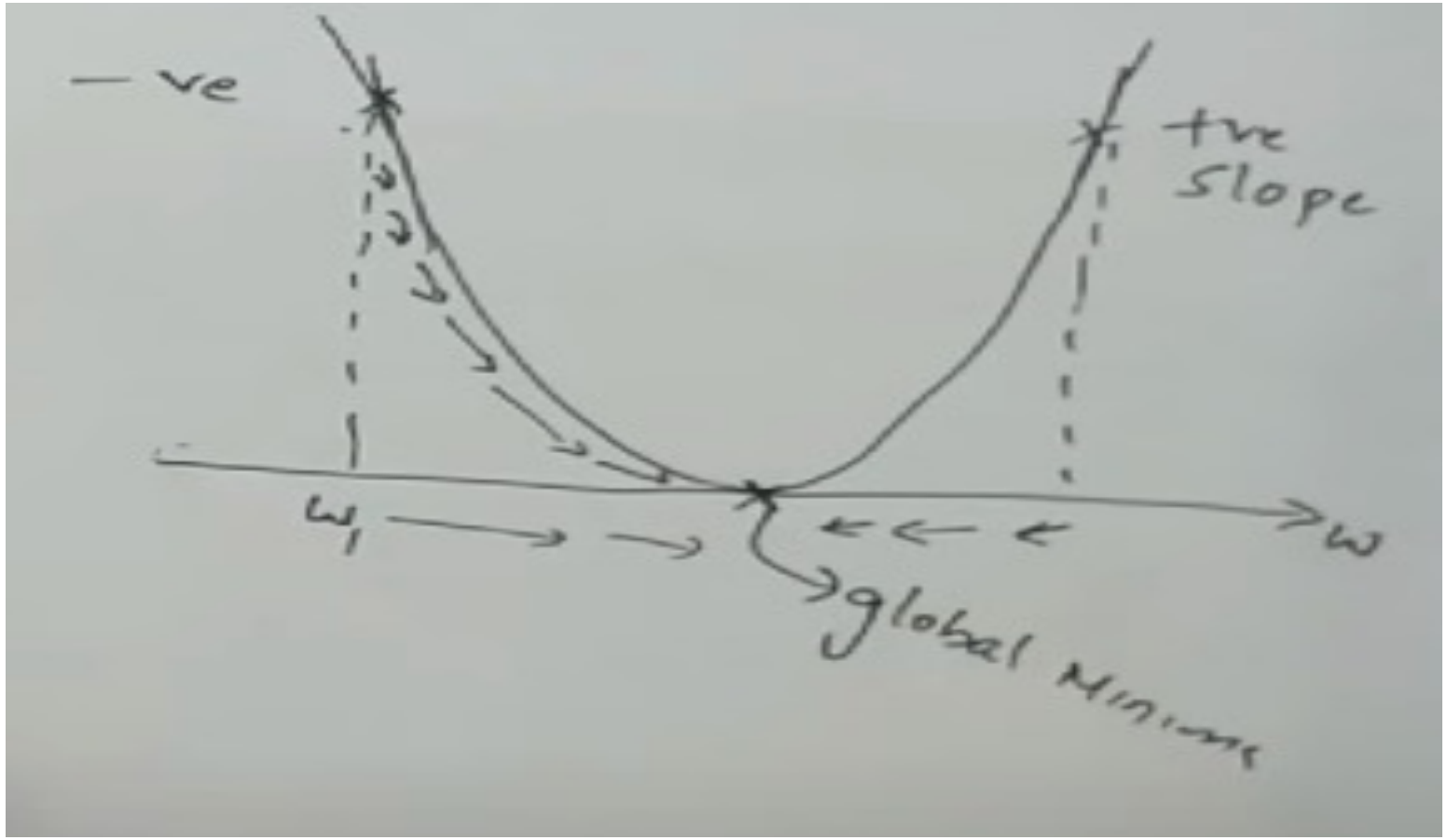
When x is positive, ƒ(x) approaches 1, indicating a high probability of y=1. When x is negative, ƒ(x) approaches 0, indicating a low probability of y=1. When x is zero, ƒ(x) equals 0.5, indicating that y=1 and y=0 are equally likely.

The logistic function is the core component of logistic regression models, where it models the relationship between the input variables and the probability of a binary outcome. It's a useful tool for problems where you want to estimate the probability of an event happening based on one or more predictor variables.

Gradient Descent

Weight Updation Formula- New weight is equal to old weight minus learning rate multiple by derivative(slope) of loss divided by derivative of old weight

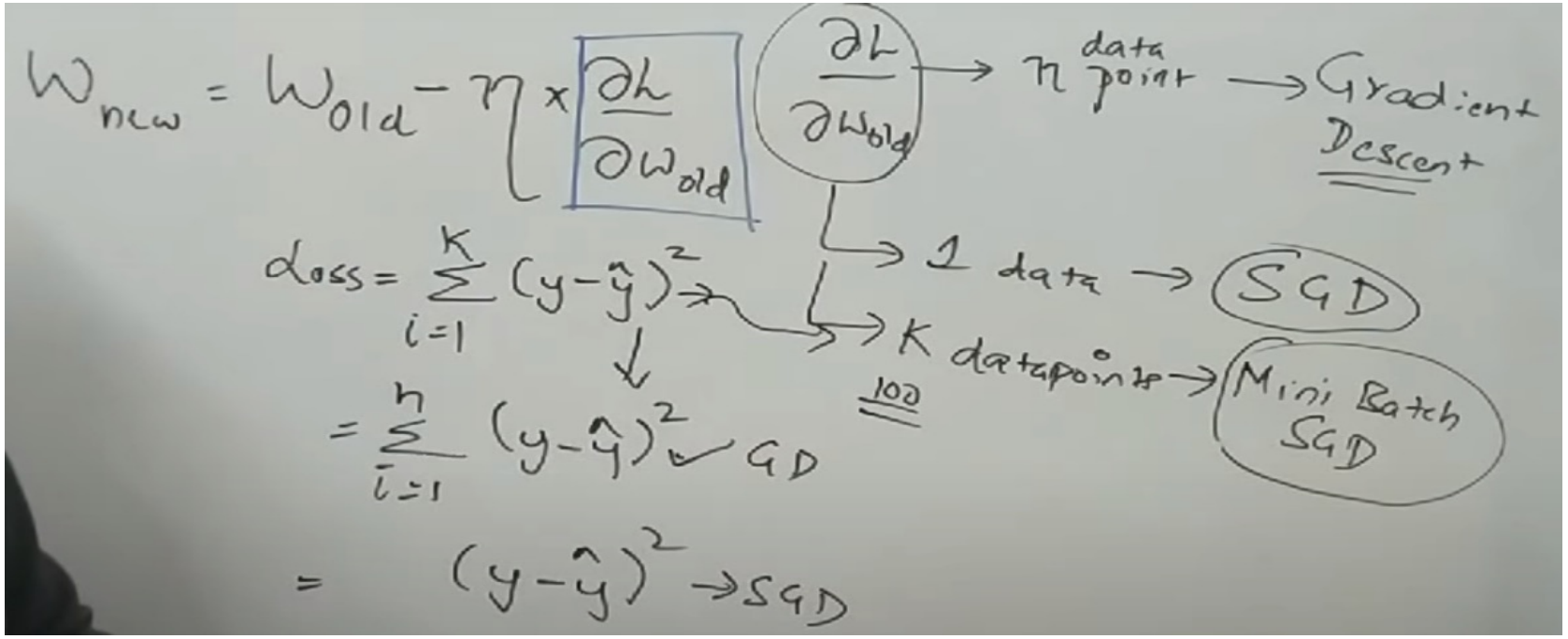
$$W_{new} = W_{old} - \eta \times \frac{\partial h}{\partial W_{old}}$$

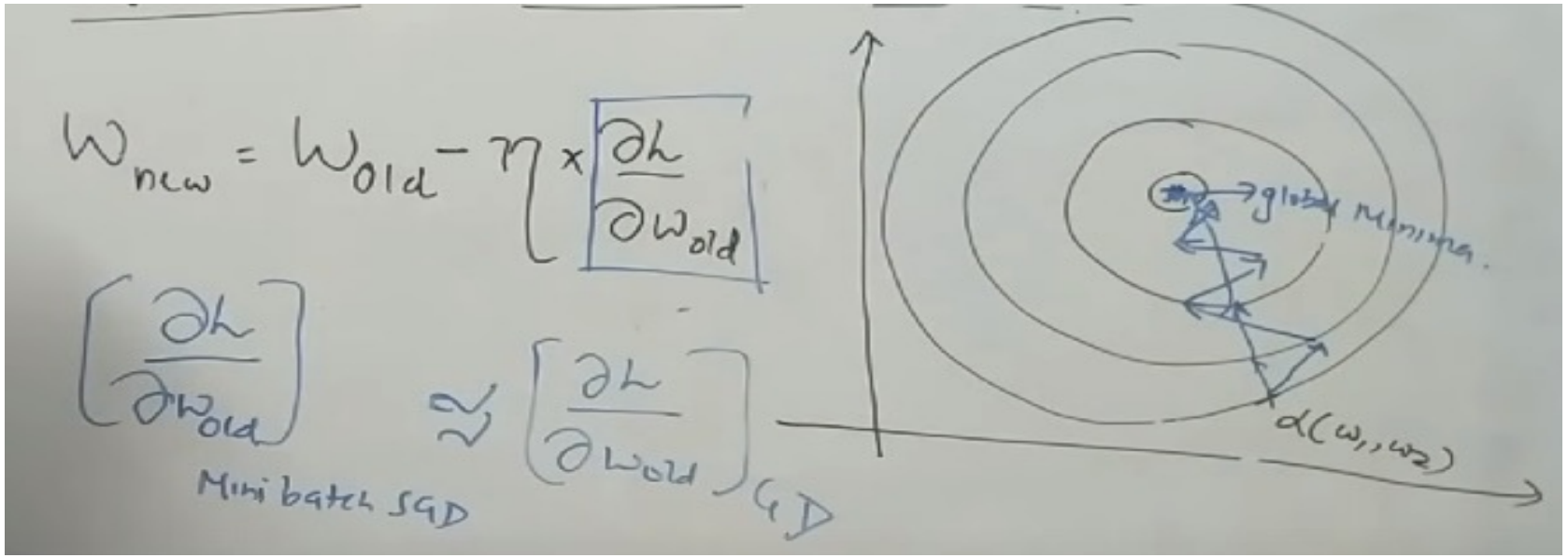


- In traditional gradient descent, the goal is to minimize a cost or loss function J(θ), where θ represents the model parameters (weights and biases).

- Gradient descent iteratively updates θ by taking steps in the direction of the negative gradient of the cost function with respect to θ. This means moving in the direction that reduces the cost.

Gradient Descent and Stochastic Gradient Descent





Gradient Descent and Stochastic Gradient Descent

Advantages:

- SGD is computationally less expensive than batch gradient descent since it processes one training example at a time.
- It can handle large datasets because it doesn't require storing the entire dataset in memory.
- The stochastic nature of the updates can help escape local minima and explore the parameter space more effectively.

Challenges:

- The stochastic updates introduce randomness and can lead to noisy convergence, making the optimization process less stable.
- The learning rate (α) needs to be carefully tuned. A fixed learning rate or a learning rate schedule is often used.
- Since it processes data in random order, convergence may be slower than batch gradient descent.

Confusion Matrix

		Predicted	
		Negative **(N)** **-**	Positive **(P)** **+**
Actual	Negative **-**	True Negative **(TN)**	**False Positive (FP)** **Type I Error**
	Positive **+**	**False Negative (FN)** **Type II Error**	True Positive **(TP)**

Key Metrics

Accuracy, precision, recall, and AUC-ROC are common evaluation metrics used in binary classification tasks to assess the performance of machine learning models. Each metric provides a different perspective on the model's performance, and they are often used together to gain a comprehensive understanding of how well a model is doing.

1. **Accuracy**:
   - Accuracy is a straightforward metric that measures the proportion of correctly classified instances (both true positives and true negatives) out of the total number of instances.
   - Formula: $\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$
   - Accuracy provides an overall assessment of a model's correctness but may not be suitable for imbalanced datasets, where one class dominates.
2. **Precision**:
   - Precision (also known as positive predictive value) measures the proportion of true positive predictions among all positive predictions made by the model.
   - Formula: $\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$
   - Precision is valuable when the cost of false positives is high. It tells us how well the model avoids making incorrect positive predictions.

Key Metrics

3. **Recall**:
   - Recall (also known as sensitivity or true positive rate) measures the proportion of true positive predictions among all actual positive instances.
   - Formula: $\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$
   - Recall is valuable when the cost of false negatives is high. It tells us how well the model captures all positive instances.
4. **AUC-ROC (Area Under the Receiver Operating Characteristic Curve)**:
   - AUC-ROC is a metric that evaluates the ability of a binary classification model to distinguish between positive and negative classes across various classification thresholds.
   - It quantifies the area under the ROC curve, where the ROC curve plots the true positive rate (Recall) against the false positive rate at different threshold settings.
   - AUC-ROC ranges from 0 to 1, with higher values indicating better model performance.
   - AUC-ROC is useful for assessing a model's overall discriminatory power and is less affected by class imbalance.

Key Metrics

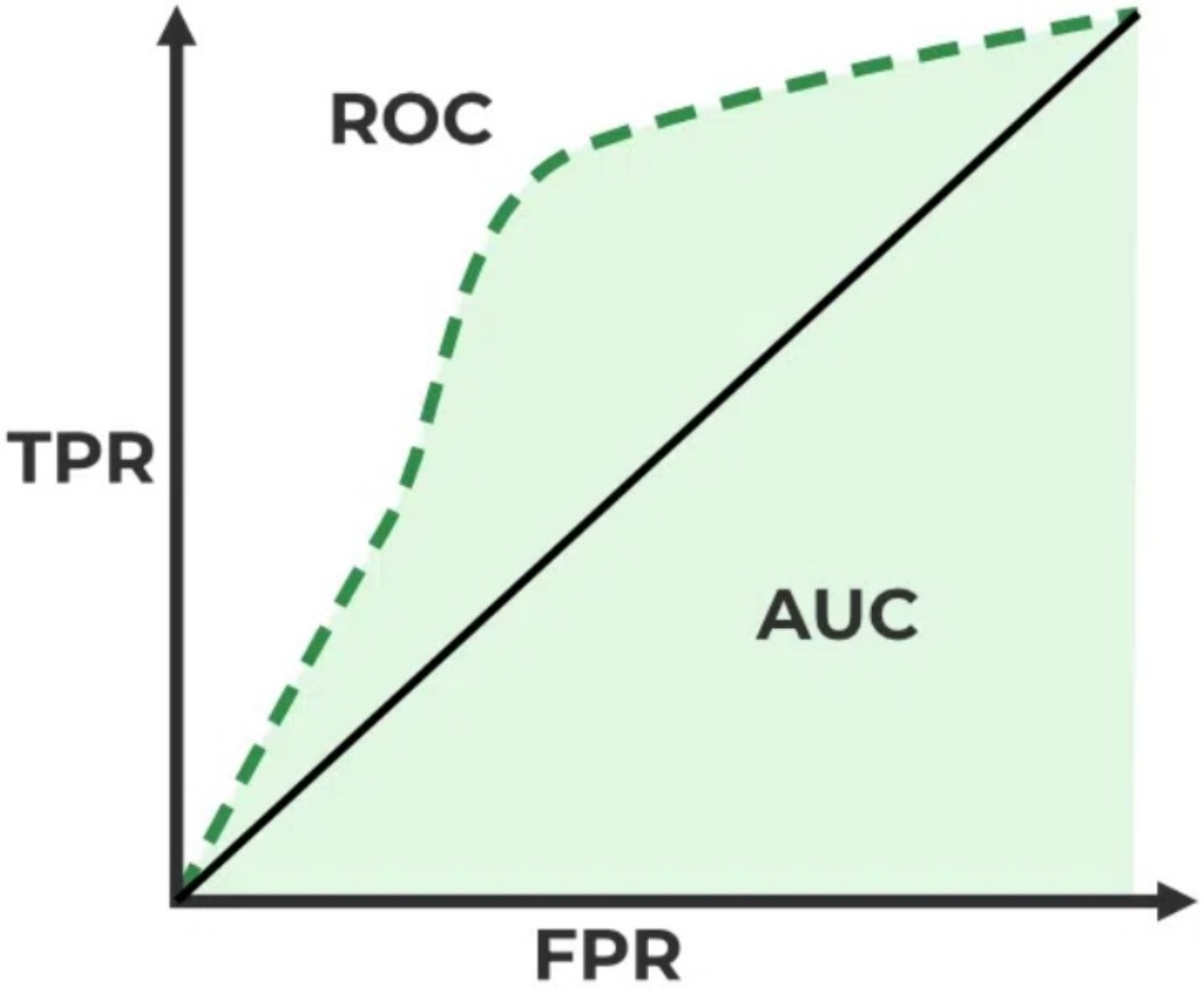
Specificity:

- Specificity, also known as the True Negative Rate or TN Rate, measures the proportion of true negatives (correctly predicted negatives) out of all actual negatives.
- It quantifies how well the model can correctly identify instances of the negative class.
- Formula: $\text{Specificity} = \frac{\text{True Negatives}}{\text{True Negatives} + \text{False Positives}}$
- Specificity ranges from 0 to 1, with higher values indicating better performance in correctly identifying negatives.

Error Rate:

- The error rate (or misclassification rate) represents the proportion of misclassified instances, i.e., instances that were classified incorrectly.
- Formula: $\text{Error Rate} = \frac{\text{False Positives} + \text{False Negatives}}{\text{Total Number of Instances}}$
- Error rate provides an overall measure of how often the model's predictions are incorrect.

ROC AUC Curve



ROC Curve

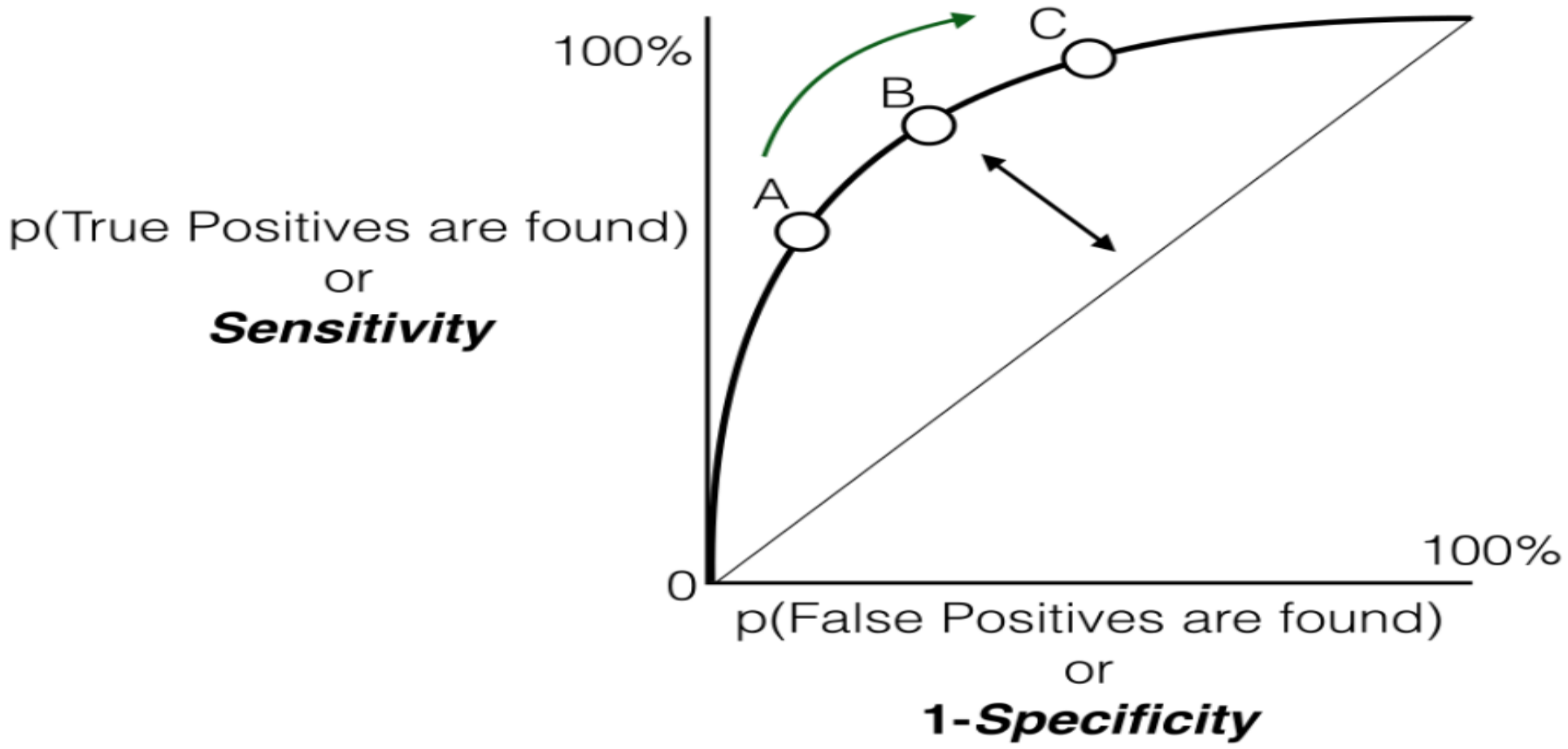
ROC stands for Receiver Operating Characteristics, and the ROC curve is the graphical representation of the effectiveness of the binary classification model. It plots the true positive rate (TPR) vs the false positive rate (FPR) at different classification thresholds.

AUC Curve:

AUC stands for Area Under the Curve, and the AUC curve represents the area under the ROC curve. It measures the overall performance of the binary classification model. As both TPR and FPR range between 0 to 1, So, the area will always lie between 0 and 1, and A greater value of AUC denotes better model performance. Our main goal is to maximize this area in order to have the highest TPR and lowest FPR at the given threshold. The AUC measures the probability that the model will assign a randomly chosen positive instance a higher predicted probability compared to a randomly chosen negative instance. ***AUC measures how well a model is able to distinguish between classes.***

ROC AUC Curve

And as said earlier ROC is nothing but the plot between TPR and FPR across all possible thresholds and AUC is the entire area beneath this ROC curve.



SGD Regressor-Key parameters

Loss:
Specifies the loss function to be used for optimization. Common options include:

- ``'squared_loss'`` (default): Ordinary least squares (OLS) regression, which minimizes the mean squared error.
- ``'huber'``: Huber loss, which is a combination of squared loss and absolute loss, providing robustness to outliers.
- ``'epsilon_insensitive'``: Linear Support Vector Regression (SVR) loss.
- ``'squared_epsilon_insensitive'``: Squared SVR loss.

Penalty:
Determines the type of regularization to apply to the model to prevent overfitting. Common options include:

- ``'none'`` (default): No regularization.
- ``'l2'``: L2 regularization (ridge regression).
- ``'l1'``: L1 regularization (lasso regression).
- ``'elasticnet'``: Elastic Net regularization, which is a combination of L1 and L2 regularization.

Alpha:
Controls the strength of regularization. Higher values result in stronger regularization. It's equivalent to the inverse of the regularization strength (1/lambda).

Learning_rate:
Determines the learning rate schedule for updating the model's weights during optimization. Common options include:

- ``'constant'`` (default): Use a constant learning rate defined by the `eta0` parameter.
- ``'optimal'``: Use an optimal learning rate calculated based on the data and the current iteration number.
- ``'invscaling'``: Use an inverse scaling learning rate schedule.
- ``'adaptive'``: Use an adaptive learning rate schedule.

eta0:
The initial learning rate when `learning_rate` is set to ``'constant'``.

Power_t:
The exponent for inverse scaling learning rate when `learning_rate` is set to ``'invscaling'``.

epsilon_insensitive:
This parameter specifies the margin of tolerance with respect to the true output value within which no penalty is associated with the errors. It is used when the loss parameter is set to 'epsilon_insensitive' or 'squared_epsilon_insensitive'.

You can set it to control how large or small errors are ignored during training. Smaller values of epsilon_insensitive make the model less sensitive to errors within that margin.

If let’s say , epsilon_insensitive is set to 0.1, which means that errors within a margin of 0.1 units around the true output values are ignored during training when using the epsilon-insensitive loss function.

Passive Aggressive Model(Extra Information for learning)

Passive-Aggressive (PA) is a type of machine learning model, particularly used for classification tasks. It is called "Passive-Aggressive" because of its approach to updating model parameters in response to the observed data. PA models are especially useful in online learning scenarios where data arrives sequentially, and the model needs to adapt quickly to changing patterns. Here's an overview of the Passive-Aggressive machine learning model:

Objective: Passive-Aggressive models are primarily used for binary and multiclass classification tasks, although there are variants for regression as well.

Working Principle:
1. Initialization: The model's parameters (weights and bias) are initialized to some initial values.

2. For Each Data Point:
   - Prediction: The current model is used to make a prediction on the current data point.
   - Loss Computation: The loss or error between the predicted label and the true label is calculated.
   - Update Rule: Depending on whether the prediction is correct or incorrect, the model updates its parameters using an update rule. This update rule is where the "Passive" and "Aggressive" components come into play.

3. Update Rule:
   - If the prediction is correct (i.e., the model's prediction matches the true label), the model doesn't make a significant parameter change; it is passive.
   - If the prediction is incorrect, the model makes an aggressive update to correct its mistake. The size of this update is controlled by a hyperparameter called "C."

The Hyperparameter C: The aggressiveness of the updates is controlled by the hyperparameter C. A smaller C value makes the updates more passive, allowing the model to learn more slowly, while a larger C value makes the updates more aggressive, allowing the model to adapt quickly but potentially overfitting to noisy data.

Variants of Passive-Aggressive Models:
- Passive-Aggressive (PA): The original PA algorithm.
- Passive-Aggressive for Regression (PA-R): Used for regression tasks.
- Passive-Aggressive for Large-Scale Learning (PA-I and PA-II): Variants optimized for large-scale datasets.

Applications:
Passive-Aggressive models are particularly useful in scenarios where data is continuously arriving, and the model needs to update itself with each new data point. Common applications include text classification, spam detection, and online advertising.

```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import PassiveAggressiveClassifier
from sklearn.metrics import accuracy_score

# Load a sample dataset (Iris dataset for classification)
data = load_iris()
X = data.data
y = data.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create a Passive-Aggressive Classifier
clf = PassiveAggressiveClassifier(C=1.0, max_iter=1000, random_state=42)

# Train the classifier on the training data
clf.fit(X_train, y_train)

# Make predictions on the test data
y_pred = clf.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

Cost Function of Logistic Regression

The loss function of logistic regression is known as the log loss or cross-entropy loss. It is used to quantify the error or the discrepancy between the predicted probabilities produced by the logistic regression model and the actual binary outcomes in the training data.

Mathematically, the log loss for logistic regression is defined as:

$L(y, p) = -[y * \log(p) + (1 - y) * \log(1 - p)]$

Where:

- L(y, p) is the log loss.
- y is the actual binary outcome (0 or 1).
- p is the predicted probability that the sample belongs to class 1 (i.e., the probability of success).

The log loss penalizes the model more severely when it makes inaccurate predictions with high confidence. It is a commonly used loss function for binary classification problems in logistic regression because it is differentiable and well-suited for gradient-based optimization algorithms.

The goal in logistic regression is to minimize the log loss by adjusting the model's parameters (coefficients) during the training process, typically using techniques like gradient descent. This process helps the model learn to produce accurate probabilities for classifying observations into one of the two classes.

Deviance

Deviance is a number that measures the goodness of fit of a logistic regression model. Think of it as the distance from the perfect fit — a measure of how much your logistic regression model deviates from an ideal model that perfectly fits the data.

Deviance ranges from 0 to infinity. The smaller the number the better the model fits the sample data (deviance = 0 means that the logistic regression model describes the data perfectly). Higher values of the deviance correspond to a less accurate model.

What is a good value for the deviance?
In general, the lower the deviance the better but there is no threshold for an acceptable value. The deviance is not meant to be interpreted on its own, instead you can use it to compare your logistic regression model to either:
A reference model
Another model that includes either a larger or smaller subset of predictors

1. Comparison with a reference model

- You can compare the deviance of your logistic regression model with that of a reference/null model with no independent variables (including only the intercept).
- This reference model is the one that predicts the average of the outcome Y for all observations.
- Since your model includes some predictors, we would expect it to fit the data better than the reference model i.e. to have a lower deviance.
- The size of the difference between the deviance of your model and that of the reference reflects how important the independent variables are in your model.

2. Comparison with a model of different size

- You can also compare the deviance of your model with that of another one that includes, for example, 1 additional independent variable.
- The goal is to get a sense of the importance of this additional predictor.
- The lower the deviance of the larger model is (compared to the smaller model), the more important this variable is.

```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import log_loss

# Load a sample dataset (Iris dataset for binary classification)
data = load_iris()
X = data.data
y = (data.target == 0).astype(int)  # Convert to binary classification (setosa vs. non-setosa)

# Split the data into a training and testing set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Create and fit a logistic regression model
model = LogisticRegression()
model.fit(X_train, y_train)

# Predict probabilities for the test set
y_pred_prob = model.predict_proba(X_test)[:, 1]  # Probability of class 1 (setosa)

# Calculate the log loss (deviance) for the model's predictions
deviance = log_loss(y_test, y_pred_prob)

print(f"Deviance (Log Loss): {deviance:.4f}")
```

Null Deviance: This represents the deviance when you have a model with only the intercept term (no predictor variables). In other words, it's the deviance of the simplest possible model where you don't use any information from your predictors to make predictions.

Residual Deviance: This represents the deviance of your fitted logistic regression model, which includes predictor variables. It's the deviance after accounting for the effects of your predictor variables.

The difference between the null deviance and the residual deviance measures how much better your model is at explaining the variation in the response variable compared to the null model. Specifically, a lower residual deviance indicates that your model is doing a better job of fitting the data, as it's able to explain more of the variance.

Psuedo R-Square

The pseudo R-squared, also known as a **model fit statistic**, is a measure of how well a statistical model, particularly in the context of regression analysis, explains the variation in the dependent variable. It is called "pseudo" because it's not a true R-squared like what you might find in linear regression, but it serves a similar purpose in logistic regression and other models where the likelihood function is used to assess fit.

There are several different pseudo R-squared statistics used in logistic regression, and each has its strengths and weaknesses. Here are a few common ones:

1. **McFadden's R-squared (McFadden's pseudo R-squared)**:
   - Formula: 1 - (Log-Likelihood of Fitted Model / Log-Likelihood of Null Model)
   - McFadden's R-squared compares the log-likelihood of your fitted model to the log-likelihood of a null model with no predictors. It's a measure of how much better your model is at explaining the variation in the response variable compared to a null model. Values range from 0 (poor fit) to 1 (perfect fit).
2. **Cox and Snell R-squared**:
   - Formula: 1 - (Likelihood of Null Model / Likelihood of Fitted Model)^(2/n)
   - Cox and Snell R-squared also compares the likelihood of the null model to that of the fitted model but takes the sample size (n) into account. It is an improvement over McFadden's R-squared for larger sample sizes.
3. **Nagelkerke R-squared**:
   - Formula: (1 - (Likelihood of Null Model / Likelihood of Fitted Model)) / (1 - (Likelihood of Null Model^(2/n)))
   - Nagelkerke's R-squared is a modification of Cox and Snell R-squared that attempts to make the values closer to the true R-squared values observed in linear regression. It is scaled between 0 and 1.

Noise in Data

In the context of machine learning and data analysis, "noise" refers to random or irrelevant information or variability in data that can obscure the underlying patterns and make it more challenging for machine learning algorithms to make accurate predictions or classifications. Noise can originate from various sources and can negatively impact the performance of a machine learning model. Here are some common sources and types of noise in data:

1. **Random Variation**: Random noise is inherent in many real-world data collection processes. It can manifest as small, unpredictable fluctuations in data points due to measurement errors or natural variability.

2. **Outliers**: Outliers are data points that significantly deviate from the majority of the data. Outliers can introduce noise because they may not represent typical or meaningful patterns and can lead the model to make incorrect generalizations.

3. **Measurement Errors**: Errors during data collection or recording can introduce noise. For example, sensor inaccuracies, data entry mistakes, or equipment malfunctions can all contribute to measurement errors.

4. **Missing Data**: Incomplete or missing data points can be a source of noise. Handling missing data is a critical preprocessing step, and how you impute or remove missing values can affect the quality of the dataset.

5. **Class Imbalance**: In classification tasks, class imbalance can lead to noise. When one class significantly outnumbers another, the model may struggle to learn the minority class patterns effectively, leading to inaccurate predictions for that class.

6. **Labeling Errors**: In supervised learning, if the labels or ground truth are incorrect or noisy, it can negatively impact model training and evaluation.

7. **Data Perturbations**: Data perturbations refer to deliberate modifications made to a dataset for various purposes, including privacy protection, data augmentation, or model robustness testing. These modifications typically involve altering the values or characteristics of data points while preserving the overall statistical properties of the dataset.

Dealing with noise in data is an essential part of the data preprocessing and feature engineering stages in machine learning. Strategies to address noise include:

1. **Data Cleaning**: Identifying and handling outliers, errors, and missing data through data cleaning techniques.

2. **Feature Engineering**: Creating relevant features and removing irrelevant ones to reduce noise and improve the model's ability to capture meaningful patterns.

3. **Data Smoothing**: Applying techniques like moving averages or kernel smoothing to reduce the impact of random noise.

4. **Resampling**: Balancing class distributions in imbalanced datasets to mitigate the noise caused by class imbalance.

5. **Regularization**: Using regularization techniques in machine learning models to prevent overfitting, which can be exacerbated by noisy data.

6**. Cross-Validation**: Employing cross-validation to assess model performance on different subsets of data and identify potential issues caused by noise.

Overall, noise in data is a common challenge in machine learning, and addressing it effectively is crucial for building robust and accurate models.

Recursive Feature Elimination-Feature Selection Method

Recursive Feature Elimination (RFE) is a feature selection technique used in machine learning to automatically select the most important features or variables from a given dataset. It is especially useful when dealing with high-dimensional datasets, where the number of features is large and you want to reduce the dimensionality to improve model performance, reduce overfitting, and speed up training.

Here's how RFE works:

1**. Initial Model:** RFE starts by training a machine learning model (typically a classifier or regressor) on the entire dataset using all available features. The choice of the initial model can vary depending on the problem you are solving.

2**. Feature Ranking:** After training the initial model, RFE assigns a weight or importance score to each feature based on how much it contributed to the model's performance. Features that have the least impact on the model's performance are considered less important.

3. **Feature Elimination:** The least important feature(s) are then removed from the dataset. The number of features to remove at each iteration is a user-defined parameter. It can be one feature at a time or more, depending on the problem.

4. **Model Re-training**: The model is re-trained on the reduced feature set (i.e., the dataset with the least important features removed).

5. **Repeat**: Steps 2 to 4 are repeated iteratively until a predefined stopping criterion is met. This criterion can be a fixed number of features to select or a specific performance threshold.

6. **Final Model**: Once the stopping criterion is reached, the final selected features are those that remain in the dataset. These are considered the most important features according to the RFE process.

In scikit-learn, the `RFE` (Recursive Feature Elimination) method is used for feature selection by recursively removing less important features from the dataset. It takes several parameters to customize its behavior. Here's an explanation of the most important parameters:

1. **estimator**: This parameter specifies the base estimator or model that will be used for feature ranking and elimination. You need to pass a machine learning estimator object that has a `coef_` or `feature_importances_` attribute. Common choices include classifiers like `LogisticRegression`, `RandomForestClassifier`, `SVM`, or regressors like `LinearRegression`, `RandomForestRegressor`, etc.

2. **n_features_to_select**: It allows you to specify the number of features you want to select. By default, it selects half of the features if the total number of features is more than 50, or all of them if there are 50 or fewer features. If you set this parameter, it will select the specified number of features.

3. **step**: This parameter determines how many features are removed at each iteration. The default is 1, which means one feature is eliminated at a time. You can increase this value if you want to remove multiple features in each iteration.

4. **verbose**: Controls the verbosity of the output. If set to 0 (the default), there is no output during the fitting process. If set to 1, it prints messages about the progress of the feature elimination.

5. **importance_getter**: This parameter is a callable that takes an estimator fitted on the dataset and returns feature importances or coefficients. If not specified, the default behavior is to use the `coef_` attribute for classifiers and `feature_importances_` for regressors.

6. **cv**: The cross-validation strategy used for scoring the features at each step of elimination. You can specify an integer (e.g., 5 for 5-fold cross-validation) or a cross-validation object. Cross-validation helps estimate the performance of the model with the selected subset of features.

7. **scoring**: The scoring metric used to evaluate the performance of the model during feature selection. It should be a string corresponding to a valid scoring metric for your specific problem (e.g., 'accuracy' for classification or 'r2' for regression).

8. **fit_params**: Dictionary of additional parameters to pass to the `fit` method of the estimator.

RFECV Parameters

The `RFECV` (Recursive Feature Elimination with Cross-Validation) class in scikit-learn combines the functionality of RFE (Recursive Feature Elimination) with cross-validation to perform feature selection while estimating the model's performance. Here are the key parameters of the `RFECV` class:

1. **estimator**: This parameter specifies the base estimator or model that will be used for feature ranking and elimination. Like in `RFE`, you need to pass a machine learning estimator object that has a `coef_` or `feature_importances_` attribute. Common choices include classifiers like `LogisticRegression`, `RandomForestClassifier`, `SVM`, or regressors like `LinearRegression`, `RandomForestRegressor`, etc.

2. **step**: This parameter determines how many features are removed at each iteration. The default is 1, which means one feature is eliminated at a time. You can increase this value if you want to remove multiple features in each iteration.

3. **cv**: The cross-validation strategy used for scoring the features at each step of elimination. You can specify an integer (e.g., 5 for 5-fold cross-validation) or a cross-validation object. Cross-validation helps estimate the performance of the model with the selected subset of features.

4. **scoring**: The scoring metric used to evaluate the performance of the model during feature selection. It should be a string corresponding to a valid scoring metric for your specific problem (e.g., 'accuracy' for classification or 'r2' for regression).

5. **min_features_to_select**: Specifies the minimum number of features to select. It can be an integer or None (the default). If None, it starts with all features and reduces them until only `n_features_to_select` features remain.

6. **verbose**: Controls the verbosity of the output during the fitting process. Set to 0 (default) for no output, 1 for progress messages, and 2 for more detailed debugging information.

7. **n_jobs**: The number of CPU cores to use for parallelization. If -1 (the default), all available CPU cores are used.

8. **importance_getter**: This parameter is a callable that takes an estimator fitted on the dataset and returns feature importances or coefficients. If not specified, the default behavior is to use the `coef_` attribute for classifiers and `feature_importances_` for regressors.

9. **step:** Specifies the number of features to remove at each iteration. The default is 1, which means one feature is eliminated at a time. Increasing this value removes more features in each step.

10. **verbose**: Controls the verbosity of the output during the fitting process. Set to 0 (default) for no output, 1 for progress messages, and 2 for more detailed debugging information.

11. **fit_params**: Dictionary of additional parameters to pass to the `fit` method of the estimator.

SelectKBest-Feature Selection Method

The **SelectKBest** method in scikit-learn is a feature selection technique used for selecting the top k most important features from a dataset. This method is particularly useful when you want to reduce the dimensionality of your dataset by selecting only the most informative features, which can lead to improved model performance and reduced overfitting.

```python
from sklearn.datasets import load_iris
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2

# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Create a SelectKBest instance and select the top 2 features
selector = SelectKBest(score_func=chi2, k=2)
X_new = selector.fit_transform(X, y)

# Get the selected feature indices and scores
selected_feature_indices = selector.get_support(indices=True)
feature_scores = selector.scores_

print("Selected Feature Indices:", selected_feature_indices)
print("Feature Scores:", feature_scores)
print("X_new (Top 2 Features):\n", X_new)
```

Some of the commonly used scoring functions in `SelectKBest`, along with explanations of each:

1. **f_classif (ANOVA F-statistic for classification)**:
   - Suitable for classification problems.
   - Measures the difference in means between classes normalized by the variance within each class.
   - Higher values indicate more significant differences between classes for a particular feature.
2. **f_regression (F-statistic for regression):**
   - Suitable for regression problems.
   - Measures the linear relationship between each feature and the target variable.
   - Higher values indicate a stronger linear relationship with the target.
3. **chi2 (Chi-squared statistic):**
   - Suitable for classification problems with categorical (discrete) features and a categorical target.
   - Measures the dependence between each feature and the target variable by comparing the observed and expected frequencies in contingency tables.
   - Higher values indicate greater dependence between the feature and the target.
4. **mutual_info_classif (Mutual information for classification):**
   - Suitable for classification problems.
   - Measures the mutual information between each feature and the target, which quantifies the amount of information that a feature provides about the target.
   - Higher values indicate a higher level of information shared between the feature and the target.
5. **mutual_info_regression (Mutual information for regression):**
   - Suitable for regression problems.
   - Measures the mutual information between each feature and the target in a regression context.
   - Higher values indicate a higher level of information shared between the feature and the target.
6. **SelectPercentile:**
   - This is not a specific scoring function but rather a method that selects the top percentile of features based on their highest scores using any of the above scoring functions.
   - You specify the percentile as a parameter when creating the `SelectKBest` instance.
7. **SelectFpr (False Positive Rate):**
   - Selects features based on a user-defined false positive rate threshold.
   - It's often used for feature selection in the context of statistical hypothesis testing.

The choice of scoring function depends on the nature of your machine learning problem (classification or regression), the type of data (categorical or continuous), and your specific goals. You can experiment with different scoring functions to determine which one works best for your dataset and problem.

Variance Threshold Method-Feature Selection

The **Variance Threshold method** is a simple feature selection technique used to remove features (columns) from a dataset that have low variance. Features with low variance are those that have almost constant values across all samples or nearly identical values, which means they don't provide much information for modeling and can often be safely removed. This method is particularly useful when dealing with datasets where some features have little to no variability.

Here's how the Variance Threshold method works:

1. Calculate the variance for each feature in the dataset. Variance is a measure of how much the values of a feature vary from the mean. Features with low variance have values that are close to each other, while features with high variance have values that are more spread out.
2. Set a threshold value. This threshold is a user-defined value that determines the minimum amount of variance a feature must have to be considered relevant. Features with a variance below this threshold are candidates for removal.
3. Remove features with variance below the threshold. Any feature with variance lower than the specified threshold is removed from the dataset.

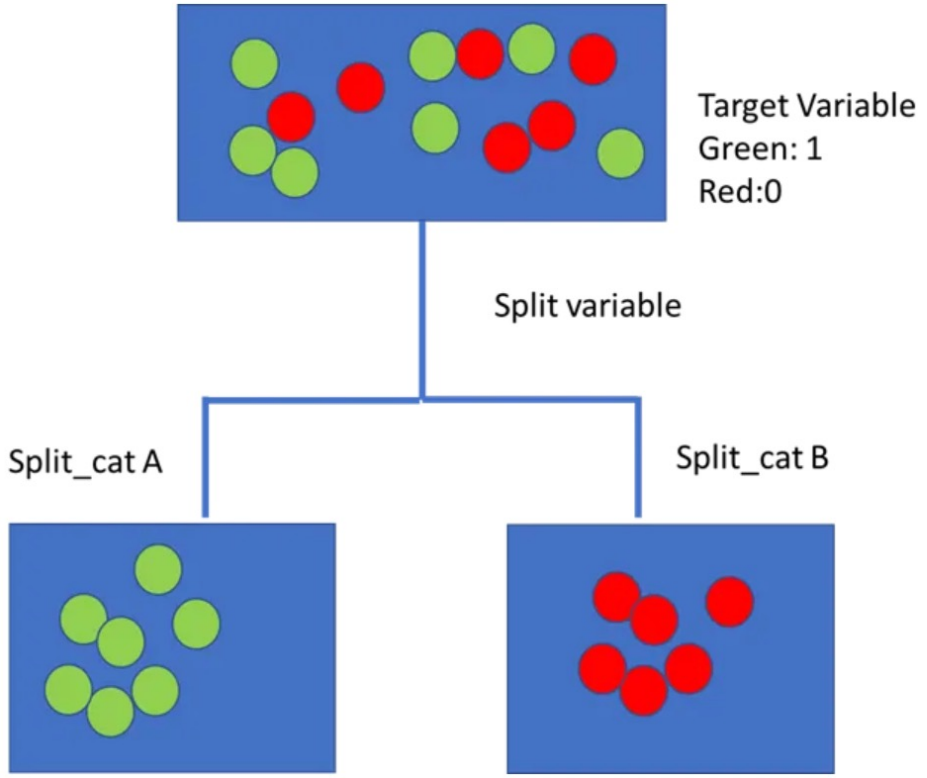
```
from sklearn.feature_selection import VarianceThreshold

# Create a VarianceThreshold instance with a specified threshold (e.g., 0.1)
selector = VarianceThreshold(threshold=0.1)

# Fit the selector to your feature matrix and transform the data
X_new = selector.fit_transform(X)

# Get the selected features
selected_feature_indices = selector.get_support(indices=True)
```

In this example:

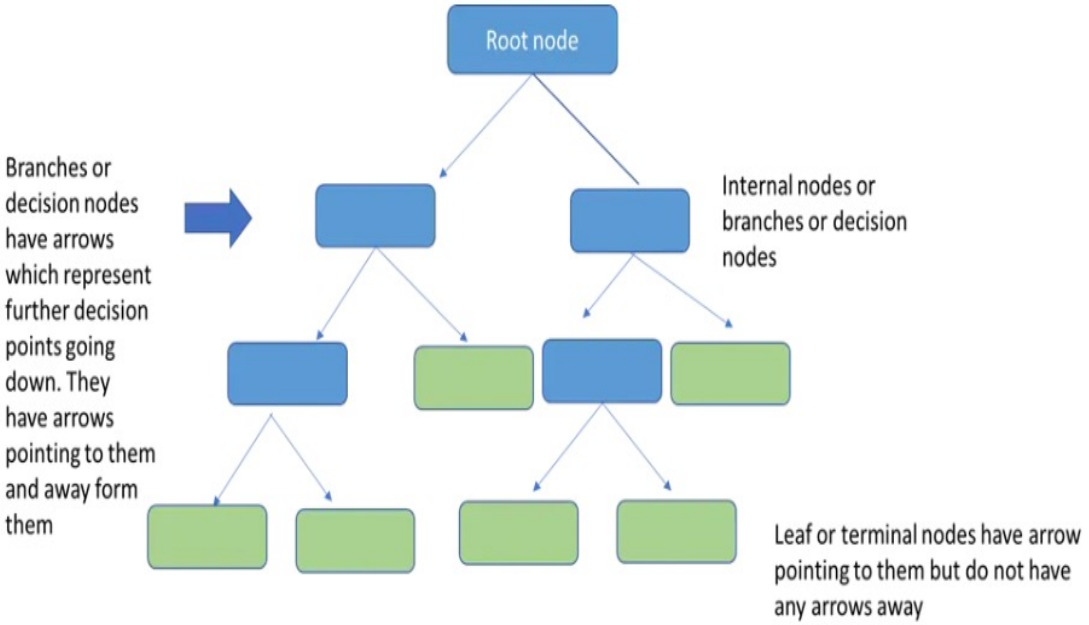
- 'X' is your original feature matrix.
- 'threshold' is set to 0.1, meaning any feature with a variance less than 0.1 will be removed.
- 'X_new' will contain the selected features after applying the Variance Threshold method.
- 'selected_feature_indices' will give you the indices of the selected features if you want to identify which features were kept.

Decision Tree

The first thing to understand in Decision Trees is that they split the predictor space, i.e., the target variable into different sub groups which are relatively more homogenous from the perspective of the target variable. For example, if the target variable is binary, with categories 1 and 0 (shown by green and red dots in the image below, then the decision tree works to split the target variable space into sub groups that are more homogenous in terms of having either 1's or 0's.



Understanding components of a Decision Tree



A decision tree is a branching flow diagram or tree chart. It comprises of the following components:

A target variable such as diabetic or not and its initial distribution.

- A root node: this is the node that begins the splitting process by finding the variable that best splits the target variable
- Node purity: Decision nodes are typically impure, or a mixture of both classes of the target variable (0,1 or green and red dots in the image). Pure nodes are those that have one class — hence the term pure. They either have green or red dots only in the image.
- Decision nodes: these are subsequent or intermediate nodes, where the target variable is again split further by other variables
- Leaf nodes or terminal nodes are pure nodes, hence are used for making a prediction of a numerical or class is made.

Problem

We have data for 15 data points of student data on pass or fail an online ML exam. To understand the basic process we begin with a dataset which comprises a target variable that is binary (Pass/Fail) and various binary or categorical predictor variables such as:

- Whether enrolled in other online courses
- Whether student is from a maths, computer science or other background
- Whether working or not working

Notice that only one variable, ‘Student Background’ has more than 2 levels or categories — Maths, CS, Others. It is one for the advantages of Decision Trees compared to other classification models such as Logistic Regression or SVM, that we do not need to carry out one hot encoding to make these into dummy variables.

Resp srl no	Target variable	Predictor variable	Predictor variable	Predictor variable
	Exam Result	Other online courses	Student background	Working Status
1	Pass	Y	Maths	NW
2	Fail	N	Maths	W
3	Fail	y	Maths	W
4	Pass	Y	CS	NW
5	Fail	N	Other	W
6	Fail	Y	Other	W
7	Pass	Y	Maths	NW
8	Pass	Y	CS	NW
9	Pass	n	Maths	W
10	Pass	n	CS	W
11	Pass	y	CS	W
12	Pass	n	Maths	NW
13	Fail	y	Other	W
14	Fail	n	Other	NW
15	Fail	n	Maths	W

Flow of Decision Tree

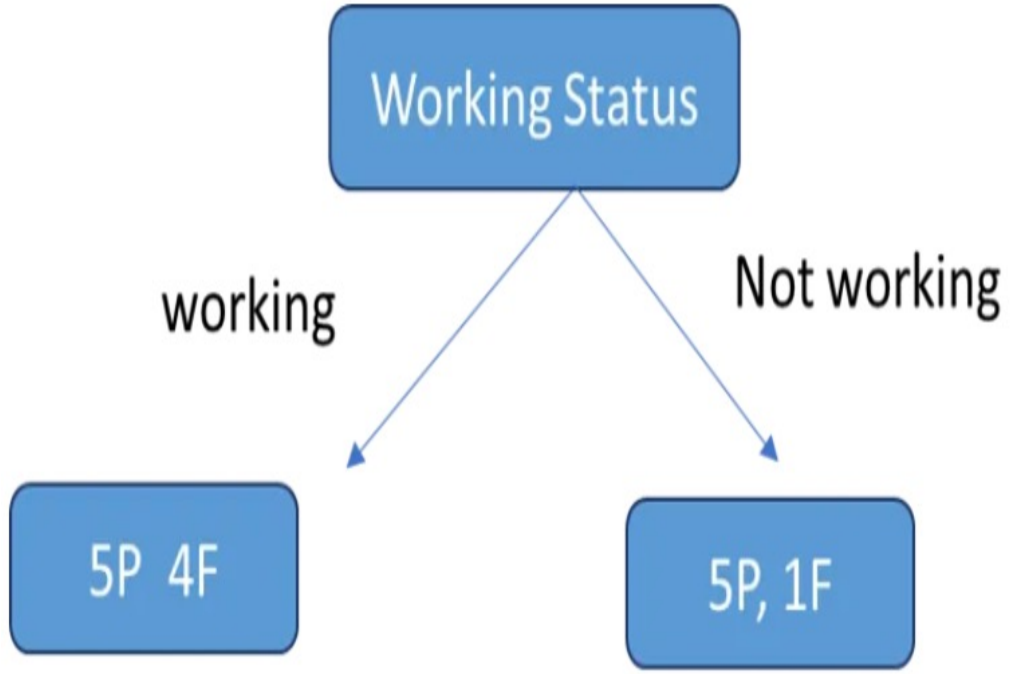
A decision tree begins with the target variable. This is usually called the parent node. The Decision Tree then makes a sequence of splits based in hierarchical order of impact on this target variable. From the analysis perspective the first node is the **root node**, which is the first variable that splits the target variable.

To identify the root node we would evaluate the impact of all the variables that we have currently on the target variable to identify the variable that splits the exam Pass/Fail classes into the most homogenous groups. Our candidates for splitting this are: Background, Working Status and Other Online Courses.

What do we hope to achieve with this split? Suppose we begin with Working Status as the root node. This splits into 2 sub nodes, one each for Working and Not working. Thus the Pass/Fail status is updated in each sub node respectively.

So, this is the basic flow of the Decision Tree. As long as there is a mixture of Pass and Fail in a sub node, there is scope to split further to try and get it to be only one category. This is termed the purity of the node. For example, Not Working has 5 Pass and 1 Fail, hence it is purer than the Working node which has 5P and 4F. A leaf node would be one which contains either Pass or Fail class only.

A node which is impure can be branched further for improving purity. However, most of the time we do not necessarily go down to the point where each leaf is ‘pure’. It is also important to understand that each node is standalone and hence the attribute that best splits the ‘Working’ node may not be the one that best splits the ‘Not Working’ node.



Greedy top down approach

Decision trees follow a top-down, greedy approach that is known as recursive binary splitting. The recursive binary splitting approach is top-down because it begins at the top of the tree and then it successively splits the predictor space. At each split the predictor space gets divided into 2 and is shown via two new branches pointing downwards. The algorithm is termed is greedy because at each step of the process, the best split is made for that step. It does not project forwards and try and pick a split that might be more optimal for the overall tree.

The algorithm therefore evaluates all variables on some statistical criteria and then chooses the variable that performs best on the criteria.

Variable Selection Criteria

Entropy-

Entropy is a term that comes from physics and means a measure of disorder. In the context of Decision Trees, entropy is a measure of disorder or impurity in a node. Thus, a node with more variable composition, such as 2Pass and 2 Fail would be considered to have higher Entropy than a node which has only pass or only fail. The maximum level of entropy or disorder is given by 1 and minimum entropy is given by a value 0.

Leaf nodes which have all instances belonging to 1 class would have an entropy of 0. Whereas, the entropy for a node where the classes are divided equally would be 1.

Entropy is measured by the formula:

$$E = -\sum_{i=1}^{n} p_i \log_2(p_i)$$

Where the pi is the probability of randomly selecting an example in class i. Let us understand this a bit better in the context of our example. So, the initial entropy of at the parent node is given by the probability of getting a pass vs fail. In our dataset, the target variable has 9 passes and 6 fails. Hence the probabilities for the entropy formula are:

P_{pass} = Probability of passing/Total no. of instances = 9/15

P_{fail}= Probability of failing/Total no. of instances = 6/15

Variable Selection Criteria

Now essentially what a Decision Tree does to determine the root node is to calculate the entropy for each variable and its potential splits. For this we have to calculate a potential split from each variable, calculate the average entropy across both or all the nodes and then the change in entropy vis a vis the parent node. This change in entropy is termed Information Gain and represents how much information a feature provides for the target variable.

$$Information\ Gain = Entropy_{parent} - Entropy_{children}$$

Entropy_parent is the entropy of the parent node and Entropy_children represents the average entropy of the child nodes that follow this variable. In the current case since we have 3 variables for which this calculation must be done from the perspective of the split.

1. Work Status

2. Online Course Status

3. Student Background

To calculate entropy, first let us put our formulas for Entropy and Information Gain in terms of variables in our dataset:

1. Probability of pass and fail at each node, i.e, the Pi:

$$P_i = \frac{\#Pass\ or\ Fail}{\#\ subnode}$$

2. Entropy:

$$E = -(P_{pass}\ log_2(P_{pass}) + P_{fail}\ log_2(P_{fail})$$

3. Average Entropy at child nodes:

$$\text{Average Entropy} = \frac{(n_{subnode\ 1})}{n\_parent} E\_subnode1 + \frac{(n_{subnode\ 2})}{n_{parent}} E\_subnode2$$

Note, average Entropy is the weighted average of all the sub nodes that a parent node splits into. Thus, in our example this would be 2 sub nodes for Working Status and 3 sub nodes for Student Background.

4. Information Gain:

$$Information\ Gain = E_{parent} - AvgE_{child}$$

Parent Node Calculations

First we will calculate parent node entropy using the formula above. Use any log2 calculator online to calculate the log values. In our case they work out to:

$$E = -\left[\frac{8}{15}log_2\left(\frac{8}{15}\right) + \frac{7}{15}log_2\left(\frac{7}{15}\right)\right]$$

E=-(.5333* (-0.9069) + .4667* (-1.0995)) = 0.9968

(Mathematical note: log to base 2 of anything less than 1 is a negative number, hence we multiply by a minus sign to get a positive number)

So far, this is just the entropy of the parent node. Now we have to decide which attribute or variable to use to split this to get the root node.

Calculating the Root Node

For this we have to calculate a potential split from each variable, calculate the average entropy across both the nodes and then the change in entropy via a vis the parent node.

Let us begin with Work Status variable and calculate the entropy of the split.

$$Entropy_{working} = -\left[\frac{3}{9} * log_2\left(\frac{3}{9}\right) + \frac{6}{9}log_2\left(\frac{6}{9}\right)\right] = 0.9183$$

$$\text{Entropy\_Notworking} = \left[\frac{5}{6} * log_2(6) + \frac{1}{6}log_2\left(\frac{1}{6}\right)\right] = .6500$$

We then calculate the average entropy for the Working status split as a weighted average with weights of the share of observations from the total number of observations that fall in each sub node.

$$\text{Entropy\_working\_status} = \left[\frac{9}{15} * 0.9183 + \frac{6}{15}0.6500\right] = 0.8110$$

Information Gain = Entropy_Parent — Entropy_child =

0.9183–0.8119 = .1858

In a similar fashion we can evaluate the entropy and information gain for Student Background and Online Courses variables. The results are provided in the table below:

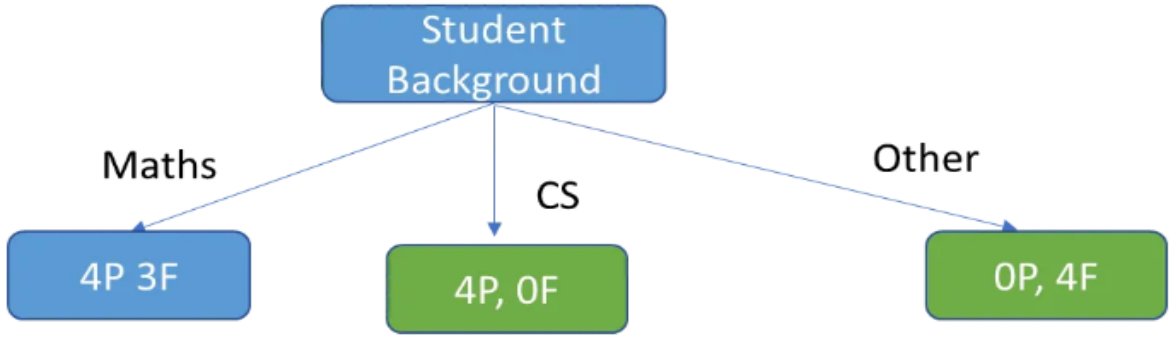
We will drop this variable for the time being and move on to evaluate the other variables. The spreadsheet below shows the Entropy calculations for all the variables:

	Entropy Node	Average Entropy	Information Gain
Parent	0.9968		
working	0.9183	0.8110	
Not_work	0.6500		0.1858
Bkgrd_Ma	0.9852	0.4598	
Bkgrd_CS	0.0000		
Bkgrd_oth	0.0000		0.5370
online_co	0.9544	0.9688	
online_no	0.9852		0.0280

Root Node Entropy Calculations (source: author)

To find the root node attribute we look at the Information gain from Student Background vis a vis initial parent entropy. This shows the maximum reduction of .5370. Hence, this is the attribute that would be selected as the root node. The other's variables — 'Working Status' and 'Online Courses' show a much lower decrease in entropy vis a vis the parental node.

So, on the basis of the above calculations, we have determined what the root node would be. The tree would now look as follows:



Root Node of Decision Tree

Student Background splits the target variable into 3 groups. Everyone from CS background clearly passes and hence this is a terminal or leaf node. Everyone from Other backgrounds fails and this is also a terminal node. Maths background is split into 3 Pass and 4 Fail and hence it is impure and there is some scope for further splitting to attain greater purity.

Now to split the Maths background sub node, we need to calculate Entropy and Information Gain for the remaining variables, i.e., Working Status and Online Courses. We would then select the variable that shows the highest Information Gain.

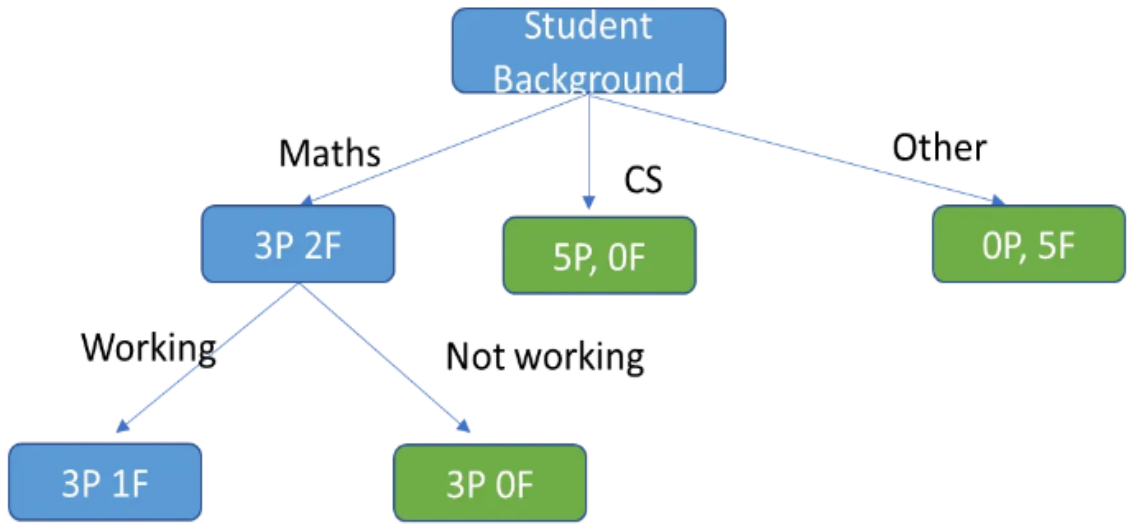
The Entropy and Information Gain Calculations for the Maths Background node can be seen in the table below. Notice, we now have the Maths Background as the node that is being split, hence average Entropy for the splits is calculated using it as a base.

The Entropy for each potential split is:

	Entropy Node	Average Entropy	Information Gain
Bkgrd_Ma	0.9852		
working	0.8113	0.4636	0.5216
Not_work	0.0000		
online_co	0.9183	0.9533	0.0319
online_no	1.0000		

Splitting the Maths Subnode (Image source: author)

As we can see Information Gain is higher for the Working Status variable. Hence this is the variable used to continue branching.



Maths Node Branching

We now see that the Maths node has split into 1 terminal node on the right and one node which is still impure. Notice, now almost all our nodes are terminal nodes. There is only one node which is not terminal. We can try splitting it further using Other Online Courses. Anyway, you get the picture. In any case most Decision Trees do not necessarily split to the point where every node is a terminal node. Most algorithms have built in stops which we will discuss a little further down. Further, if the Decision Tree continues to split we have another problem which is that of overfitting. Again we shall discuss that below after we have briefly reviewed an alternative approach to developing a Decision Tree using the Gini Index.

Gini Index

The other way of splitting a decision tree is via the Gini Index. The Entropy and Information Gain method focuses on purity and impurity in a node. The Gini Index or Impurity measures the probability for a random instance being misclassified when chosen randomly. The lower the Gini Index, the better the lower the likelihood of misclassification.

The formula for Gini Index

$$Gini = 1 - \sum_{i=1}^{j} P(i)^2$$

Where j represents the no. of classes in the target variable — Pass and Fail in our example

P(i) represents the ratio of Pass/Total no. of observations in node.

So, Let's take an example from the decision tree above. Let's begin with the root node and calculate the Gini Index for each of the splits. The Gini Index has a minimum (highest level of purity) of 0. It has a maximum value of .5. If Gini Index is .5, it indicates a random assignment of classes.

Now let us calculate the Gini index for the root node for Student Background attribute. In this case we have 3 nodes. Gini formula requires us to calculate the Gini Index for each sub node. Then do a weighted average to calculate the overall Gini Index for the node.

Maths sub node: 4Pass, 3Fail

$$Gini_{maths} = 1 - \left(\frac{4}{7}\right)^2 - \left(\frac{3}{7}\right)^2 = .4897$$

CS sub node: 4Pass, 0 Fail

$$Gini_{CS} = 1 - \left(\frac{4}{4}\right)^2 - \left(\frac{0}{5}\right)^2 = 0$$

Others sub node: 0Pass, 4 Fail

$$Gini_{others} = 1 - \left(\frac{0}{4}\right)^2 - \left(\frac{4}{4}\right)^2 = 0$$

As we can see the probability for misclassification in CS node is zero, since everyone passes. Similarly no scope for misclassification on Others node as everyone fails. Only the maths node has possibility of misclassification, and this is quite high, given that the maximum Gini Index is .5.

The overall Gini Index for this split is calculated similarly to the entropy as weighted average of the distribution across the 3 nodes.

$$Gini_{bkgrd} = \frac{7}{15} * .4897 + \frac{4}{15} * 0 + \frac{4}{15} * 0 = .2286$$

Similarly, we can also compute the Gini Index for Working Status and Online Courses. These are given below:

Working/Not working

$$Gini_{working} = 1 - \left(\frac{6}{9}\right)^2 - \left(\frac{3}{9}\right)^2 = .44$$

$$Gini_{notworking} = 1 - \left(\frac{5}{6}\right)^2 - (6)^2 = .278$$

$$Gini_{workstatus} = \frac{9}{15} * .44 + \frac{6}{15}.278 = .378$$

Online Courses

$$Gini_{online} = 1 - \left(\frac{5}{8}\right)^2 - \left(\frac{3}{8}\right)^2 = .4688$$

$$Gini_{notonline} = 1 - \left(\frac{3}{7}\right)^2 - \left(\frac{4}{7}\right)^2 = .4898$$

$$Gini_{online} = \frac{8}{15} * .4688 + \frac{7}{15}.4898 = .479$$

The Gini Index is lowest for the Student Background variable. Hence, similar to the Entropy and Information Gain criteria, we pick this variable for the root node. In a similar fashion we would again proceed to move down the tree, carrying out splits where node purity is less

Gini Index vs Information Gain

Depending on which impurity measurement is used, tree classification results can vary. This can make small (or sometimes large) impact on your model. There seems to be no one preferred approach by different Decision Tree algorithms. For example, CART uses Gini; ID3 and C4.5 use Entropy.

The Gini index has a maximum impurity is 0.5 and maximum purity is 0, whereas Entropy has a maximum impurity of 1 and maximum purity is 0.

The degree of gini index varies from 0 to 1,

- Where 0 depicts that all the elements be allied to a certain class, or only one class exists there.
- The gini index of value as 1 signifies that all the elements are randomly zdistributed across various classes, and
- A value of 0.5 denotes the elements are uniformly distributed into some classes.

Pruning

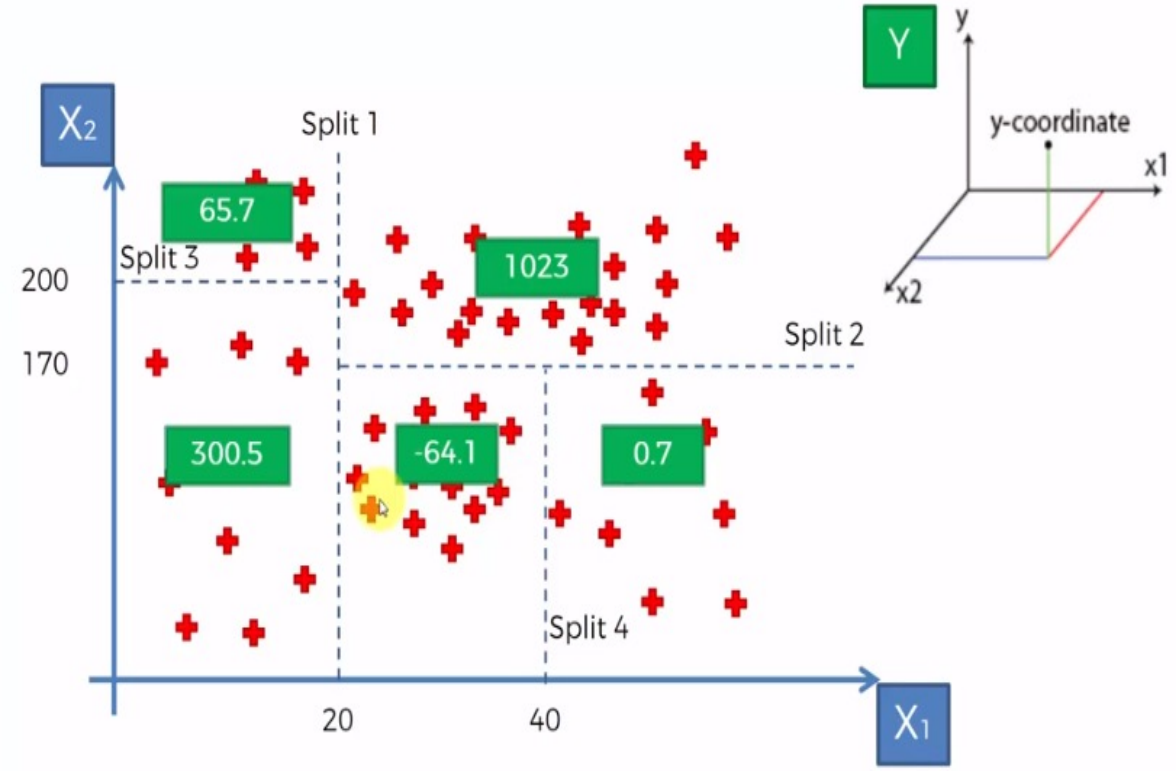
Overfitting can be a big challenge with Decision Trees. Even in our toy example, we can see the algorithm continues to split till it reaches a leaf node. Often the leaf node may just have one or two instances. This will clearly lead to a complex tree structure which may not generalize well to a test scenario. This is because each leaf will represent a very specific set of attribute combinations that are seen in the training data, and the tree will not be able to classify attribute combinations not seen in the training data. There are several ways we can prevent the decision tree from becoming too unwieldy: 3 broad approaches to avoiding overfitting are distinguished:

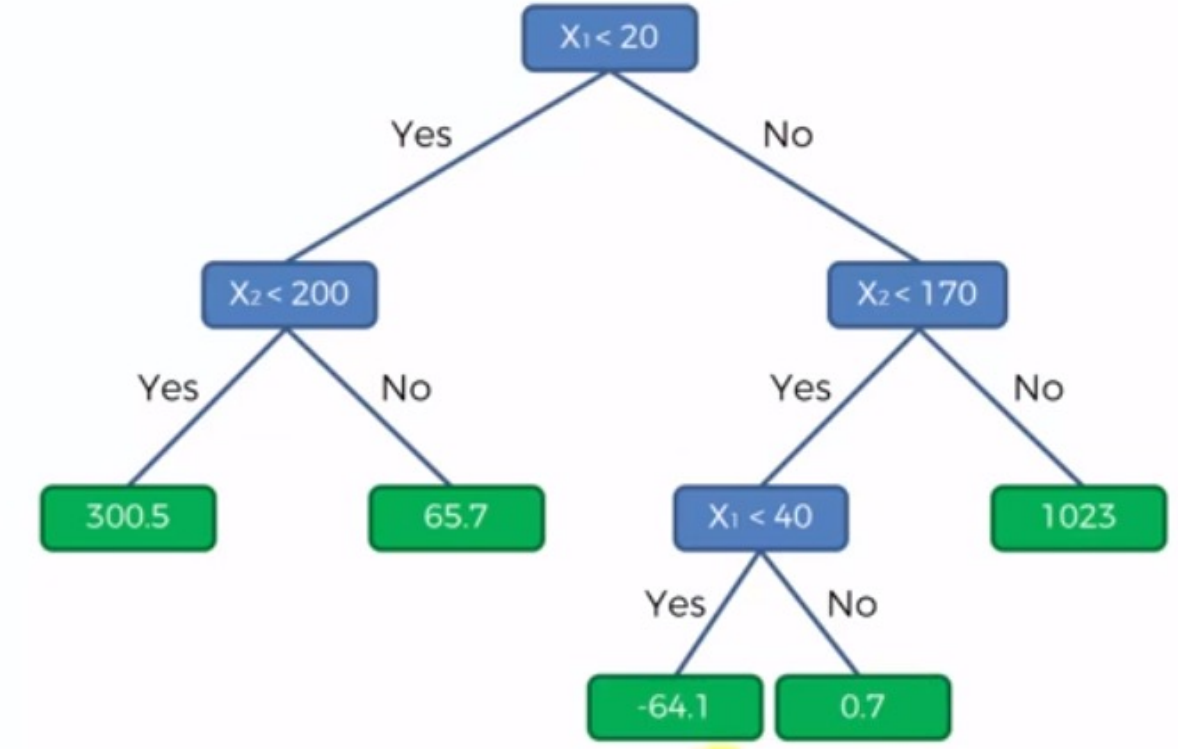
1) **Pre pruning or Early stopping**: Preventing the tree from growing too big or deep. The pre-pruning technique involves tuning the hyperparameters of the decision tree model prior to the training pipeline. The hyperparameters of the DecisionTreeClassifier in SkLearn include max_depth, min_samples_leaf, min_samples_split which can be tuned to early stop the growth of the tree and prevent the model from overfitting.

2) **Post Pruning:** Allowing a tree to grow to its full depth and then getting rid of various branches based on various criteria. A challenge with post pruning is that a decision tree can grow very deep and large and hence evaluating every branch can be computationally expensive. An important post pruning technique is **Cost complexity pruning** (ccp) which provides a more efficient solution in this regard.

3) **Ensembling** or using averages of multiple models such as Random Forest

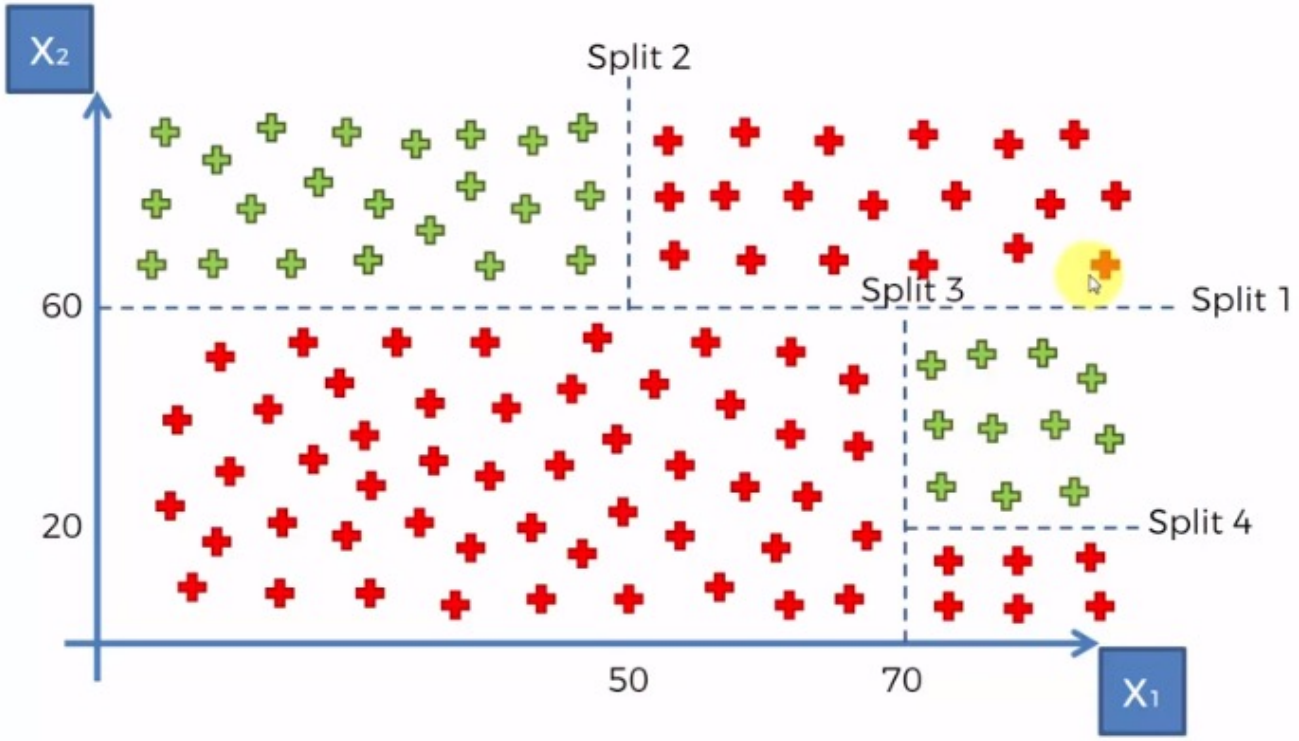
Decision Tree- Regression

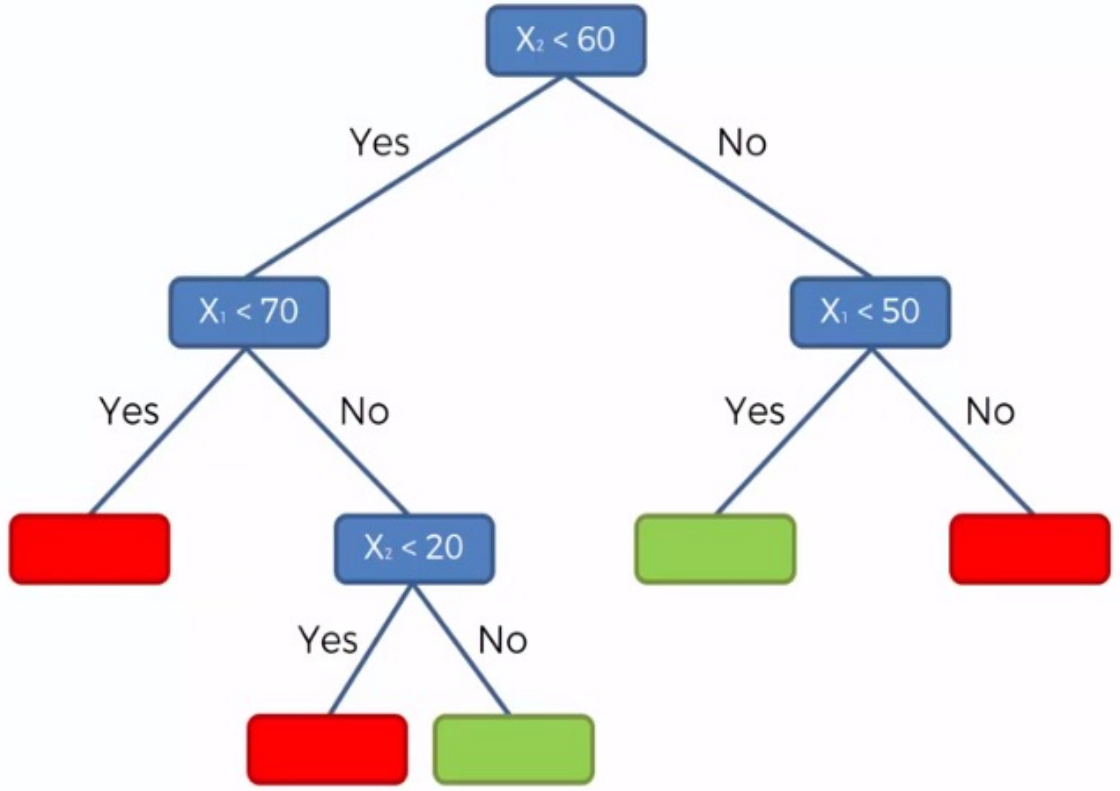




The way it works is you just take the averages of each of your terminal leaves.So you take the average of y for all of these points in each terminal. Now, the value of Y for any new point will be assigned based on the average value of that terminal it falls into.

Decision Tree- Classification





If we don't go down the decision tree and check more conditions so instead of going down to the very end we can stop at any point and then just use the probabilities to predict our classification. Instead of split 4, we can stop at split 3 and based on the probability of green and red dots on the right terminal we can easily assign the terminal as green as green dots probability is more than red.

Hyper Parameter Tuning Decision Tree

Hyperparameter tuning for decision trees involves optimizing the parameters that control the tree's growth and structure. Decision trees are prone to overfitting, so tuning these hyperparameters is essential to improve their performance. Common hyperparameters to consider for decision trees include:

1. **Max Depth (max_depth):** This hyperparameter controls the maximum depth of the decision tree. A deeper tree can capture more complex relationships in the data but is also more likely to overfit. You can try different values for `max_depth` and select the one that performs best on your validation data.

2. **Min Samples Split (min_samples_split):** This parameter sets the minimum number of samples required to split an internal node. It helps to control overfitting. Higher values lead to a simpler tree. Start with values like 2 or 5 and increase as needed.

3. **Min Samples Leaf (min_samples_leaf):** This parameter sets the minimum number of samples required to be at a leaf node. It helps prevent tiny leaves that could lead to overfitting. Like `min_samples_split`, you can start with small values and increase them.

4. **Max Features (max_features):** This hyperparameter determines the maximum number of features to consider when splitting a node. It's crucial to prevent the tree from being too sensitive to specific features. Common values are 'sqrt' (square root of the total number of features), 'log2' (base-2 logarithm of the total number of features), or an integer representing a fixed number of features.
   - If int, then consider max_features features at each split.
   - If float, then max_features is a fraction and max(1, int(max_features * n_features_in_)) features are considered at each split.
   - If “sqrt”, then max_features=sqrt(n_features).
   - If “log2”, then max_features=log2(n_features).
   - If None, then max_features=n_features

5. **Criterion:** Decision trees can use different criteria to measure the quality of splits, such as 'gini' for Gini impurity or 'entropy' for information gain. The choice of criterion can impact the tree's performance, and it's worth experimenting with both.

6. **Max Leaf Nodes (max_leaf_nodes):** This parameter limits the maximum number of leaf nodes in the tree. Setting this can help control the size of the tree and prevent overfitting.

7. **Min Impurity Decrease (min_impurity_decrease):** It specifies a threshold for splitting nodes based on impurity. A node will be split only if the impurity decrease is above this threshold.

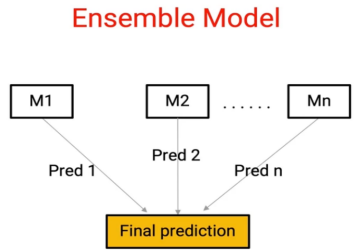
To tune these hyperparameters effectively, you can use techniques like grid search or randomized search along with cross-validation. Grid search involves specifying a range of values for each hyperparameter, and it exhaustively searches through all combinations. Randomized search, on the other hand, samples from a distribution over hyperparameters and is often more efficient.

```python
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV

# Define the hyperparameters and their possible values
parameters = {
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['auto', 'sqrt', 'log2'],
    'criterion': ['gini', 'entropy']
}

# Create a decision tree classifier
dtree = DecisionTreeClassifier(random_state=42)

# Perform grid search with cross-validation
grid_search = GridSearchCV(estimator=dtree,
                           param_grid=parameters,
                           cv=5,
                           scoring='accuracy',
                           n_jobs=-1)
grid_search.fit(X_train, y_train)

# Print the best hyperparameters and corresponding accuracy
print("Best Hyperparameters:", grid_search.best_params_)
print("Best Accuracy:", grid_search.best_score_)
```

N_jobs=-1 to run all processors in your machine which will optimized the grid search process

Ensemble Technique

Ensembling is nothing but the technique to combine several individual predictive models to come up with the final predictive model.



Simple Ensemble Techniques

1. Max Voting-

The max voting method is generally used for classification problems. In this technique, multiple models are used to make predictions for each data point. The predictions by each model are considered as a 'vote'. The predictions which we get from the majority of the models are used as the final prediction.

For example, when you asked 5 of your colleagues to rate your movie (out of 5); we'll assume three of them rated it as 4 while two of them gave it a 5. Since the majority gave a rating of 4, the final rating will be taken as 4. **You can consider this as taking the mode of all the predictions.**

The result of max voting would be something like this:

Colleague 1	Colleague 2	Colleague 3	Colleague 4	Colleague 5	Final rating
5	4	5	4	4	4

2. Averaging- Similar to the max voting technique, multiple predictions are made for each data point in averaging. In this method, we take an average of predictions from all the models and use it to make the final prediction. Averaging can be used for making predictions in regression problems or while calculating probabilities for classification problems.

For example, in the below case, the averaging method would take the average of all the values.

i.e. (5+4+5+4+4)/5 = 4.4

Colleague 1	Colleague 2	Colleague 3	Colleague 4	Colleague 5	Final rating
5	4	5	4	4	4.4

Ensemble Technique

3. Weighted Average

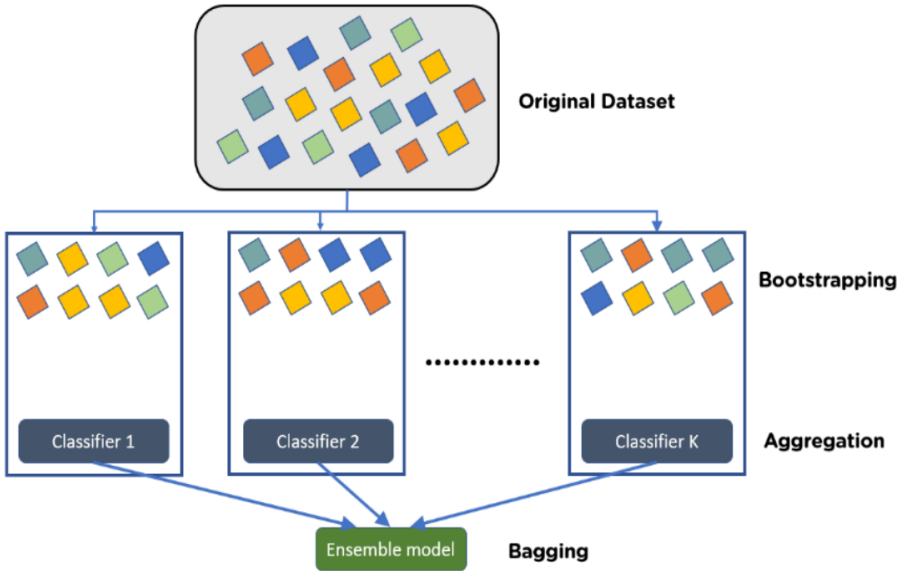
This is an extension of the averaging method. All models are assigned different weights defining the importance of each model for prediction. For instance, if two of your colleagues are critics, while others have no prior experience in this field, then the answers by these two friends are given more importance as compared to the other people.

The result is calculated as [(5*0.23) + (4*0.23) + (5*0.18) + (4*0.18) + (4*0.18)] = 4.41.

Colleague 1	Colleague 2	Colleague 3	Colleague 4	Colleague 5	Final rating	
weight	0.23	0.23	0.18	0.18	0.18	
rating	5	4	5	4	4	4.41

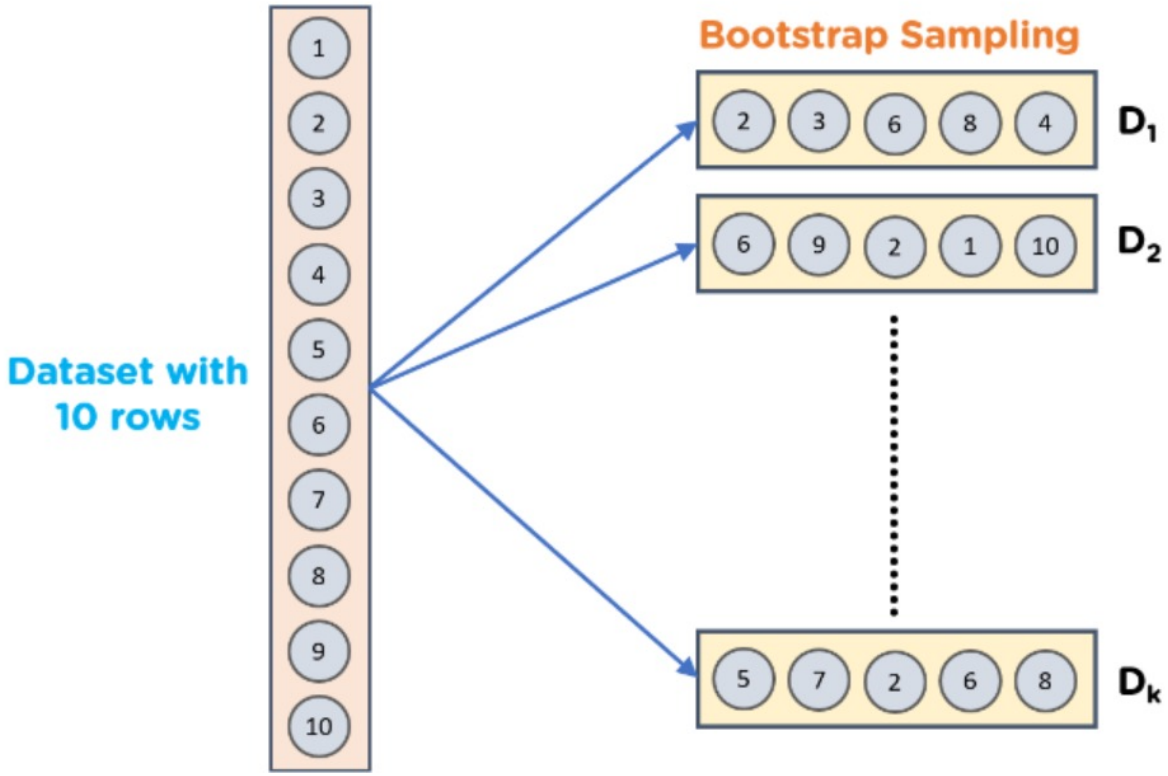
Bagging

Bagging, also known as Bootstrap aggregating, is an ensemble learning technique that helps to improve the performance and accuracy of machine learning algorithms. It is used to deal with bias-variance trade-offs and reduces the variance of a prediction model. Bagging avoids overfitting of data and is used for both regression and classification models, specifically for decision tree algorithms



What Is Bootstrapping?

Bootstrapping is the method of randomly creating samples of data out of a population with replacement to estimate a population parameter.



Implementation Steps of Bagging

- **Step 1:** Multiple subsets are created from the original data set with equal tuples, selecting observations with replacement.
- **Step 2:** A base model is created on each of these subsets.
- **Step 3:** Each model is learned in parallel with each training set and independent of each other.
- **Step 4:** The final predictions are determined by combining the predictions from all the models.

Feature Importance

Load the data set and split for training and testing.

```
boston = load_boston()
X = pd.DataFrame(boston.data, columns=boston.feature_names)
y = boston.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state
```

Fit the Random Forest Regressor with 100 Decision Trees:

```
rf = RandomForestRegressor(n_estimators=100)
rf.fit(X_train, y_train)
```

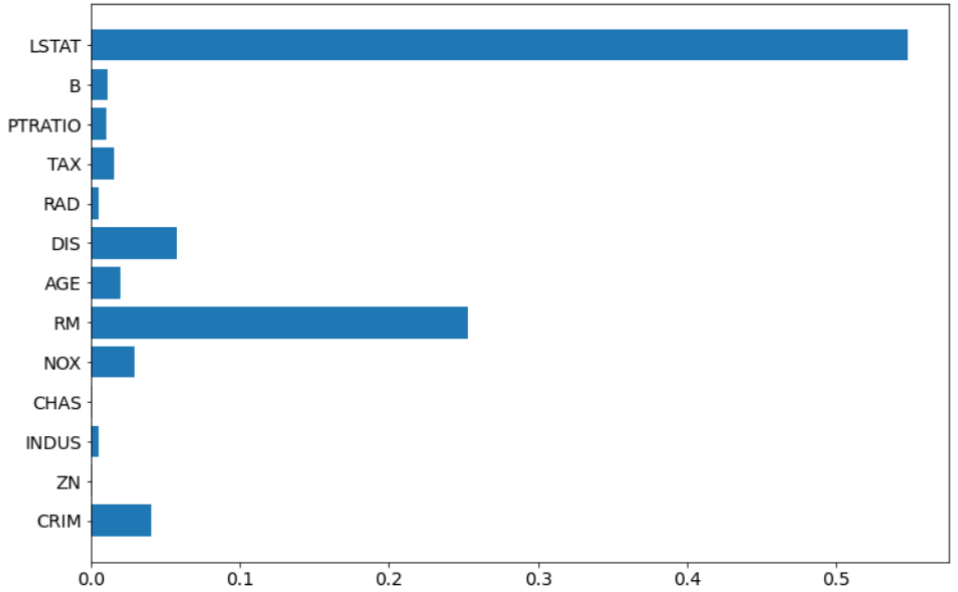
To get the feature importances from the Random Forest model use the `feature_importances_` attribute:

```
rf.feature_importances_
```

```
array([0.04054781, 0.00149293, 0.00576977, 0.00071805, 0.02944643,
       0.25261155, 0.01969354, 0.05781783, 0.0050257 , 0.01615872,
       0.01066154, 0.01185997, 0.54819617])
```

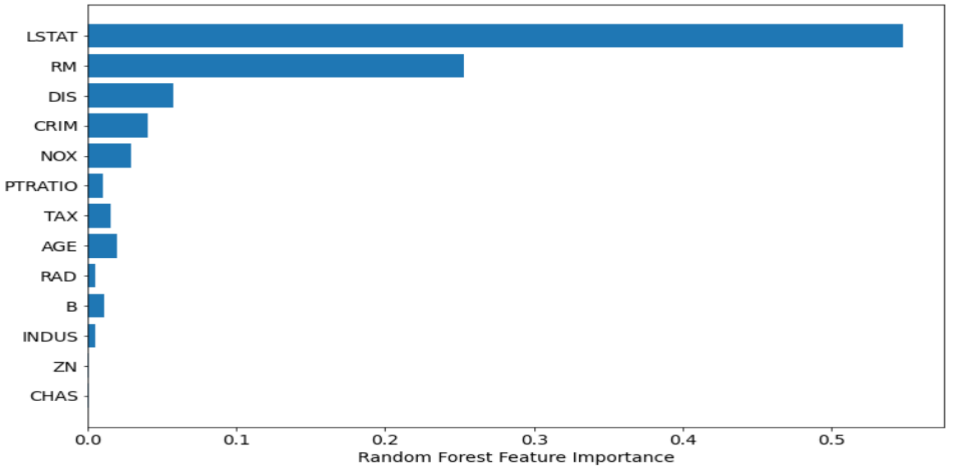
Let's plot the importances (chart will be easier to interpret than values).

```
plt.barh(boston.feature_names, rf.feature_importances_)
```



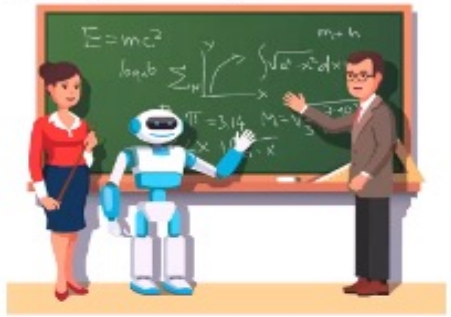
To have even better chart, let's sort the features, and plot again:

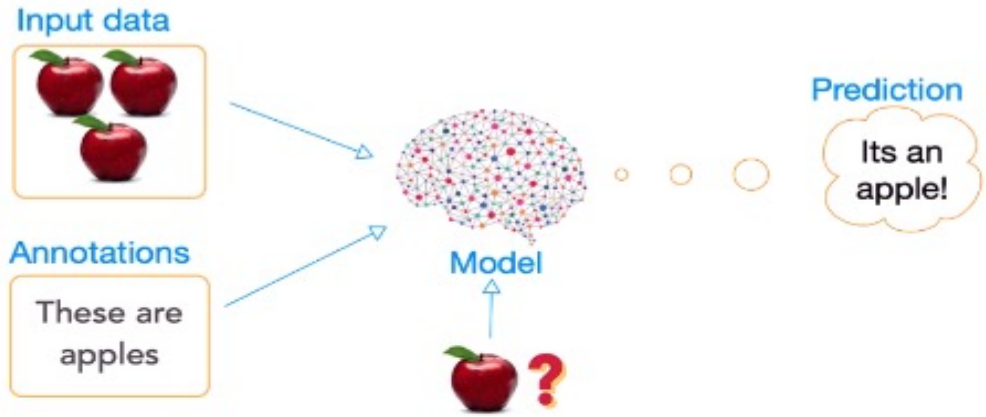
```
sorted_idx = rf.feature_importances_.argsort()
plt.barh(boston.feature_names[sorted_idx], rf.feature_importances_[sorted_idx])
plt.xlabel("Random Forest Feature Importance")
```



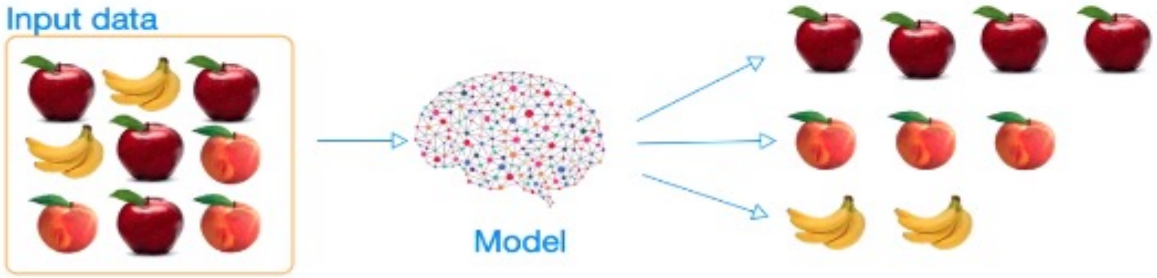
Clustering

Supervised Learning
(e.g. Regression, Classification)



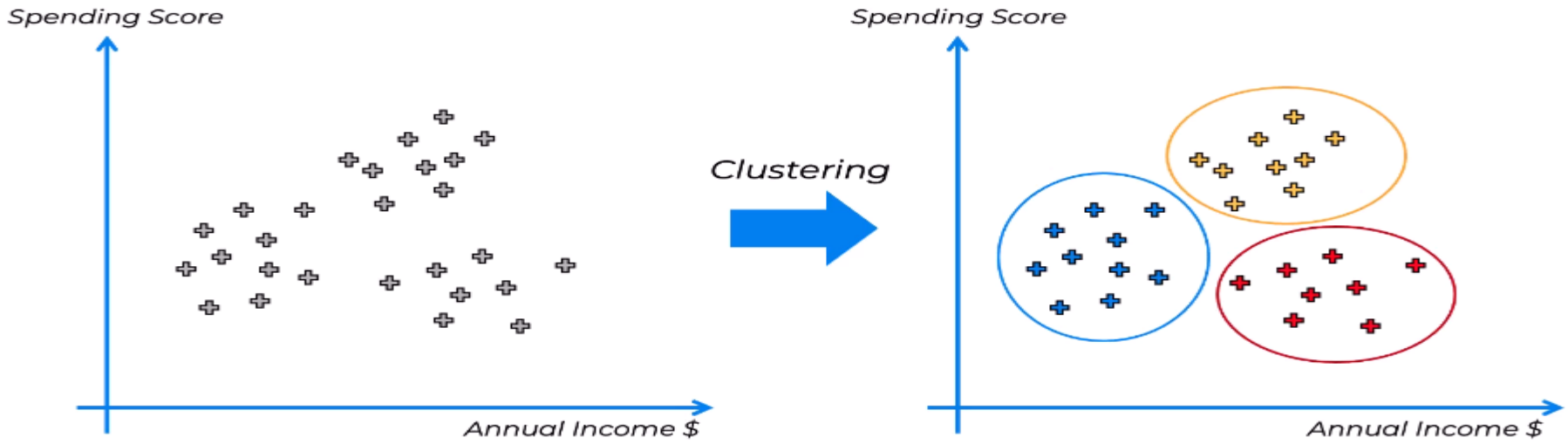


Unsupervised Learning
(e.g. Clustering)



In a nutshell, in supervised learning, you give the model an opportunity to train where it has the answers. In unsupervised learning, you don't have the answers to supply to the model.

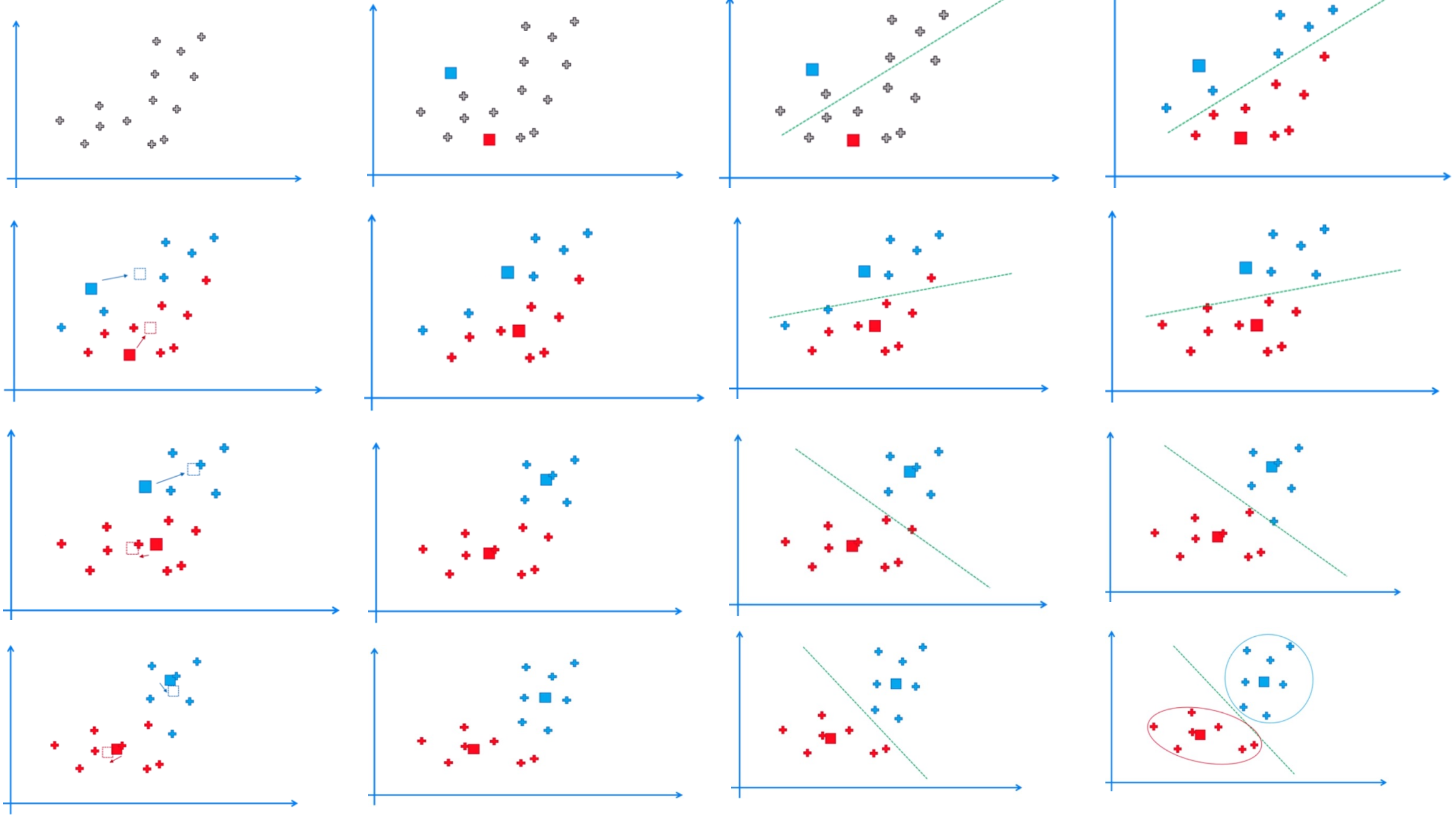
Clustering



Here we have an X Y-axis with annual income of your customers, of a store, for example and the spending score. So how often do they spend? How much do they buy? All their spending patterns. All of this is combined already into a spending score.

So when you plot your customers, it might look like above, and you don't have any preexisting categories.. You don't have any preexisting classes or groups of customers to group them into so that's where you would apply clustering to show that these are probably likely groups of customers and then you could go into this further, dig deeper and understand why these groups might be emerging, what this might be in a business sense, in a spending sense, in a customer sense, understand how to best service these customers, to what kind of promotions to send to one group versus the other, what kind of reminders or what kind of offers to create for these different customers and how to best use this information in your business.

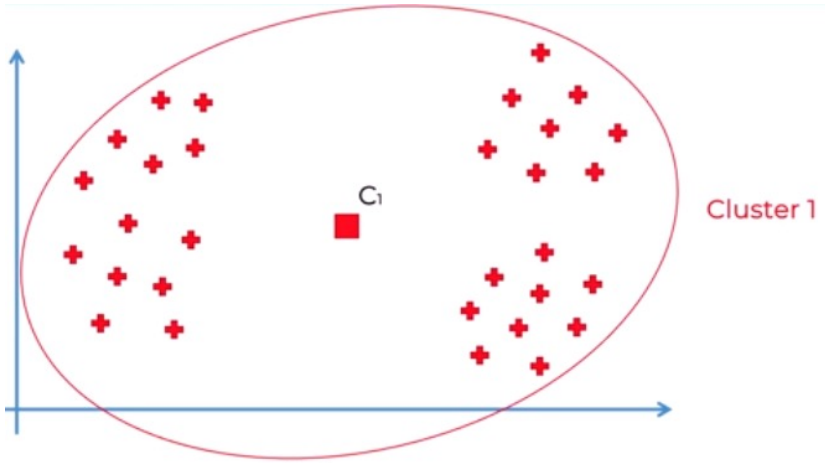
K-Means Clustering



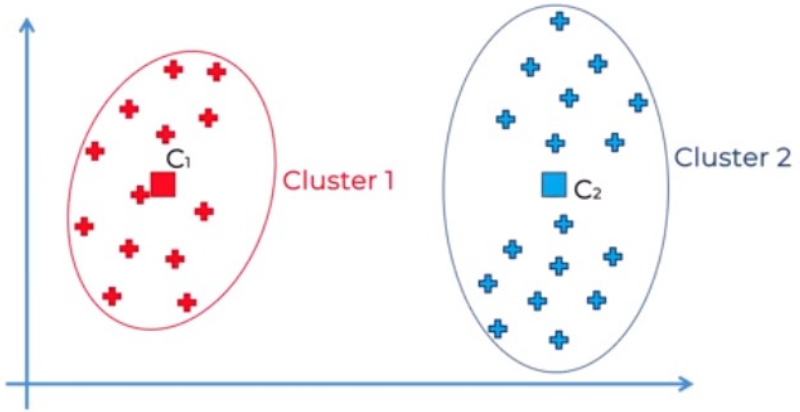
Elbow Method

Within Cluster Sum of Squares:

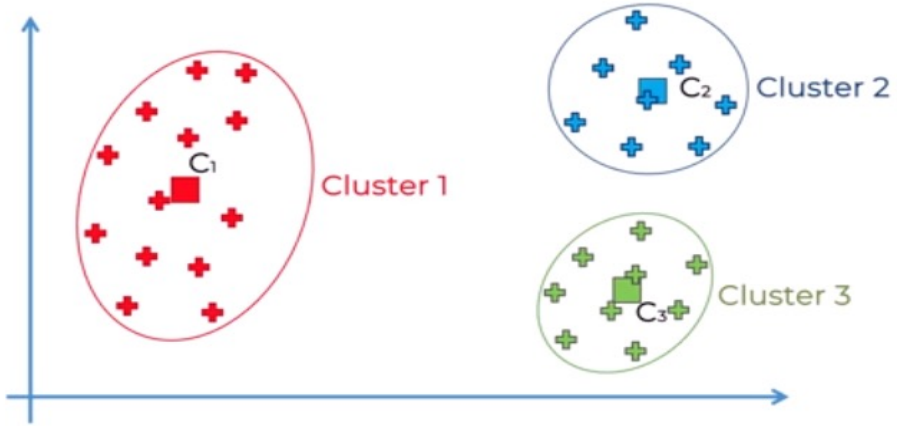
$$\text{WCSS} = \sum_{P_i \text{ in Cluster 1}} \text{distance}(P_i, C_1)^2 + \sum_{P_i \text{ in Cluster 2}} \text{distance}(P_i, C_2)^2 + \ldots$$



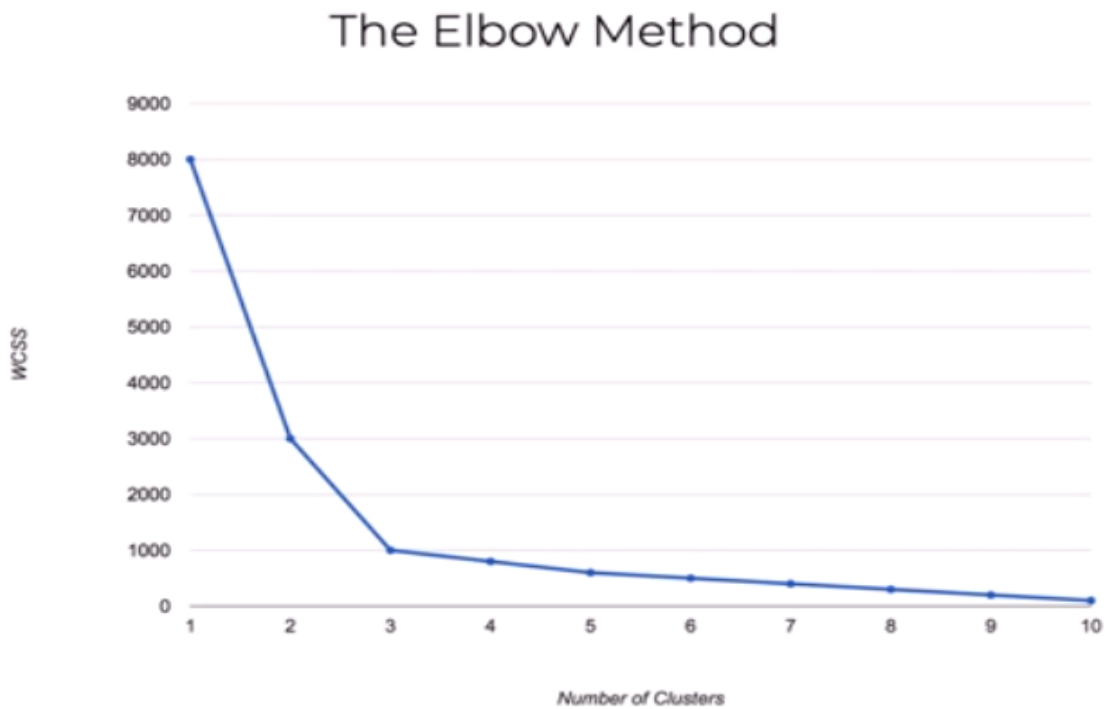
$$\text{WCSS} = \sum_{P_i \text{ in Cluster 1}} \text{distance}(P_i, C_1)^2$$

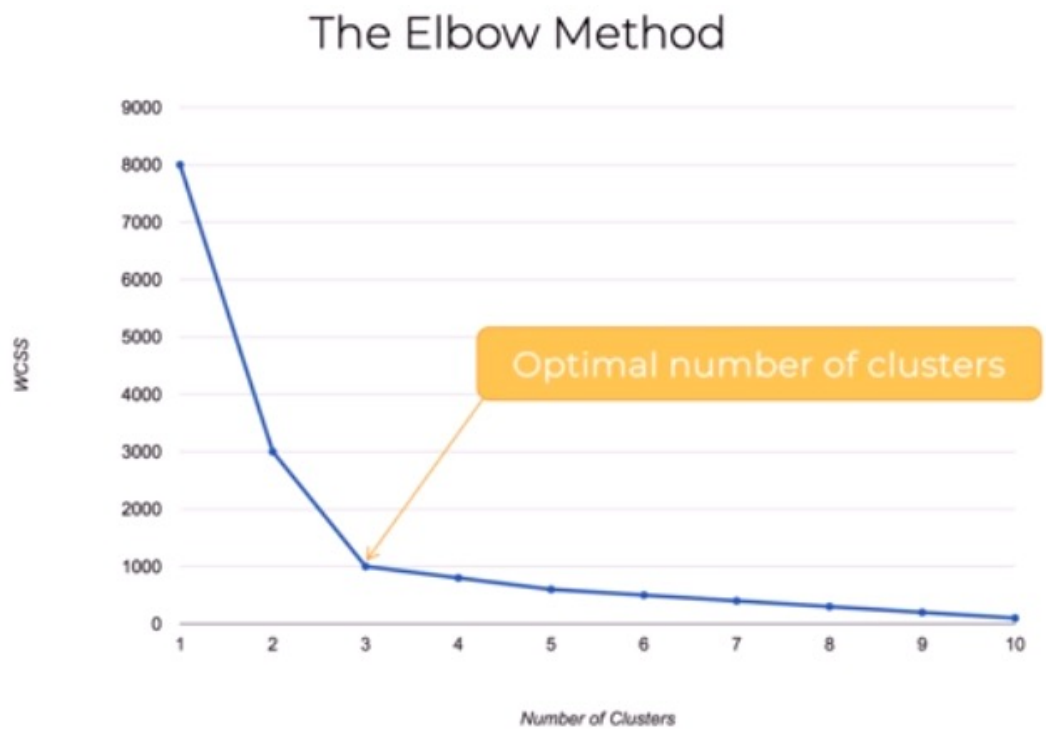


$$\text{WCSS} = \sum_{P_i \text{ in Cluster 1}} \text{distance}(P_i, C_1)^2 + \sum_{P_i \text{ in Cluster 2}} \text{distance}(P_i, C_2)^2$$



$$\text{WCSS} = \sum_{P_i \text{ in Cluster 1}} \text{distance}(P_i, C_1)^2 + \sum_{P_i \text{ in Cluster 2}} \text{distance}(P_i, C_2)^2 + \sum_{P_i \text{ in Cluster 3}} \text{distance}(P_i, C_3)^2$$


The Elbow Method
WCSS
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
1
2
3
4
5
6
7
8
9
10
Number of Clusters


The Elbow Method
WCSS
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
Optimal number of clusters
1
2
3
4
5
6
7
8
9
10
Number of Clusters

Silhouette Analysis

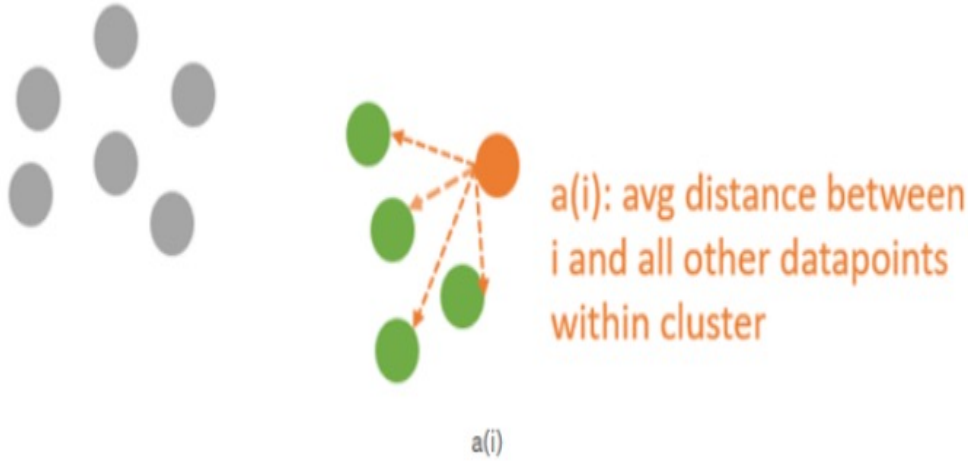
The silhouette coefficient or silhouette score kmeans is a measure of how similar a data point is within-cluster (cohesion) compared to other clusters (separation). The Silhouette score can be easily calculated in Python using the metrics module of the scikit-learn/sklearn library.

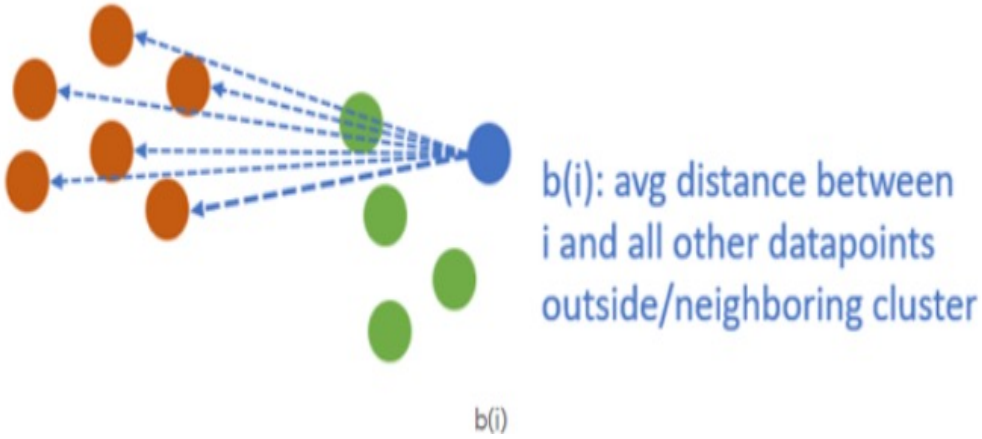
- Select a range of values of k (say 1 to 10).
- Plot Silhouette coefficient for each value of K.

The equation for calculating the silhouette coefficient for a particular data point:

$$S(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}$$

- S(i) is the silhouette coefficient of the data point i.
- a(i) is the average distance between i and all the other data points in the cluster to which i belongs.
- b(i) is the average distance from i to all clusters to which i does not belong.





Silhouette Analysis

We will then calculate the average_silhouette for every k.

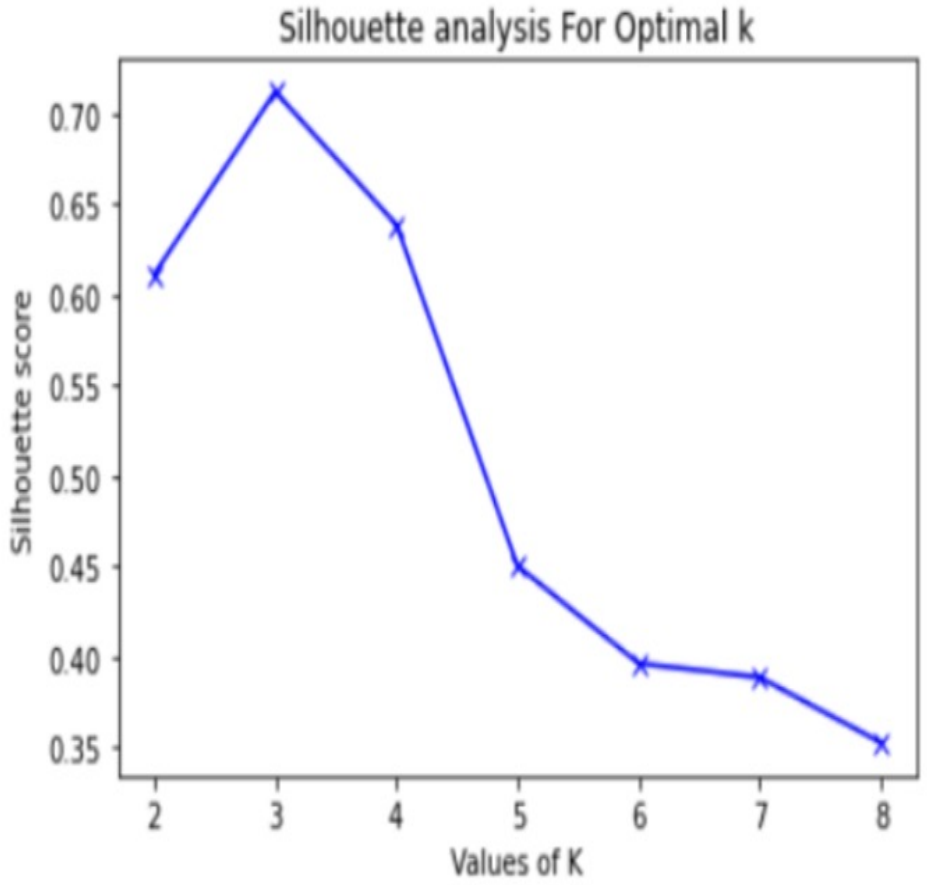
$$\text{AverageSilhouette} = \text{mean}\{S(i)\}$$

Then plot the graph between average_silhouette and K.

Points to Remember While Calculating Silhouette Coefficient:

- The value of the silhouette coefficient is between [-1, 1].
- A score of 1 denotes the best, meaning that the data point i is very compact within the cluster to which it belongs and far away from the other clusters.
- The worst value is -1. Values near 0 denote overlapping clusters.

```
range_n_clusters = [2, 3, 4, 5, 6, 7, 8]
silhouette_avg = []
for num_clusters in range_n_clusters:

 # initialise kmeans
 kmeans = KMeans(n_clusters=num_clusters)
 kmeans.fit(data_frame)
 cluster_labels = kmeans.labels_

 # silhouette score
silhouette_avg.append(silhouette_score(data_frame, cluster_labels))
plt.plot(range_n_clusters,silhouette_avg,'bx-')
plt.xlabel('Values of K')
plt.ylabel('Silhouette score')
plt.title('Silhouette analysis For Optimal k')
plt.show()
```

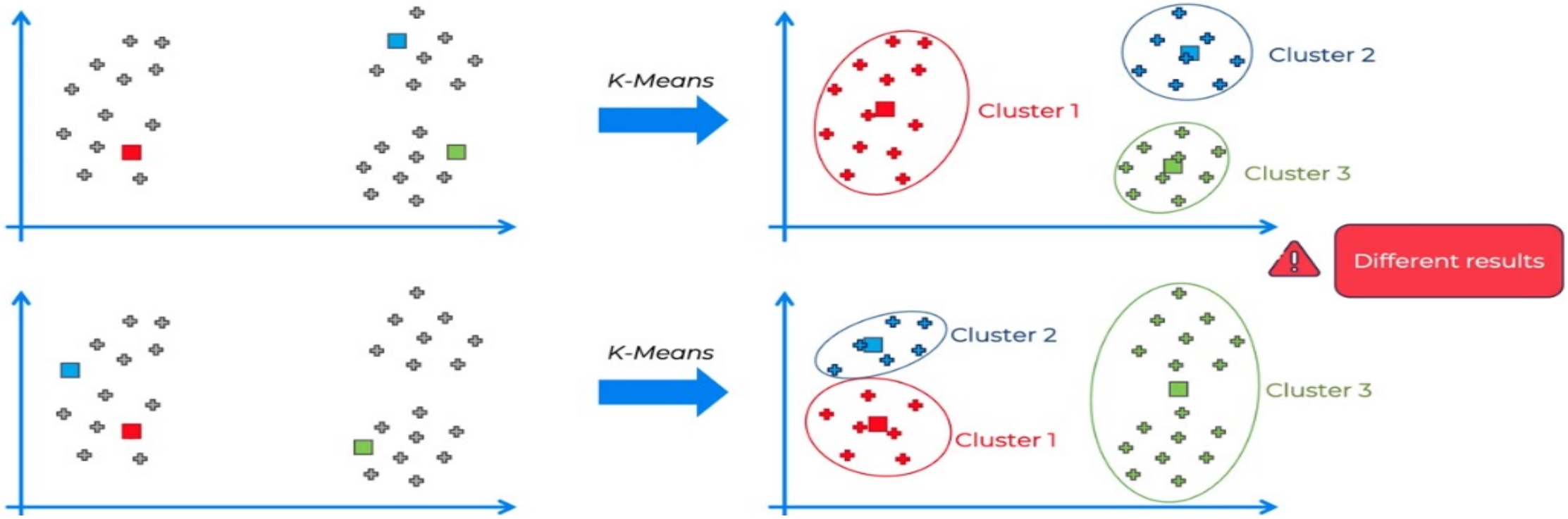


Line plot between K and Silhouette score

We see that the silhouette score is maximized at k = 3. So, we will take 3 clusters.

NOTE: The silhouette Method is used in combination with the Elbow Method for a more confident decision.

K-means ++



The results are different in these two runs of the same machine learning model of the k-means and they're only different because the initialization of the centroids was different. We didn't change anything else and this is called the Random Initialization Trap. It is bad because when you run a machine learning model like in this case, like clustering you want it to be deterministic means we want the result to be the same. So that's what we want to avoid and that's what K-Means ++ is designed to combat. It's basically the same as K-Means, but it adds some steps at the beginning to initialize centroids in a certain way.

K-means ++

K-Means++ Initialization Algorithm:

Step 1: Choose first centroid at random among data points

Step 2: For each of the remaining data points compute the distance (D) to the nearest out of already selected centroids

Step 3: Choose next centroid among remaining data points using weighted random selection – weighted by D^2

Step 4: Repeat Steps 2 and 3 until all k centroids have been selected

Step 5: Proceed with standard k-means clustering

Let's say that the first centroid was initialized here at random. Now what we're going to do, or what K-Means ++ will do, is it will measure the distance from every remaining data point to this centroid and then it'll take the value of the distance, and it'll square that value. So now we will select the next centroid at random but it'll be a weighted random selection. So it'll be weighted by this distance squared. So the centroid in the top right, the furthest away, has the highest chance of being picked and let's say that it was selected. Now, we do the procedure again.