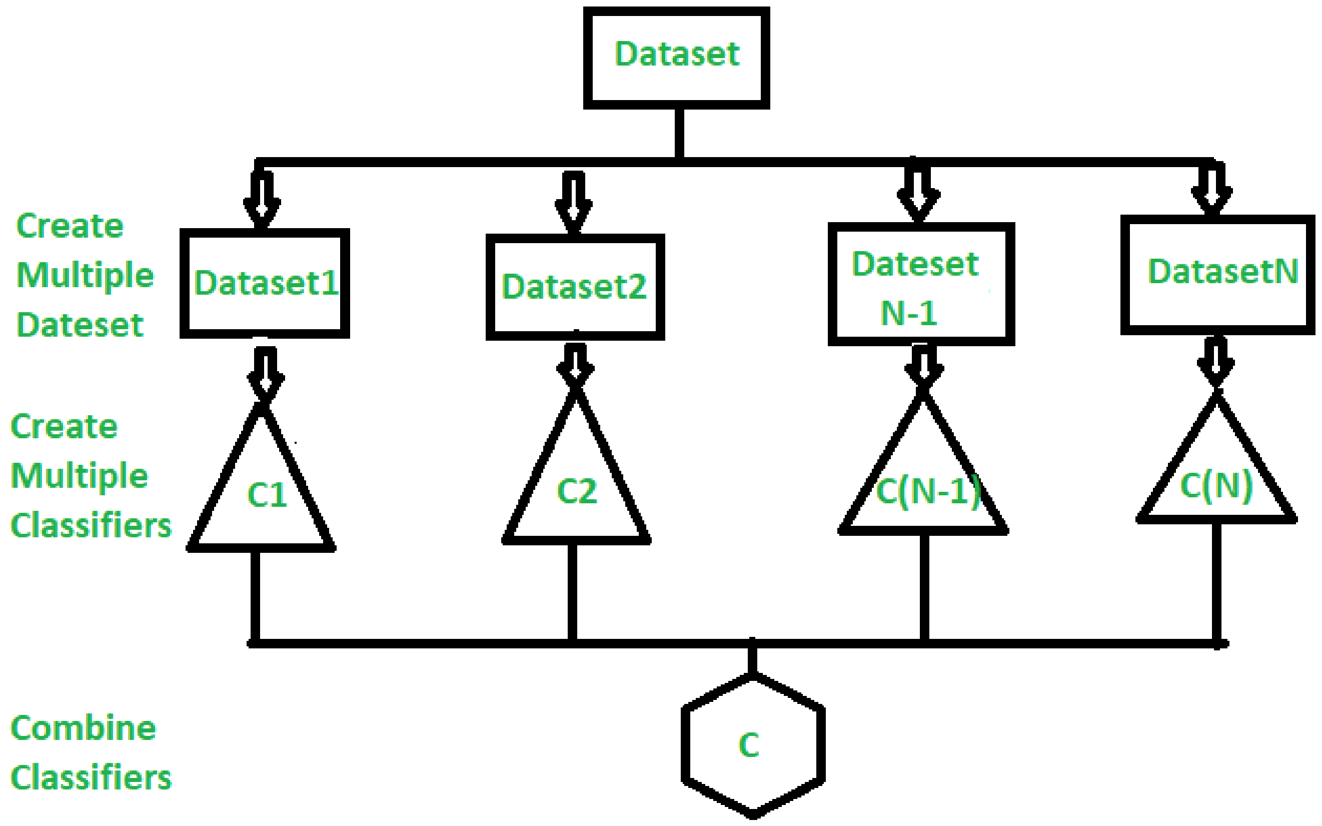
Ensemble Bagging

Agenda:

- Introduction to Ensemble Learning
- Bagging
    - Bootstrapping
    - Steps to Perform Bagging
    - Advantages of Bagging in Machine Learning
- Use Case
    - Problem Statement
    - Exploring Data
    - Data Preprocessing
- Model Prepartion
    - Overfitting Problem in Decision Tree Model
- Random Forest
- Hyperparameter Tuning
    - Grid Search Cross Validation (CV)
    - Random Search Cross Validation (CV)
- Pickling
- Variable Importance

Introduction to Ensemble Learning

- Ensemble learning in machine learning is a technique that combines multiple models to improve the overall performance and robustness of a predictive model.
- Instead of relying on a single model to make predictions, ensemble methods leverage the wisdom of crowds by aggregating predictions from multiple models.
- The underlying idea is that different models might capture different aspects of the data or have different strengths and weaknesses, and by combining them, the ensemble can achieve better predictive accuracy and generalization.



There are several popular ensemble methods, including:

1. Bagging (Bootstrap Aggregating):
   - In bagging, multiple instances of the same base learning algorithm are trained on different subsets of the training data, typically sampled with replacement.
   - The final prediction is often made by averaging or taking a majority vote of the predictions from individual models.
2. Boosting:
   - Boosting is an iterative ensemble technique where base learners are trained sequentially, and each subsequent model focuses on the instances that the previous models misclassified.
   - Boosting algorithms assign higher weights to misclassified instances to force subsequent models to focus more on them, thereby improving overall performance.
3. Random Forest:
   - Random Forest is a specific ensemble method that uses bagging and decision trees as base learners.
   - It constructs multiple decision trees using random subsets of features and averages their predictions to make the final decision. Random Forest tends to reduce overfitting and can handle large datasets efficiently.
4. Stacking (Stacked Generalization):

- Stacking combines the predictions of multiple base models by training a meta-model on top of them.
- Instead of using simple averaging or majority voting, stacking learns to combine the predictions of base models using a higher-level algorithm, often resulting in improved performance.

5. Voting:
    - Voting, also known as majority voting or plurality voting, aggregates predictions from multiple models and selects the class label that receives the most votes.
    - It can be applied to classification tasks and can use different types of models as its base classifiers.

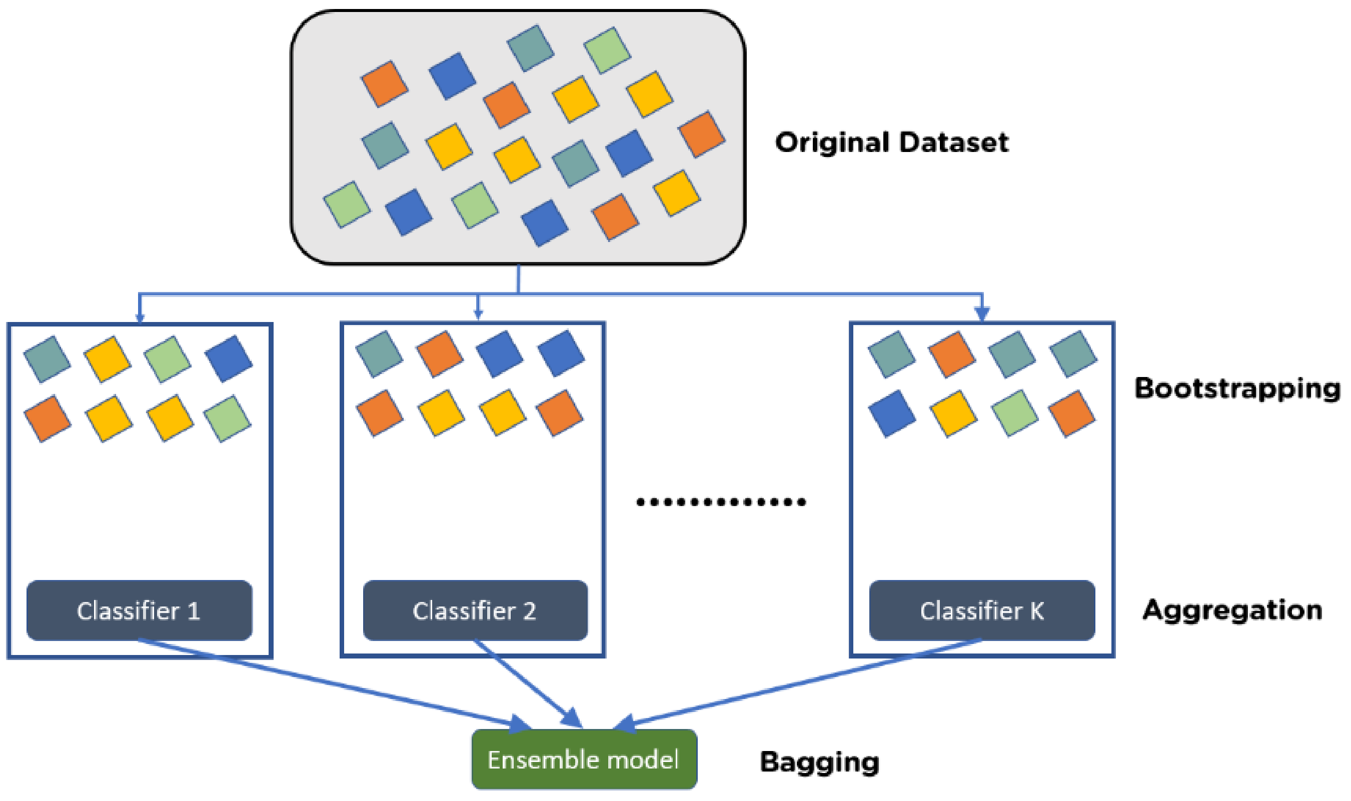
Ensemble learning is widely used in practice because it often leads to better generalization and performance than individual models, especially when dealing with complex and noisy datasets. It is a powerful technique in machine learning and is utilized across various domains, including classification, regression, and anomaly detection.

Bagging

- **Bagging**, which stands for **Bootstrap Aggregating**, is a machine learning ensemble technique used to improve the performance of models, particularly in the case of unstable learners like decision trees.

Here's how it works:

1. **Bootstrap Sampling:** Bagging involves creating multiple subsets of the original dataset through bootstrap sampling. Bootstrap sampling means randomly sampling from the dataset with replacement. This means that some instances may be selected multiple times while others may not be selected at all for a particular subset.
2. **Model Training:** A base model (often the same model type) is trained on each of these subsets independently. Each model is trained on a slightly different version of the original dataset due to the random sampling with replacement.
3. **Aggregation:** Once all the models are trained, predictions are made by each model on unseen data. For regression tasks, the final prediction can be the average of predictions from all models (thus reducing variance), and for classification tasks, the final prediction can be determined by majority voting.



- Bagging helps to reduce overfitting by providing stability to the model.
- By training multiple models on different subsets of the data, it reduces the variance of the final model.
- Additionally, it can help in improving the accuracy and robustness of the model by reducing the impact of outliers and noise in the dataset.

Advantages of Bagging

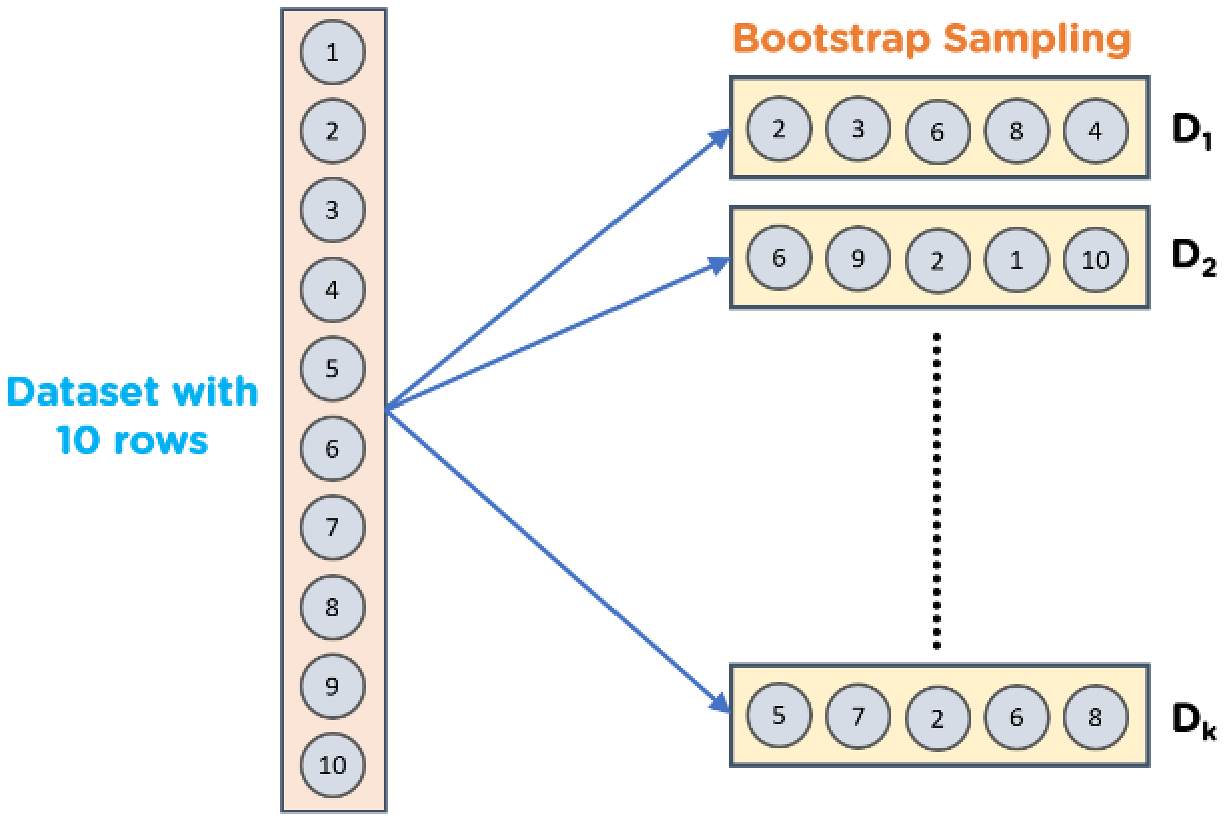
- **Reduces Overfitting:** By averaging multiple models, bagging reduces variance and helps prevent overfitting, especially for high-variance models like decision trees.
- **Improves Stability:** Predictions are more stable and robust because they are averaged over many models.
- **Handles High Variance Models:** Works well with models that tend to overfit, such as decision trees.

Bootstrapping

- Bootstrapping is a statistical resampling technique used to estimate the sampling distribution of a statistic by repeatedly sampling with replacement from the observed data.

Here's how it works:

1. **Sampling with Replacement:** Bootstrapping involves repeatedly drawing samples of the same size as the original dataset, with replacement. This means that each data point has an equal chance of being selected for each sample, and some data points may be selected multiple times while others may not be selected at all.
2. **Estimating Statistics:** After generating multiple bootstrap samples, the statistic of interest (such as the mean, median, standard deviation, or any other parameter) is calculated for each sample.
3. **Estimating the Sampling Distribution:** The distribution of the calculated statistics across all bootstrap samples provides an estimate of the sampling distribution of the statistic. This distribution can then be used to make inferences about the population parameter, such as estimating the confidence intervals or conducting hypothesis tests.



- Bootstrapping is particularly useful when the underlying distribution of the data is unknown or when the sample size is small.
- It allows for robust estimation of parameters and provides a non-parametric approach to statistical inference.
- Additionally, bootstrapping can be applied to various statistical techniques, including regression analysis, hypothesis testing, and model evaluation in machine learning.

Steps to Perform Bagging

1. **Random Sampling from Training Dataset:** In Bagging, you start by randomly selecting samples from the training dataset. However, unlike typical random sampling, in Bagging, you sample with replacement. This means that each data point has an equal chance of being selected for each sample, and some data points may be selected multiple times while others may not be selected at all.
2. **Random Subset of Features:** For each sample, you also randomly select a subset of features. This helps in introducing diversity among the base models. By considering only a subset of features for each model, Bagging ensures that the individual models don't rely on the same set of features, which can lead to more diverse and robust models.
3. **Model Creation:** With the sampled observations and features, a base model (typically a decision tree) is created. This model is trained on the randomly sampled subset of the training data. Each base model is essentially learning from a slightly different perspective of the data due to the randomness introduced through sampling.
4. **Node Splitting:** As the decision tree is being built, at each node, the algorithm chooses the best feature among the randomly selected subset of features to split the node. This process continues recursively until a stopping criterion is met (e.g., reaching a maximum depth or minimum number of samples per leaf node).
5. **Aggregation of Predictions:** Once all base models are trained, predictions are made for new instances by each model. For regression tasks, the final prediction can be the average of predictions from all models, while for classification tasks, the final prediction can be determined by majority voting.
6. **Final Prediction:** The final prediction from Bagging is the aggregated prediction from all individual models. This ensemble approach often leads to improved generalization performance compared to using a single model, as it helps in reducing variance and overfitting.

Advantages of Bagging in Machine Learning

1. **Minimization of Overfitting:**
    - One of the primary advantages of Bagging is its ability to minimize overfitting.
    - Overfitting occurs when a model learns to capture noise or irrelevant patterns in the training data, leading to poor performance on unseen data.
    - By training multiple base models on different subsets of the training data and aggregating their predictions, Bagging helps in reducing the impact of overfitting.
    - Each base model learns from a slightly different perspective of the data due to the randomness introduced through sampling, which helps in generalizing better to unseen data.

2. **Improved Model Accuracy:**
    - Bagging typically leads to improved model accuracy compared to using a single model.
    - By combining predictions from multiple base models, Bagging reduces the variance of the final prediction, resulting in a more stable and reliable model.
    - This ensemble approach often results in better performance metrics such as higher accuracy, precision, recall, or F1-score, depending on the task at hand.
3. **Efficient Handling of Higher-Dimensional Data:**
    - Bagging is particularly effective for handling higher-dimensional data, where the number of features (dimensions) is large.
    - When dealing with high-dimensional data, it becomes more challenging for a single model to capture all relevant patterns and relationships in the data without overfitting.
    - By randomly selecting subsets of features for each base model, Bagging introduces diversity among the models, which helps in capturing different aspects of the data efficiently.
    - This makes Bagging well-suited for tasks involving datasets with a large number of features, such as image classification, text classification, or genomic data analysis.

Use Case

This is the same case study which we have discussed in Decision Tree.

Problem Statement

- An organization aims to predict employee attrition using a machine learning model.
- The dataset provided contains various employee attributes, including demographic information, job-related factors, and indicators of attrition.
- The goal is to preprocess the dataset, address any data quality issues, perform feature encoding, and split the data into training and testing sets.
- Additionally, strategies to handle class imbalance in the target variable ("Attrition") need to be implemented to ensure the model's predictive accuracy and reliability.
- The objective is to develop a robust machine learning model capable of accurately predicting employee attrition, thereby assisting the organization in identifying potential turnover risks and implementing proactive retention strategies.

Exploring Data

```
# Import necessary libraries
import numpy as np  # NumPy for numerical operations
import pandas as pd  # Pandas for data manipulation
from matplotlib import pyplot as plt  # Matplotlib for data visualization
import io  # io for handling file I/O
import random  # Random for generating random numbers
```

```
# Import necessary libraries
import warnings  # Import the warnings module

# Filter out specific warnings
warnings.filterwarnings("ignore", category=UserWarning)  # Ignore UserWarnings
warnings.filterwarnings("ignore", category=RuntimeWarning)  # Ignore RuntimeWarni
```

```
# Importing the necessary function from the google.colab library
from google.colab import drive

# Mounting Google Drive to the Colab environment
drive.mount('/content/drive')

# - This code imports the `drive` function from the google.colab library.
# - The `drive.mount()` function is then called with the argument '/content/drive
# - This allows access to files stored in Google Drive within the Colab notebook
```

```
Mounted at /content/drive
```

```
df = pd.read_csv('/content/drive/MyDrive/WA_Fn-UseC_-HR-Employee-Attrition.csv')
df.head()
```

	Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Edu
0	41	Yes	Travel_Rarely	1102	Sales	1	
1	49	No	Travel_Frequently	279	Research & Development	8	
2	37	Yes	Travel_Rarely	1373	Research & Development	2	
3	33	No	Travel_Frequently	1392	Research & Development	3	
4	27	No	Travel_Rarely	591	Research & Development	2	

5 rows × 35 columns

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1470 entries, 0 to 1469
Data columns (total 35 columns):
 #   Column                    Non-Null Count  Dtype
```

```
---  ------                    --------------  -----
 0   Age                       1470 non-null   int64
 1   Attrition                 1470 non-null   object
 2   BusinessTravel            1470 non-null   object
 3   DailyRate                 1470 non-null   int64
 4   Department                1470 non-null   object
 5   DistanceFromHome          1470 non-null   int64
 6   Education                 1470 non-null   int64
 7   EducationField            1470 non-null   object
 8   EmployeeCount             1470 non-null   int64
 9   EmployeeNumber            1470 non-null   int64
 10  EnvironmentSatisfaction   1470 non-null   int64
 11  Gender                    1470 non-null   object
 12  HourlyRate                1470 non-null   int64
 13  JobInvolvement            1470 non-null   int64
 14  JobLevel                  1470 non-null   int64
 15  JobRole                   1470 non-null   object
 16  JobSatisfaction           1470 non-null   int64
 17  MaritalStatus             1470 non-null   object
 18  MonthlyIncome             1470 non-null   int64
 19  MonthlyRate               1470 non-null   int64
 20  NumCompaniesWorked        1470 non-null   int64
 21  Over18                    1470 non-null   object
 22  OverTime                  1470 non-null   object
 23  PercentSalaryHike         1470 non-null   int64
 24  PerformanceRating         1470 non-null   int64
 25  RelationshipSatisfaction  1470 non-null   int64
 26  StandardHours             1470 non-null   int64
 27  StockOptionLevel          1470 non-null   int64
 28  TotalWorkingYears         1470 non-null   int64
 29  TrainingTimesLastYear     1470 non-null   int64
 30  WorkLifeBalance           1470 non-null   int64
 31  YearsAtCompany            1470 non-null   int64
 32  YearsInCurrentRole        1470 non-null   int64
 33  YearsSinceLastPromotion   1470 non-null   int64
 34  YearsWithCurrManager      1470 non-null   int64
dtypes: int64(26), object(9)
memory usage: 402.1+ KB
```

We have done all the preprocessing steps in our Decision Tree class, let's do that again here for this lecture.

Data Preprocessing

```
df.nunique()
```

```
Age                          43
Attrition                     2
BusinessTravel                3
DailyRate                   886
Department                    3
DistanceFromHome             29
Education                     5
EducationField                6
EmployeeCount                 1
EmployeeNumber             1470
```

```
EnvironmentSatisfaction        4
Gender                         2
HourlyRate                    71
JobInvolvement                 4
JobLevel                       5
JobRole                        9
JobSatisfaction                4
MaritalStatus                  3
MonthlyIncome               1349
MonthlyRate                 1427
NumCompaniesWorked            10
Over18                         1
OverTime                       2
PercentSalaryHike             15
PerformanceRating              2
RelationshipSatisfaction       4
StandardHours                  1
StockOptionLevel               4
TotalWorkingYears             40
TrainingTimesLastYear          7
WorkLifeBalance                4
YearsAtCompany                37
YearsInCurrentRole            19
YearsSinceLastPromotion       16
YearsWithCurrManager          18
dtype: int64
```

```
df.columns
```

```
Index(['Age', 'Attrition', 'BusinessTravel', 'DailyRate', 'Department',
       'DistanceFromHome', 'Education', 'EducationField', 'EmployeeCount',
       'EmployeeNumber', 'EnvironmentSatisfaction', 'Gender', 'HourlyRate',
       'JobInvolvement', 'JobLevel', 'JobRole', 'JobSatisfaction',
       'MaritalStatus', 'MonthlyIncome', 'MonthlyRate', 'NumCompaniesWorked',
       'Over18', 'OverTime', 'PercentSalaryHike', 'PerformanceRating',
       'RelationshipSatisfaction', 'StandardHours', 'StockOptionLevel',
       'TotalWorkingYears', 'TrainingTimesLastYear', 'WorkLifeBalance',
       'YearsAtCompany', 'YearsInCurrentRole', 'YearsSinceLastPromotion',
       'YearsWithCurrManager'],
      dtype='object')
```

Identifying Irrelevant Variables:

1. Employee Count: This variable likely contains the same value for all observations, making it irrelevant for analysis as it doesn't provide any variability or useful information.
2. Employee Number: Assuming this is an identification number assigned to each employee, it's not useful for analysis purposes as it doesn't contribute to predicting attrition.
3. Over 18: If this variable indicates whether an employee is over 18 years old, it might not be relevant for predicting attrition unless there's a specific legal requirement related to age.
4. Standard Hours: If this variable represents the standard number of hours worked per week by an employee, it may not vary much across observations and therefore might not be informative for predicting attrition.

Removing Irrelevant Variables

```
df.drop(['EmployeeCount', 'EmployeeNumber', 'Over18', 'StandardHours'], axis = 1,
```

```
df.head()
```

	Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Edu
0	41	Yes	Travel_Rarely	1102	Sales	1	
1	49	No	Travel_Frequently	279	Research & Development	8	
2	37	Yes	Travel_Rarely	1373	Research & Development	2	
3	33	No	Travel_Frequently	1392	Research & Development	3	
4	27	No	Travel_Rarely	591	Research & Development	2	

5 rows × 31 columns

Encoding

- Encoding refers to the process of converting categorical data into a numerical format that can be used for training machine learning models. Categorical data represents attributes that have a fixed number of unique values or categories. Example include variables like Attrition, BusinessTravel, Department, or any other non-numeric data.

```
# Iterate over each column in the DataFrame starting from the second column
for col in df.columns[1:]:
    # Check if the data type of the column is 'object', indicating it contains ca
    if df[col].dtype == 'object':
        # Print the column name and the number of unique values in that column
        print(f"{col}: {df[col].nunique()}")
```

```
Attrition: 2
BusinessTravel: 3
Department: 3
EducationField: 6
Gender: 2
JobRole: 9
MaritalStatus: 3
OverTime: 2
```

Label Encoding for all Variables:

```
# lebel encoding

from sklearn.preprocessing import LabelEncoder
lb = LabelEncoder()

for col in df.columns:
  if(df[col].dtype == 'object'):
      df[col] = lb.fit_transform(df[col])
```

```
df.head()
```

	Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Edu
0	41	1	2	1102	2	1	
1	49	0	1	279	1	8	
2	37	1	2	1373	1	2	
3	33	0	1	1392	1	3	
4	27	0	2	591	1	2	

5 rows × 31 columns

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1470 entries, 0 to 1469
Data columns (total 31 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Age                       1470 non-null   int64
 1   Attrition                 1470 non-null   int64
 2   BusinessTravel            1470 non-null   int64
 3   DailyRate                 1470 non-null   int64
 4   Department                1470 non-null   int64
 5   DistanceFromHome          1470 non-null   int64
 6   Education                 1470 non-null   int64
 7   EducationField            1470 non-null   int64
 8   EnvironmentSatisfaction   1470 non-null   int64
 9   Gender                    1470 non-null   int64
 10  HourlyRate                1470 non-null   int64
 11  JobInvolvement            1470 non-null   int64
 12  JobLevel                  1470 non-null   int64
 13  JobRole                   1470 non-null   int64
 14  JobSatisfaction           1470 non-null   int64
 15  MaritalStatus             1470 non-null   int64
 16  MonthlyIncome             1470 non-null   int64
 17  MonthlyRate               1470 non-null   int64
 18  NumCompaniesWorked        1470 non-null   int64
 19  OverTime                  1470 non-null   int64
 20  PercentSalaryHike         1470 non-null   int64
 21  PerformanceRating         1470 non-null   int64
 22  RelationshipSatisfaction  1470 non-null   int64
 23  StockOptionLevel          1470 non-null   int64
 24  TotalWorkingYears         1470 non-null   int64
 25  TrainingTimesLastYear     1470 non-null   int64
 26  WorkLifeBalance           1470 non-null   int64
 27  YearsAtCompany            1470 non-null   int64
 28  YearsInCurrentRole        1470 non-null   int64
 29  YearsSinceLastPromotion   1470 non-null   int64
 30  YearsWithCurrManager      1470 non-null   int64
dtypes: int64(31)
memory usage: 356.1 KB
```

Model Prepration

```
target = df['Attrition']
X = df.drop(['Attrition'], axis = 1)
```

```
target.value_counts(normalize = True)
```

```
0    0.838776
1    0.161224
Name: Attrition, dtype: float64
```

In the Decision tree lecture we Balanced this with SMOTE techniques, but in this lecture we are not going to balance this because in ensemble we have an option to balance so will use that option to balance dataset.

```
from sklearn.model_selection import train_test_split

# Splitting the dataset into training and testing sets
# X: Features (independent variables)
# target: Target variable (dependent variable)
# test_size: Proportion of the dataset to include in the testing split (20%)
# random_state: Seed for random number generation to ensure reproducibility
# stratify: Ensures that the class distribution of the target variable is preserv
X_train, X_test, y_train, y_test = train_test_split(X, target, test_size=0.2, ran
```

```
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(1176, 30)
(294, 30)
(1176,)
(294,)
```

```
# Import the DecisionTreeClassifier class from the sklearn.tree module
from sklearn.tree import DecisionTreeClassifier

# Create an instance of the DecisionTreeClassifier class
tree_1 = DecisionTreeClassifier(random_state=1)
# Parameters:
# - random_state: Controls the randomness of the estimator. Setting it to a fixed

# Train the Decision Tree classifier using the fit method
tree_1.fit(X_train, y_train)
# Parameters:
# - X_train: The training data features (input variables).
# - y_train: The target variable (labels) corresponding to the training data.
```

```
▾       DecisionTreeClassifier
DecisionTreeClassifier(random_state=1)
```

```
# Import the classification_report function from the sklearn.metrics module
from sklearn.metrics import classification_report

# Use the trained decision tree classifier (tree_1) to make predictions on the te
y_pred = tree_1.predict(X_test)

# Generate a classification report comparing the actual test labels (y_test) with
report = classification_report(y_test, y_pred)

# Print the classification report to the console
print(report)
```

```
              precision    recall  f1-score   support

           0       0.90      0.87      0.88       247
           1       0.41      0.47      0.44        47

    accuracy                           0.81       294
   macro avg       0.65      0.67      0.66       294
weighted avg       0.82      0.81      0.81       294
```

```
# Use the trained decision tree classifier (tree_1) to make predictions on the tr
y_pred_train = tree_1.predict(X_train)

# Print a header indicating that this is the training report
print("Training Report")

# Generate a classification report comparing the actual training labels (y_train)
report_train = classification_report(y_train, y_pred_train)

# Print the classification report for the training data to the console
print(report_train)
```

```
Training Report
              precision    recall  f1-score   support
```

```
           0       1.00      1.00      1.00       986
           1       1.00      1.00      1.00       190

    accuracy                           1.00      1176
   macro avg       1.00      1.00      1.00      1176
weighted avg       1.00      1.00      1.00      1176
```

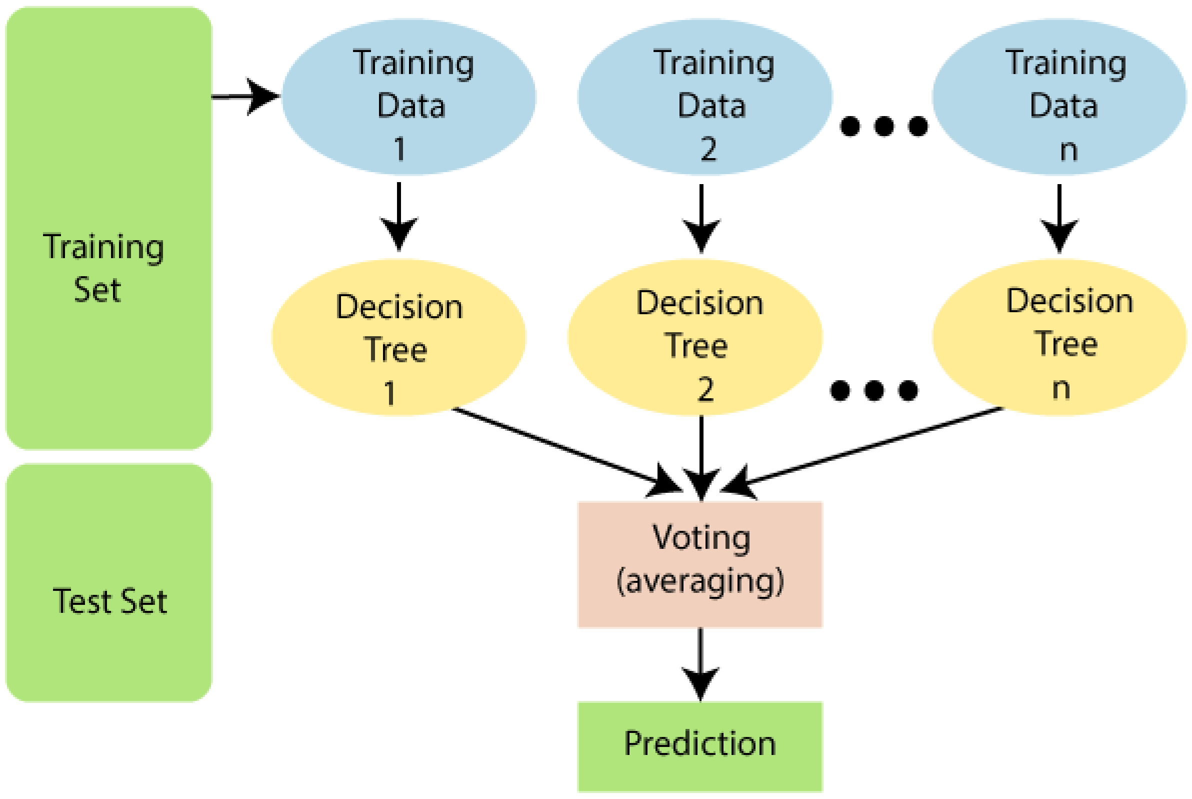
Overfitting Problem in Decision Tree Model

- It appears that the decision tree model has achieved perfect accuracy, precision, recall, and F1-score on both classes (0 and 1) in the training data. While achieving such high performance metrics may seem impressive, it also raises concerns about overfitting.
- Overfitting occurs when a model learns the training data too well, capturing noise and irrelevant patterns that do not generalize to unseen data. In the context of the provided report:
    1. Perfect Scores: The model achieving perfect precision, recall, and F1-score for both classes (0 and 1) suggests that it has perfectly fit the training data. This level of accuracy is often indicative of overfitting, especially if the dataset is not extremely simple or if there is inherent noise in the data.
    2. Imbalance in Class Distribution: The support counts for class 0 (986) and class 1 (190) indicate an imbalance in the class distribution. In cases of class imbalance, models can sometimes achieve high accuracy by simply predicting the majority class, but this may not generalize well to new data.
    3. Lack of Generalization: While the model performs exceptionally well on the training data, it's essential to evaluate its performance on unseen or test data to assess its generalization ability. If the model fails to perform similarly well on new data, it suggests overfitting.
    4. Need for Validation: Without additional information about the model's performance on a separate validation or test dataset, it's challenging to definitively conclude whether overfitting has occurred. Cross-validation or holdout validation techniques should be employed to assess the model's performance on unseen data.

Random Forest

- Random Forest is a popular machine learning algorithm that belongs to the supervised learning technique. It can be used for both Classification and Regression problems in ML. It is based on the concept of ensemble learning, which is a process of combining multiple classifiers to solve a complex problem and to improve the performance of the model.

- As the name suggests, "**Random Forest is a classifier that contains a number of decision trees on various subsets of the given dataset and takes the average to improve the predictive accuracy of that dataset.**" Instead of relying on one decision tree, the random forest takes the prediction from each tree and based on the majority votes of predictions, and it predicts the final output.
- **The greater number of trees in the forest leads to higher accuracy and prevents the problem of overfitting.**



Assumptions for Random Forest

The efficacy of a Random Forest classifier hinges upon two key assumptions:

1. Presence of Actual Values in Feature Variables: The classifier's accuracy relies on the availability of genuine, non-randomized data within the feature variables. This ensures that predictions are based on meaningful patterns rather than arbitrary guesses or noise.
2. Low Correlation Among Predictions: For optimal performance, the predictions generated by individual decision trees within the ensemble should exhibit minimal correlation with each other. This diversity in predictions allows for robust aggregation, wherein the collective output of the ensemble outperforms any single decision tree by capturing a broader range of patterns and nuances within the dataset.

Why use Random Forest?

The Random Forest algorithm offers several compelling reasons for its adoption:

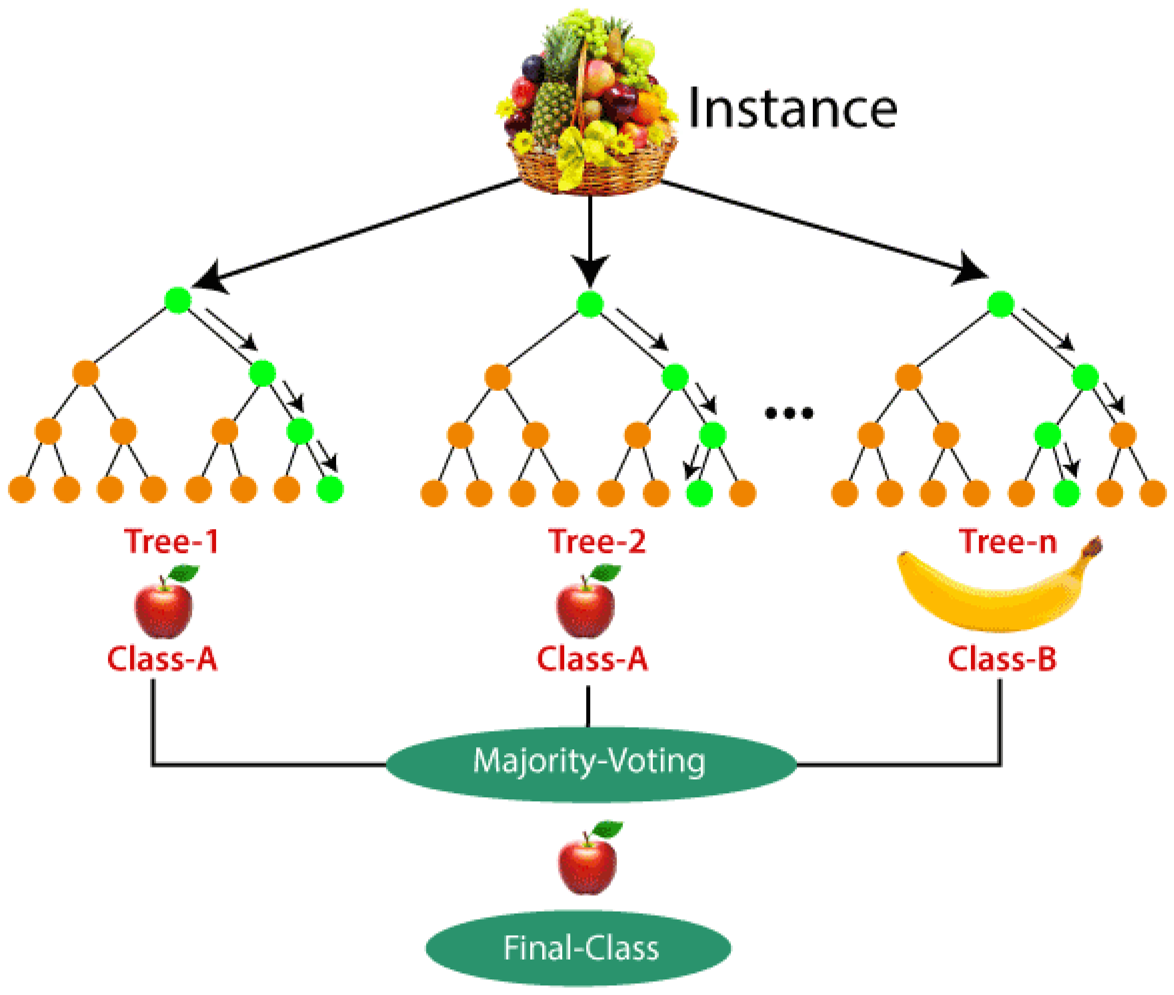
1. Efficient Training Time: Compared to alternative algorithms, Random Forest typically requires less time for training. Its parallelizable nature enables it to efficiently handle large datasets, making it particularly suitable for scenarios where time constraints are a concern.
2. High Accuracy in Predictions: Random Forest excels in delivering accurate predictions across diverse datasets, including those with substantial volumes of data. Its ensemble-based approach leverages the collective wisdom of multiple decision trees, mitigating overfitting and enhancing generalization capability, thereby yielding reliable and precise predictions.
3. Robustness to Missing Data: Random Forest exhibits resilience to missing data, enabling it to maintain predictive accuracy even when a significant portion of the dataset is incomplete. Through its mechanism of aggregating predictions from multiple trees, it can effectively handle missing values without compromising performance.

How does Random Forest algorithm work?

1. Random Forest Construction:
   - Step 1: Randomly select a subset of K data points from the training dataset.
   - Step 2: Construct decision trees using the selected subset of data points. Each decision tree is built independently, employing different subsets of data points and potentially different features at each split.
   - Step 3: Specify the desired number, N, of decision trees to be constructed.
   - Step 4: Repeat Steps 1 and 2 to create N decision trees, each trained on a unique subset of the training data.
2. Prediction Phase:
   - For a new data point, pass it through each decision tree in the forest.
   - Aggregate the predictions from all decision trees.
   - Assign the new data point to the category/class that receives the majority of votes among all decision trees (often referred to as "majority voting").

The working of the algorithm can be better understood by the below example:

Example: Suppose there is a dataset that contains multiple fruit images. So, this dataset is given to the Random forest classifier. The dataset is divided into subsets and given to each decision tree. During the training phase, each decision tree produces a prediction result, and when a new data point occurs, then based on the majority of results, the Random Forest classifier predicts the final decision. Consider the below image:



Define Function for Calculating Classification Model Performance

```
# Import necessary libraries
from sklearn.metrics import accuracy_score, classification_report
import pandas as pd

# Define a function for evaluating classification model performance
def classification_performance(model, features, target, dataset_name_string):
    # Print a header indicating the dataset for which performance metrics are bei
    print(f"{dataset_name_string} Data Performance")
    print()

    # Make predictions using the provided model and input features
    predicted_target = model.predict(features)

    # Generate a classification report as a DataFrame
    report = pd.DataFrame(classification_report(target, predicted_target, output_

    # Print the classification report
    print(report)
    print()

    # Calculate and print the accuracy score
    print("Accuracy Score: ", accuracy_score(target, predicted_target) * 100)
```

Define Function for Calculation KFold Cross Validation Score

```
# Import necessary libraries
from sklearn.model_selection import KFold, cross_val_score

# Define a function for performing k-fold cross-validation and reporting scores
def kFold_cross_validation_score(model, features, target):
    # Initialize KFold with 10 splits
    kfold = KFold(n_splits=10)

    # Perform k-fold cross-validation and obtain accuracy scores
    result = cross_val_score(model, features, target, cv=kfold, scoring='accuracy

    # Calculate and print the mean accuracy score across all folds
    print("K-Fold Accuracy Mean: ", round(result.mean() * 100, 2))

    # Calculate and print the standard deviation of accuracy scores across all fo
    print("K-Fold Accuracy Standard Deviation: ", round(result.std() * 100, 2))
```

```
# Call the classification_performance function to evaluate the performance of the
classification_performance(tree_1, X_train, y_train, 'Training')
```

```
Training Data Performance

               0      1  accuracy  macro avg  weighted avg
precision    1.0    1.0       1.0        1.0           1.0
recall       1.0    1.0       1.0        1.0           1.0
f1-score     1.0    1.0       1.0        1.0           1.0
support    986.0  190.0       1.0     1176.0        1176.0

Accuracy Score:  100.0
```

Overfitting Problem again in this aslo.

```
# Call the classification_performance function to evaluate the performance of the
classification_performance(tree_1, X_test, y_test, "Testing")
```

```
Testing Data Performance

                    0          1  accuracy   macro avg  weighted avg
precision    0.895833   0.407407  0.806122    0.651620      0.817752
recall       0.870445   0.468085  0.806122    0.669265      0.806122
f1-score     0.882957   0.435644  0.806122    0.659300      0.811448
support    247.000000  47.000000  0.806122  294.000000    294.000000

Accuracy Score:  80.61224489795919
```

Implementation of Bagging Classifier

```
# Import necessary libraries
from sklearn.ensemble import RandomForestClassifier, BaggingClassifier

# Import RandomForestClassifier and BaggingClassifier classes from the sklearn.en

# RandomForestClassifier: A classifier that fits multiple decision tree classifie
# sub-samples of the dataset and uses averaging to improve the predictive accurac

# BaggingClassifier: A meta-estimator that fits base classifiers each on random s
# original dataset and then aggregates their individual predictions to form a fin


# Create a BaggingClassifier instance
bag_classifier = BaggingClassifier(
    estimator=DecisionTreeClassifier(),  # Base estimator to be used for bagging,
    n_estimators=25,  # Number of base estimators (decision trees) to be created
    random_state=7,  # Controls the random seed for reproducibility
    oob_score=True  # Whether to use out-of-bag samples to estimate the generaliz
)


# Train the BaggingClassifier model using the fit method
bag_model = bag_classifier.fit(X_train, y_train)


# accuracy of score
# Access the out-of-bag (OOB) score of the trained BaggingClassifier model
bag_model.oob_score_
```

```
0.8469387755102041
```

```
# Call the classification_performance function to evaluate the performance of the
classification_performance(bag_model, X_train, y_train, 'Training')
```

```
Training Data Performance

                    0           1  accuracy    macro avg  weighted avg
precision    0.998987    1.000000   0.99915     0.999493      0.999151
recall       1.000000    0.994737   0.99915     0.997368      0.999150
f1-score     0.999493    0.997361   0.99915     0.998427      0.999149
support    986.000000  190.000000   0.99915  1176.000000   1176.000000

Accuracy Score:  99.91496598639455
```

```
# Call the classification_performance function to evaluate the performance of the
classification_performance(bag_model, X_test, y_test, "Testing")
```

```
Testing Data Performance

                    0          1  accuracy   macro avg  weighted avg
precision    0.869565   0.611111  0.853741    0.740338      0.828248
recall       0.971660   0.234043  0.853741    0.602851      0.853741
f1-score     0.917782   0.338462  0.853741    0.628122      0.825170
support    247.000000  47.000000  0.853741  294.000000    294.000000

Accuracy Score:  85.37414965986395
```

Implementation of Random Forest

```
# Call the kFold_cross_validation_score function to perform k-fold cross-validati
kFold_cross_validation_score(bag_classifier, X, target)
```

```
K-Fold Accuracy Mean:  85.1
K-Fold Accuracy Standard Deviation:  2.22
```

```
# Import the RandomForestClassifier class from the sklearn.ensemble module
from sklearn.ensemble import RandomForestClassifier

# Create a RandomForestClassifier instance with specified hyperparameters
rf_classifier = RandomForestClassifier(
    n_estimators=25,  # Number of decision trees in the forest
    max_features=6,  # Maximum number of features considered for splitting
    criterion='gini',  # The function to measure the quality of a split (Gini imp
    max_depth=10,  # Maximum depth of the decision trees
    min_impurity_decrease=0.05,  # Minimum impurity decrease required for a split
    bootstrap=True,  # Whether bootstrap samples are used when building trees
    oob_score=True,  # Whether to use out-of-bag samples to estimate generalizati
    class_weight='balanced',  # Weights associated with classes to handle class i
    random_state=7  # Controls the random seed for reproducibility
)
```

```
# Train the RandomForestClassifier model using the fit method
rf_model = rf_classifier.fit(X_train, y_train)
```

```
# Call the classification_performance function to evaluate the performance of the
classification_performance(rf_model, X_train, y_train, 'Training')
```

```
Training Data Performance

                    0           1  accuracy    macro avg  weighted avg
precision    0.909091    0.320442  0.727891     0.614766      0.813986
recall       0.750507    0.610526  0.727891     0.680517      0.727891
f1-score     0.822222    0.420290  0.727891     0.621256      0.757284
support    986.000000  190.000000  0.727891  1176.000000   1176.000000

Accuracy Score:  72.78911564625851
```

```
# Call the classification_performance function to evaluate the performance of the
classification_performance(rf_model, X_test, y_test, "Testing")
```

```
Testing Data Performance

                    0          1  accuracy   macro avg  weighted avg
precision    0.888372   0.291139  0.727891    0.589756      0.792896
recall       0.773279   0.489362  0.727891    0.631321      0.727891
f1-score     0.826840   0.365079  0.727891    0.595960      0.753021
support    247.000000  47.000000  0.727891  294.000000    294.000000
```

```
Accuracy Score:  72.78911564625851
```

- Equal accuracy observed between training and test datasets indicates effective mitigation of overfitting in the RandomForestClassifier model, highlighting its robust generalization capability.
- This parity underscores the model's reliability and suitability for real-world applications by achieving a balance between capturing underlying patterns and avoiding the pitfalls of over-reliance on training data characteristics.

Adjusting min_impurity_decrease from 0.05 to 0.01 for getting higher f1 score.

```
# Import the RandomForestClassifier class from the sklearn.ensemble module
from sklearn.ensemble import RandomForestClassifier

# Create a RandomForestClassifier instance with specified hyperparameters
rf_classifier = RandomForestClassifier(
    n_estimators=25,  # Number of decision trees in the forest
    max_features=6,  # Maximum number of features considered for splitting
    criterion='gini',  # The function to measure the quality of a split (Gini imp
    max_depth=10,  # Maximum depth of the decision trees
    min_impurity_decrease=0.01,  # Minimum impurity decrease required for a split
    bootstrap=True,  # Whether bootstrap samples are used when building trees
    oob_score=True,  # Whether to use out-of-bag samples to estimate generalizati
    class_weight='balanced',  # Weights associated with classes to handle class i
    random_state=7  # Controls the random seed for reproducibility
)


# Train the RandomForestClassifier model using the fit method
rf_model = rf_classifier.fit(X_train, y_train)


# Call the classification_performance function to evaluate the performance of the
classification_performance(rf_model, X_train, y_train, 'Training')
```

```
Training Data Performance

                    0           1  accuracy    macro avg  weighted avg
precision    0.937642    0.459184  0.818027     0.698413      0.860340
recall       0.838742    0.710526  0.818027     0.774634      0.818027
f1-score     0.885439    0.557851  0.818027     0.721645      0.832512
support    986.000000  190.000000  0.818027  1176.000000   1176.000000

Accuracy Score:  81.80272108843538
```

```
# Call the classification_performance function to evaluate the performance of the
classification_performance(rf_model, X_test, y_test, "Testing")
```

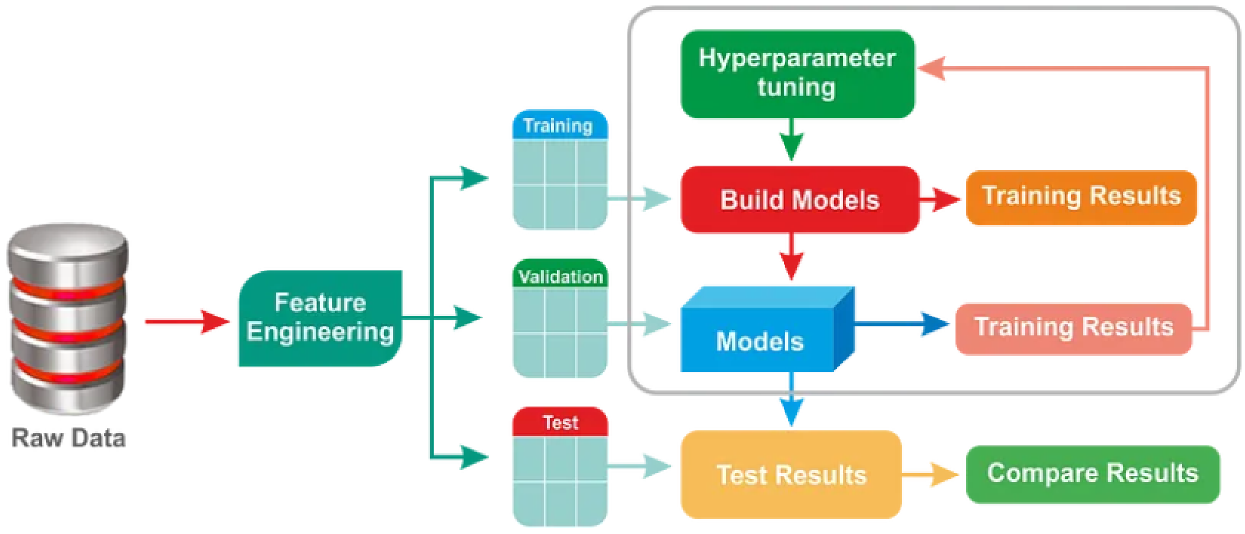
```
Testing Data Performance

                    0          1  accuracy   macro avg  weighted avg
```

```
precision    0.910314   0.380282  0.782313    0.645298      0.825581
recall       0.821862   0.574468  0.782313    0.698165      0.782313
f1-score     0.863830   0.457627  0.782313    0.660728      0.798893
support    247.000000  47.000000  0.782313  294.000000    294.000000

Accuracy Score:  78.2312925170068
```

After adjusting min_impurity_decrease 0.05 to 0.01, f1 score got increased, we can see in above results.

Hyperparameter Tuning

- Hyperparameter tuning is a crucial aspect of machine learning model development, where you optimize the parameters that govern the learning process itself, as opposed to the parameters learned from the data.
- These hyperparameters influence the behavior of the training algorithm and, consequently, the performance of the model.



Here are detailed notes on hyperparameter tuning:

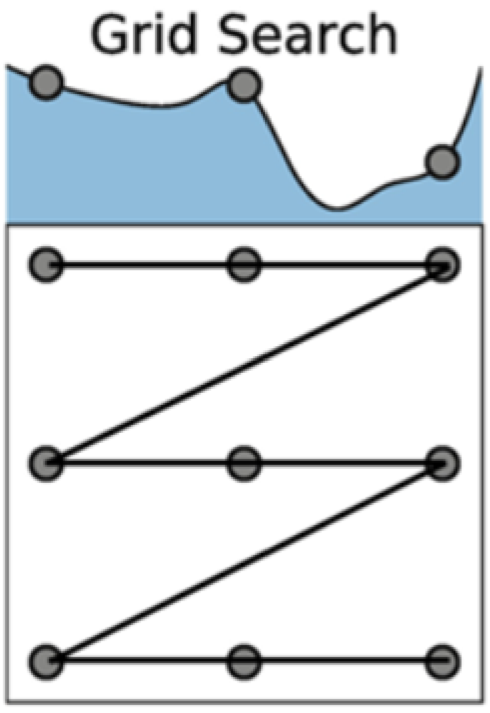
1. Understanding Hyperparameters:
   - Hyperparameters are settings or configurations for a machine learning algorithm that are set before the learning process begins.
   - They control the learning process's behavior and directly impact the performance of the model.
   - Examples include learning rate, regularization strength, number of hidden units in a neural network, depth of a decision tree, etc.
2. Importance of Hyperparameter Tuning:
   - Proper tuning of hyperparameters can significantly enhance model performance.

  - The default hyperparameters may not be optimal for all datasets or tasks.
  - Hyperparameter tuning aims to find the best combination of hyperparameters that optimize the model's performance on a specific dataset.

Grid Search Cross Validation (CV):

Introduction to Grid Search:

- Grid search is a hyperparameter optimization technique used to find the best combination of hyperparameters for a machine learning model.
- It involves creating a grid of hyperparameters, where each axis of the grid represents a hyperparameter, and the points in the grid represent specific parameter values.
- Grid search exhaustively searches through all possible combinations of hyperparameters within the defined ranges.



Evaluation Using Cross-Validation:

- Each combination of hyperparameters is evaluated using cross-validation to estimate the model's performance.
- Cross-validation, typically k-fold cross-validation, divides the dataset into k subsets (folds), trains the model on k-1 folds, and validates it on the remaining fold.
- This process is repeated k times, with each fold serving as the validation set exactly once. The average performance across all folds is used as the evaluation metric for the hyperparameter combination.

Computational Expense:

- Grid search tests every possible combination of hyperparameters, making it computationally expensive, especially for models with a large number of hyperparameters and a wide range of values for each.
- The exhaustive nature of grid search ensures that it explores the entire search space and finds the combination of hyperparameters that optimizes the chosen scoring metric.

Scoring Metric and Selection of Best Parameters:

- The scoring metric used to evaluate each combination of hyperparameters depends on the problem at hand.
- Common metrics include accuracy, precision, recall, F1-score, or others depending on the nature of the classification or regression task.
- After evaluating all combinations, the combination of hyperparameters that achieves the highest score on the validation metric is selected as the best set of parameters for the model.

```
# Import the GridSearchCV class from the sklearn.model_selection module
from sklearn.model_selection import GridSearchCV

# GridSearchCV is a method to perform hyperparameter tuning by exhaustively searc


# Define a dictionary containing the hyperparameters and their corresponding valu
params = {
    "n_estimators": [25, 50, 5, 100],  # Number of decision trees in the forest
    "criterion": ["entropy", "gini"],  # The function to measure the quality of a
    "max_depth": [5, 10, 12, 15],  # Maximum depth of the decision trees
    "max_features": [5, 6, 7, 8]  # Maximum number of features considered for spl
}


# Create a GridSearchCV instance
grid_model = GridSearchCV(
    estimator=RandomForestClassifier(),  # The base estimator to be tuned, in thi
    param_grid=params,  # The parameter grid containing hyperparameter values to
    scoring='f1',  # The scoring metric to optimize for during hyperparameter tun
    cv=10,  # Number of folds for cross-validation
    verbose=3,  # Controls the verbosity of the grid search process
    n_jobs=-1  # Number of jobs to run in parallel (-1 means using all available
)


# Fit the GridSearchCV instance to the training data
grid_model.fit(X_train, y_train)
```

```
Fitting 10 folds for each of 128 candidates, totalling 1280 fits
▸ GridSearchCV
▸ estimator: RandomForestClassifier
    ▸ RandomForestClassifier
```

```
# Create a GridSearchCV instance with specified parameters
GridSearchCV(
    cv=10,  # Number of folds for cross-validation
    estimator=RandomForestClassifier(),  # The base estimator to be tuned, in thi
    n_jobs=-1,  # Number of jobs to run in parallel (-1 means using all available
    param_grid={  # The parameter grid containing hyperparameter values to be sea
        'criterion': ['entropy', 'gini'],  # The function to measure the quality
        'max_depth': [5, 10, 12, 15],  # Maximum depth of the decision trees
        'max_features': [5, 6, 7, 8],  # Maximum number of features considered fo
        'n_estimators': [25, 50, 75, 100]  # Number of decision trees in the fore
    },
    scoring='f1',  # The scoring metric to optimize for during hyperparameter tun
    verbose=3  # Controls the verbosity of the grid search process
)
```

```
▸ GridSearchCV
▸ estimator: RandomForestClassifier
    ▸ RandomForestClassifier
```

```
# Access the best estimator found by GridSearchCV
best_rf_estimator = grid_model.best_estimator_
best_rf_estimator
```

```
▾ RandomForestClassifier
RandomForestClassifier(max_depth=15, max_features=6, n_estimators=5)
```

```
# Print information about the best estimator found by GridSearchCV
print("Best Estimator: \n{}\n".format(grid_model.best_estimator_))

# Print the best parameters found by GridSearchCV
print("Best Parameter: \n{}\n".format(grid_model.best_params_))

# Print the best test score found by GridSearchCV
print("Best Test Score: \n{}\n".format(grid_model.best_score_))

# Print the mean test scores for all parameter combinations
print("Best Test Scores: \n{}\n".format(grid_model.cv_results_['mean_test_score']
```

```
Best Estimator:
RandomForestClassifier(max_depth=15, max_features=6, n_estimators=5)

Best Parameter:
```

```
{'criterion': 'gini', 'max_depth': 15, 'max_features': 6, 'n_estimators': 5}

Best Test Score:
0.39846572740899766

Best Test Scores:
[0.15761284 0.15377094 0.17860943 0.15896104 0.20387859 0.17713156
 0.24549199 0.17425184 0.17032844 0.18619084 0.21540639 0.17045662
 0.16019386 0.21349991 0.25767707 0.25051687 0.2529192  0.26883327
 0.27963033 0.24861547 0.2603608  0.29563315 0.28186321 0.28856727
 0.32608037 0.32725789 0.34389671 0.28475438 0.27952947 0.27439925
 0.29048124 0.30672072 0.30346095 0.26135774 0.31587676 0.2745854
 0.30422219 0.28137903 0.28669722 0.26044634 0.32500048 0.26341259
 0.33722306 0.2885236  0.29929418 0.31317793 0.31201127 0.29522711
 0.30014279 0.25571385 0.26126823 0.26633403 0.28536726 0.25281533
 0.32119928 0.25173667 0.28533374 0.27906038 0.30774397 0.26151507
 0.34389931 0.29669982 0.34212953 0.30233383 0.15115884 0.14297045
 0.25973796 0.15328477 0.19528515 0.16169866 0.24175229 0.16153397
 0.18724355 0.17495975 0.24666103 0.21789385 0.2219697  0.23465608
 0.29273162 0.20753884 0.28620252 0.30077434 0.26621322 0.25456092
 0.27632725 0.26123105 0.3502906  0.30927949 0.30398578 0.31316679
 0.30787711 0.27692641 0.29964083 0.32681327 0.35173815 0.25655807
 0.29568354 0.2907444  0.2873247  0.28580972 0.30799767 0.31567668
 0.26077747 0.27500282 0.30186319 0.25171398 0.37441011 0.28843168
 0.31660189 0.32692181 0.33803571 0.31809581 0.31953972 0.27851999
 0.30352638 0.2819204  0.2790836  0.31354385 0.39846573 0.27640031
 0.31425266 0.33537294 0.31529048 0.29391993 0.32441021 0.26403592
 0.33279293 0.28527877]
```

```
# Define a RandomForestClassifier with hyperparameters set to the best values fou
rf_classifier_with_gridsearch_CV = RandomForestClassifier(
    n_estimators=5,  # Number of decision trees in the forest
    criterion='gini',  # The function to measure the quality of a split
    max_depth=15,  # Maximum depth of the decision trees
    max_features=8,  # Maximum number of features considered for splitting
    oob_score=True,  # Whether to use out-of-bag samples to estimate generalizati
    bootstrap=True,  # Whether bootstrap samples are used when building trees
    random_state=7  # Controls the random seed for reproducibility
)
```

```
# Train the RandomForestClassifier model with the hyperparameters obtained from G
rf_model_with_gridsearch_CV = rf_classifier_with_gridsearch_CV.fit(X_train, y_tra
```

```
# Evaluate the performance of the RandomForestClassifier model with GridSearchCV-
classification_performance(rf_model_with_gridsearch_CV, X_train, y_train, 'Traini
```

```
Training Data Performance

                    0           1  accuracy    macro avg  weighted avg
precision    0.975198    0.982143   0.97619     0.978671       0.97632
recall       0.996957    0.868421   0.97619     0.932689       0.97619
f1-score     0.985958    0.921788   0.97619     0.953873       0.97559
support    986.000000  190.000000   0.97619  1176.000000    1176.00000

Accuracy Score:  97.61904761904762
```

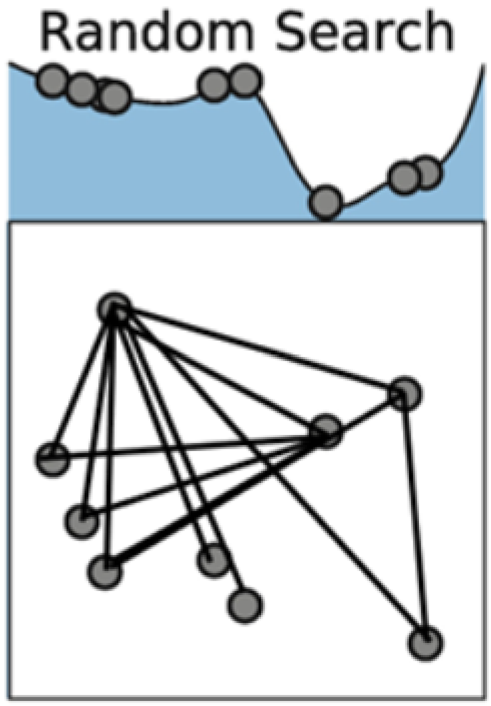
```
# Evaluate the performance of the RandomForestClassifier model with GridSearchCV-
classification_performance(rf_model_with_gridsearch_CV, X_test, y_test, "Testing"
```

```
Testing Data Performance

                     0          1  accuracy   macro avg  weighted avg
precision    0.866906   0.625000  0.853741    0.745953      0.828234
recall       0.975709   0.212766  0.853741    0.594237      0.853741
f1-score     0.918095   0.317460  0.853741    0.617778      0.822075
support    247.000000  47.000000  0.853741  294.000000    294.000000

Accuracy Score:  85.37414965986395
```

Random Search Cross Validation (CV)

Introduction to Random Search:

- Random search is another hyperparameter optimization technique used to find the best combination of hyperparameters for a machine learning model.
- Unlike grid search, which exhaustively searches through all combinations of hyperparameters, random search randomly samples combinations from predefined distributions of hyperparameters.



Random Sampling from Parameter Space:

- In random search, hyperparameters are not explored in a systematic grid-like manner. Instead, combinations are randomly selected from the parameter space.

- The user specifies the distributions or ranges for each hyperparameter, and random search samples from these distributions or ranges to generate combinations.

Efficiency and Speed:

- Random search is often more efficient and faster than grid search because it doesn't evaluate every possible combination of hyperparameters.
- By randomly sampling from the parameter space, random search can cover a wider range of hyperparameter combinations with fewer evaluations.

Utilization of Cross-Validation:

- Similar to grid search, random search still leverages cross-validation to evaluate the performance of each combination of hyperparameters.
- Cross-validation helps estimate the generalization performance of the model and prevents overfitting by providing an unbiased evaluation metric.

Lack of Exhaustiveness:

- Unlike grid search, random search may not guarantee finding the absolute best combination of hyperparameters.
- However, it often yields good results in practice, especially when the search space is large and exhaustive search is not feasible due to computational constraints.

```
# Import the RandomizedSearchCV class from the sklearn.model_selection module
from sklearn.model_selection import RandomizedSearchCV


# Create a RandomizedSearchCV instance
random_model = RandomizedSearchCV(
    estimator=RandomForestClassifier(),  # The base estimator to be tuned, in thi
    scoring='f1',  # The scoring metric to optimize for during hyperparameter tun
    param_distributions=params,  # The parameter distributions for sampling hyper
    cv=5,  # Number of folds for cross-validation
    verbose=1,  # Controls the verbosity of the search process
    n_jobs=-1  # Number of jobs to run in parallel (-1 means using all available
)


# Fit the RandomizedSearchCV instance to the training data
random_model.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
▸ RandomizedSearchCV
▸ estimator: RandomForestClassifier
  ▸ RandomForestClassifier
```

```
# Define a RandomizedSearchCV instance with specified parameters
RandomizedSearchCV(
    cv=5,  # Number of folds for cross-validation
    estimator=RandomForestClassifier(),  # The base estimator to be tuned, in thi
    n_jobs=-1,  # Number of jobs to run in parallel (-1 means using all available
    param_distributions={  # The parameter distributions for sampling hyperparame
        'criterion': ['entropy', 'gini'],  # The function to measure the quality
        'max_depth': [5, 10, 12, 15],  # Maximum depth of the decision trees
        'max_features': [5, 6, 7, 8],  # Maximum number of features considered fo
        'n_estimators': [25, 50, 75, 100]  # Number of decision trees in the fore
    },
    scoring='f1',  # The scoring metric to optimize for during hyperparameter tun
    verbose=3  # Controls the verbosity of the search process
)
```

```
▸ RandomizedSearchCV
▸ estimator: RandomForestClassifier
  ▸ RandomForestClassifier
```

```
# Access the best estimator found by RandomizedSearchCV
random_model.best_estimator_
```

```
▾ RandomForestClassifier
RandomForestClassifier(max_depth=12, max_features=6, n_estimators=5)
```

Pickling

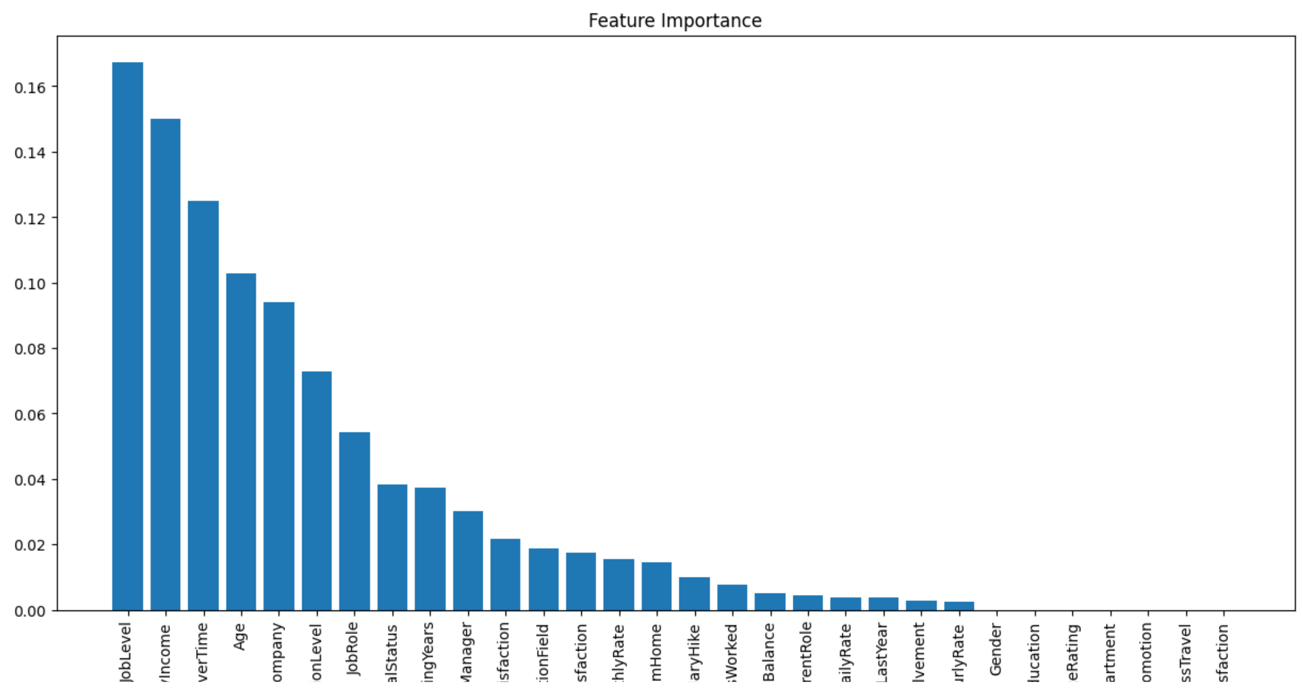
- Pickling in machine learning refers to the process of serializing Python objects into a byte stream, which can then be saved to a file or transmitted over a network.
- This allows you to save trained machine learning models, along with any associated preprocessing steps or parameters, so that they can be reused later without having to retrain the model.

```
# Pickle the trained model for later use
import pickle

# Define the filename for the pickled model
# Load the pickled model from the file and use it to make predictions
loaded_model = pickle.load(open(filename, 'rb'))

# Evaluate the loaded model on the test data
result = loaded_model.score(X_test, y_test)

# Print the evaluation result
print(result)
```

```
0.782312925170068
```

Variable Importance

```
# Calculate feature importances and plot them
importances = rf_model.feature_importances_  # Obtain feature importances from th
indices = np.argsort(importances)[::-1]  # Sort feature importances in descending
names = [X_train.columns[i] for i in indices]  # Get the feature names correspond

# Plot the feature importances
plt.figure(figsize=(15, 7))  # Set the figure size
plt.title("Feature Importance")  # Set the title of the plot
plt.bar(range(X_train.shape[1]), importances[indices])  # Create a bar plot for f
plt.xticks(range(X_train.shape[1]), names, rotation=90)  # Set x-axis ticks to fe
plt.show()  # Display the plot
```


Feature Importance
JobLevel
Income
verTime
Age
ompany
onLevel
JobRole
alStatus
ingYears
Manager
isfaction
ionField
sfaction
hlyRate
mHome
aryHike
Worked
Balance
rentRole
ailyRate
LastYear
lvement
urlyRate
Gender
lucation
eRating
artment
omotion
ssTravel
sfaction