

Chap 06

**STTHK
3013**

SOC-UUM

Ensemble Decision Tree

Azizi Ab Aziz, *PhD*

Human-Centred Computing Lab || School of Computing || UUM

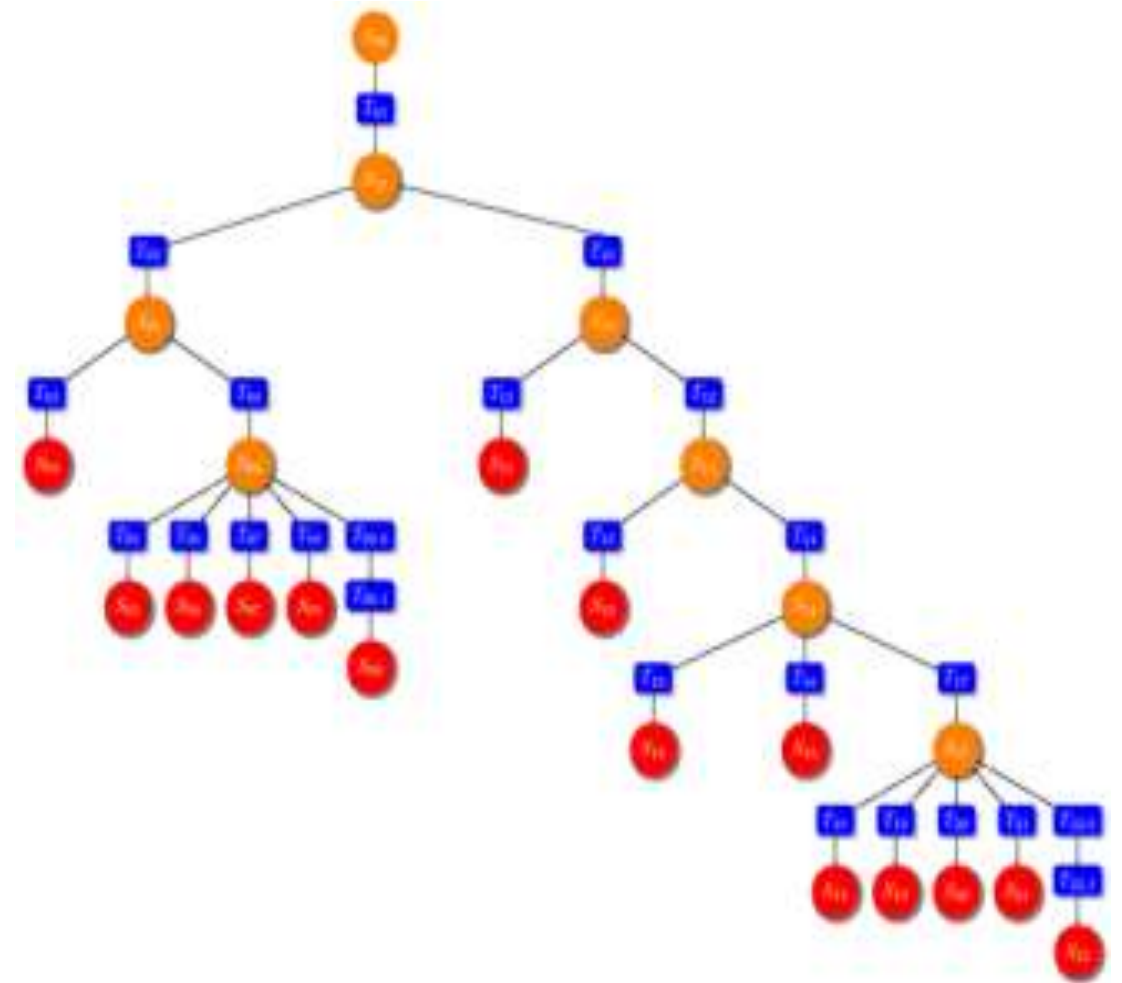
Principle of **Occam's Razor**
“simpler trees are better”

Decision Tree

- One of the most fundamental and versatile algorithms in the realm of pattern recognition methods is the Decision Tree.
- Decision trees are simple to understand, yet powerful tools that can be applied to a wide range of tasks, from classification to regression, and even feature selection.
- It has a tree structure where:
 - an internal node represents a feature(or attribute),
 - the branch represents a decision rule, and each leaf node represents the outcome.
 - The topmost node in a decision tree is known as the root node.
- It learns to partition based on the attribute value.
 - It partitions the tree in a recursive manner called recursive partitioning.

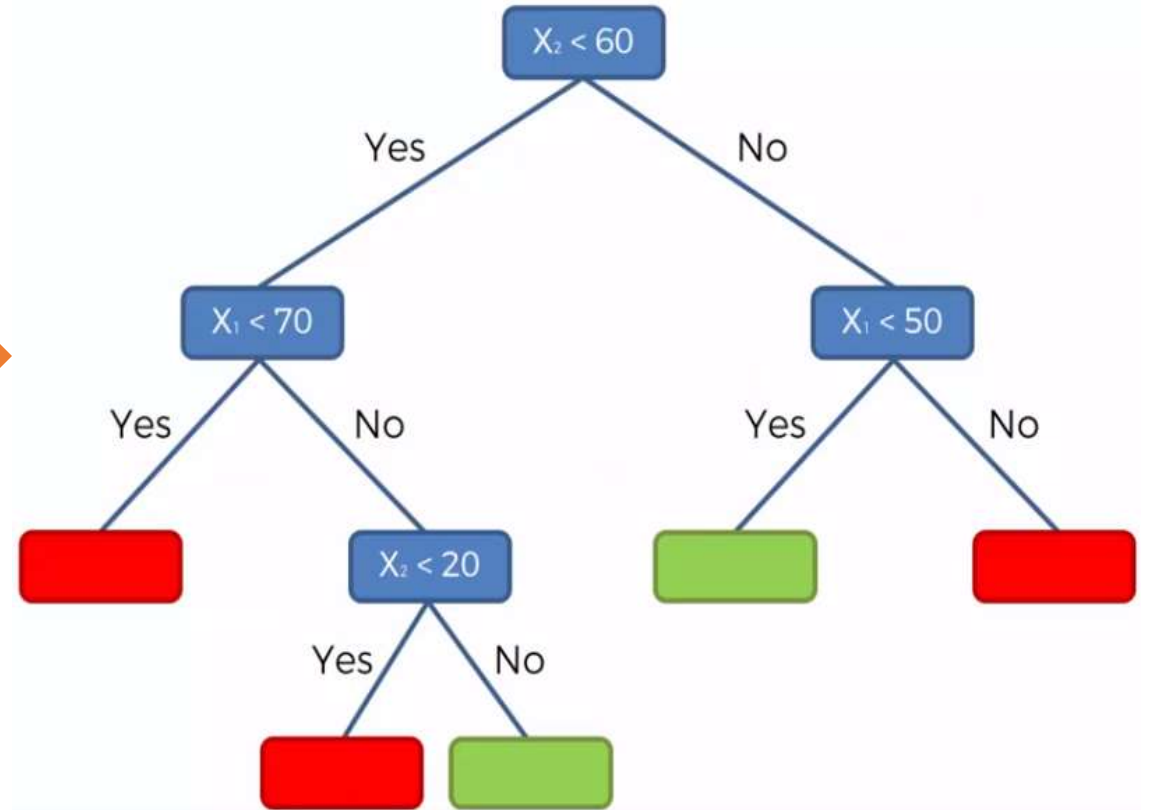
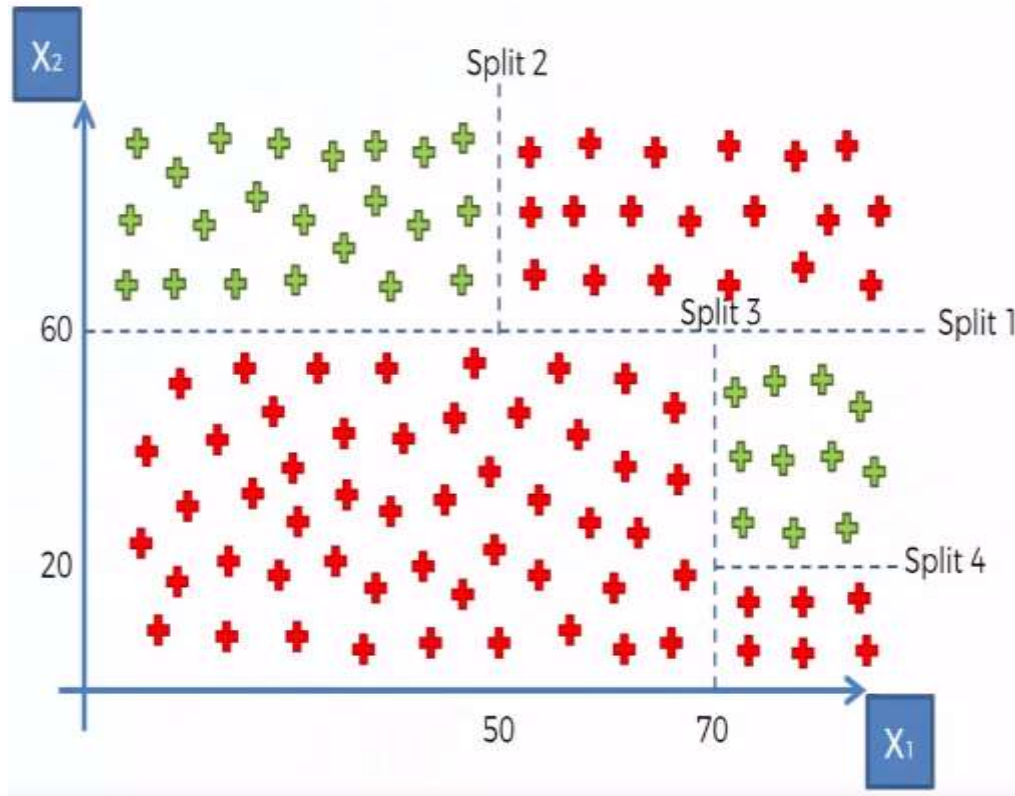
Decision Tree

- Tree-based methods operate by dividing up the feature space into rectangles
- Each rectangle is like a neighbourhood in a Nearest-Neighbours method
- You predict using the average or classify using the most common class in each rectangle
- Allows both prediction (regression tree) and classification (classification tree) problem.



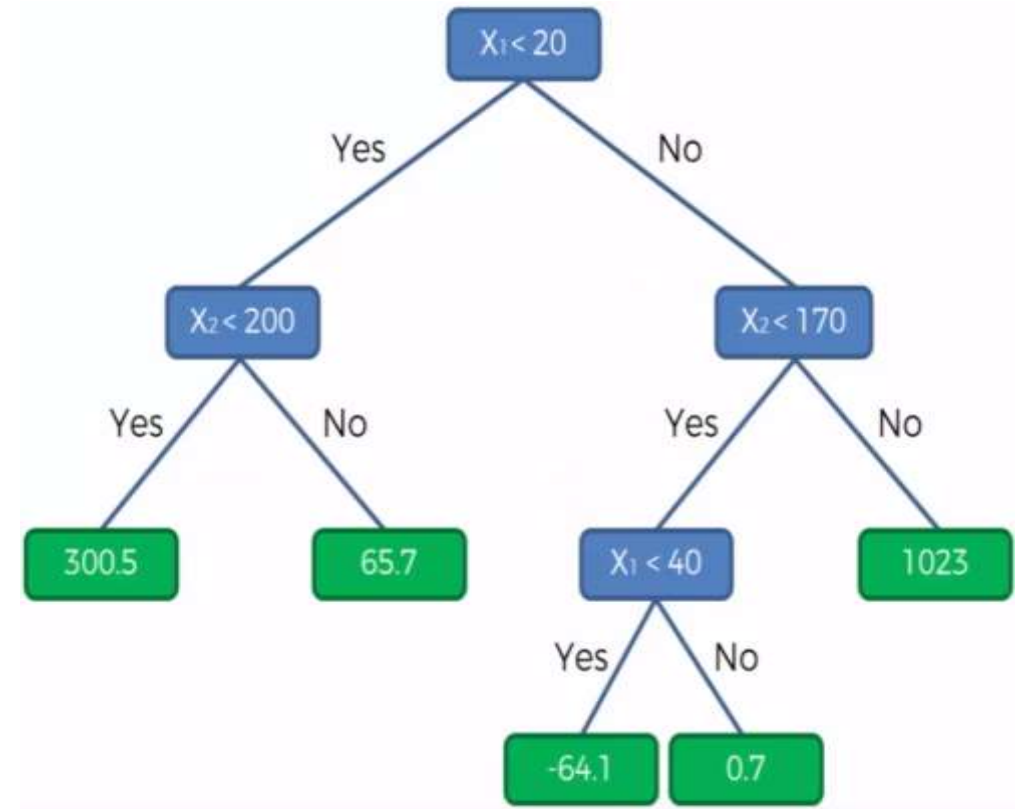
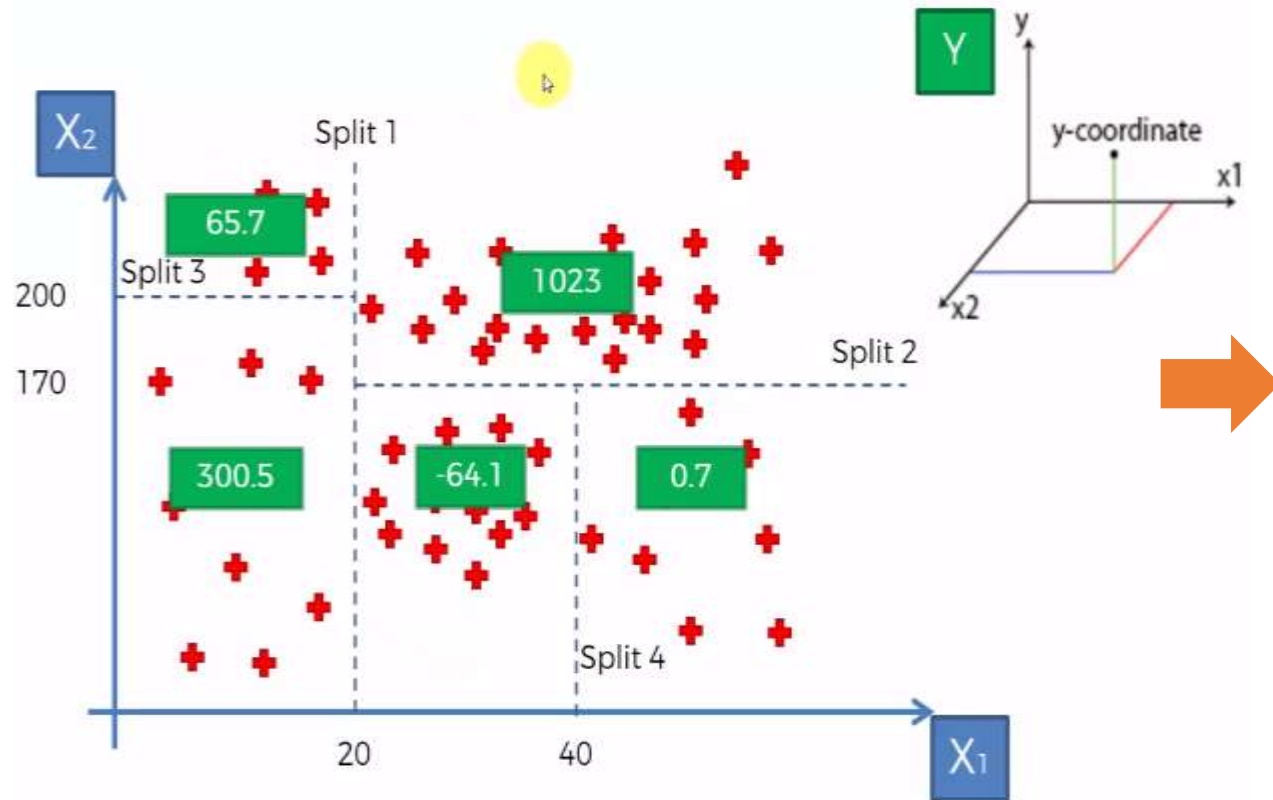
Decision Tree

Classification Tree



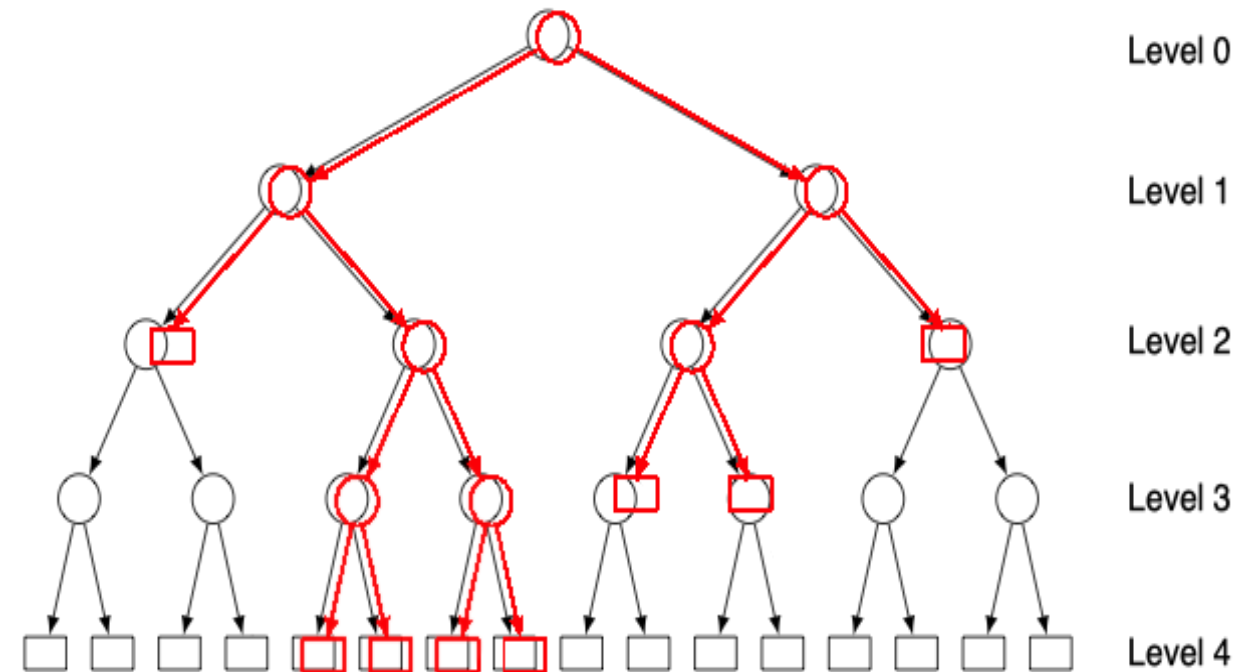
Decision Tree

Regression Tree



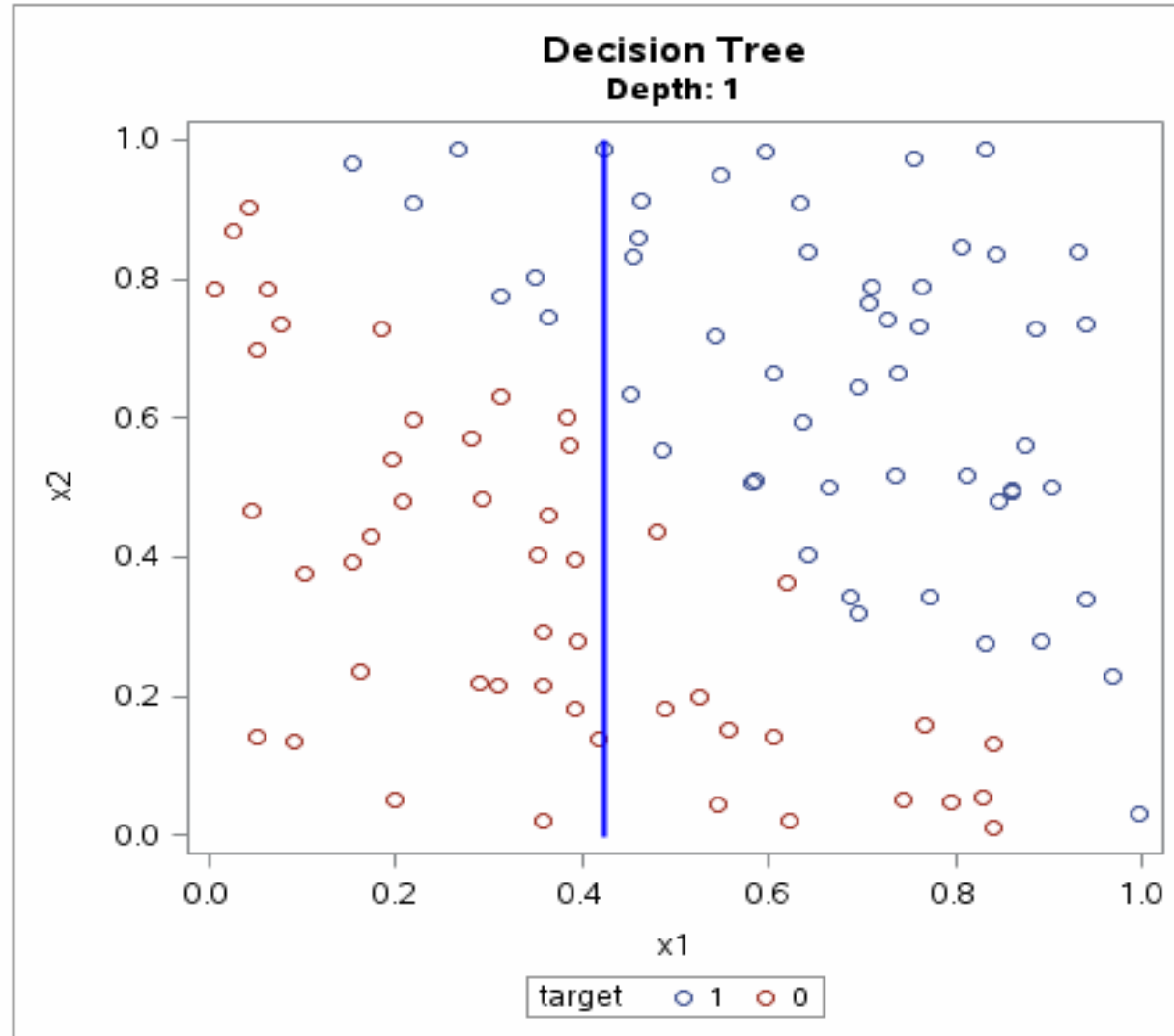
Decision Tree- Main Process

- Use supervised learning ~ samples with tagged labels
- Process:
 - Number of splits (binary vs. multi-way)
 - Query selection
 - Rule for stopping splitting and pruning
 - Rule for labelling the leaves
 - Variable combination and missing data

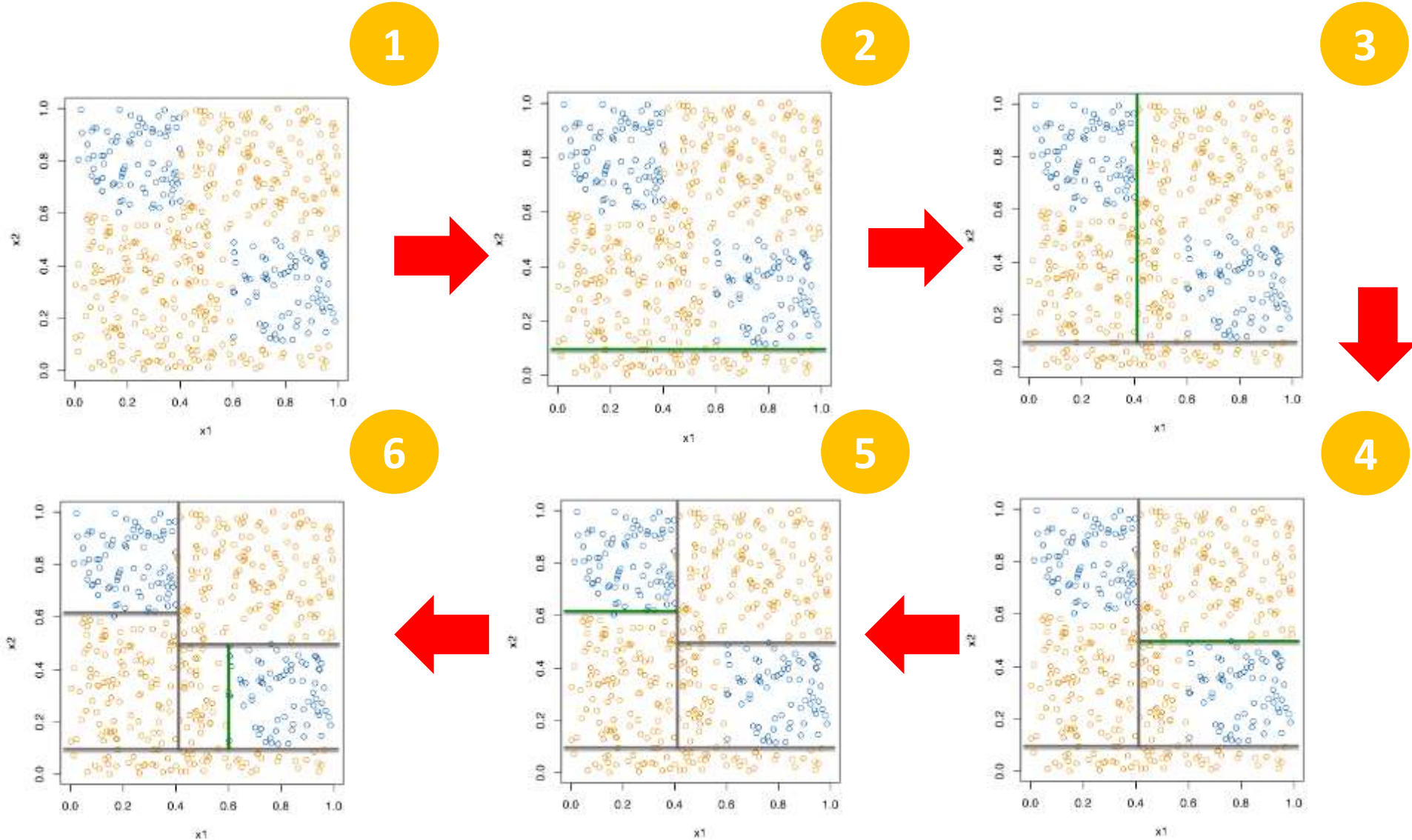


Decision Tree (Decision Space)

09:18 Friday, August 2, 2013 1



Decision Tree (Decision Space)

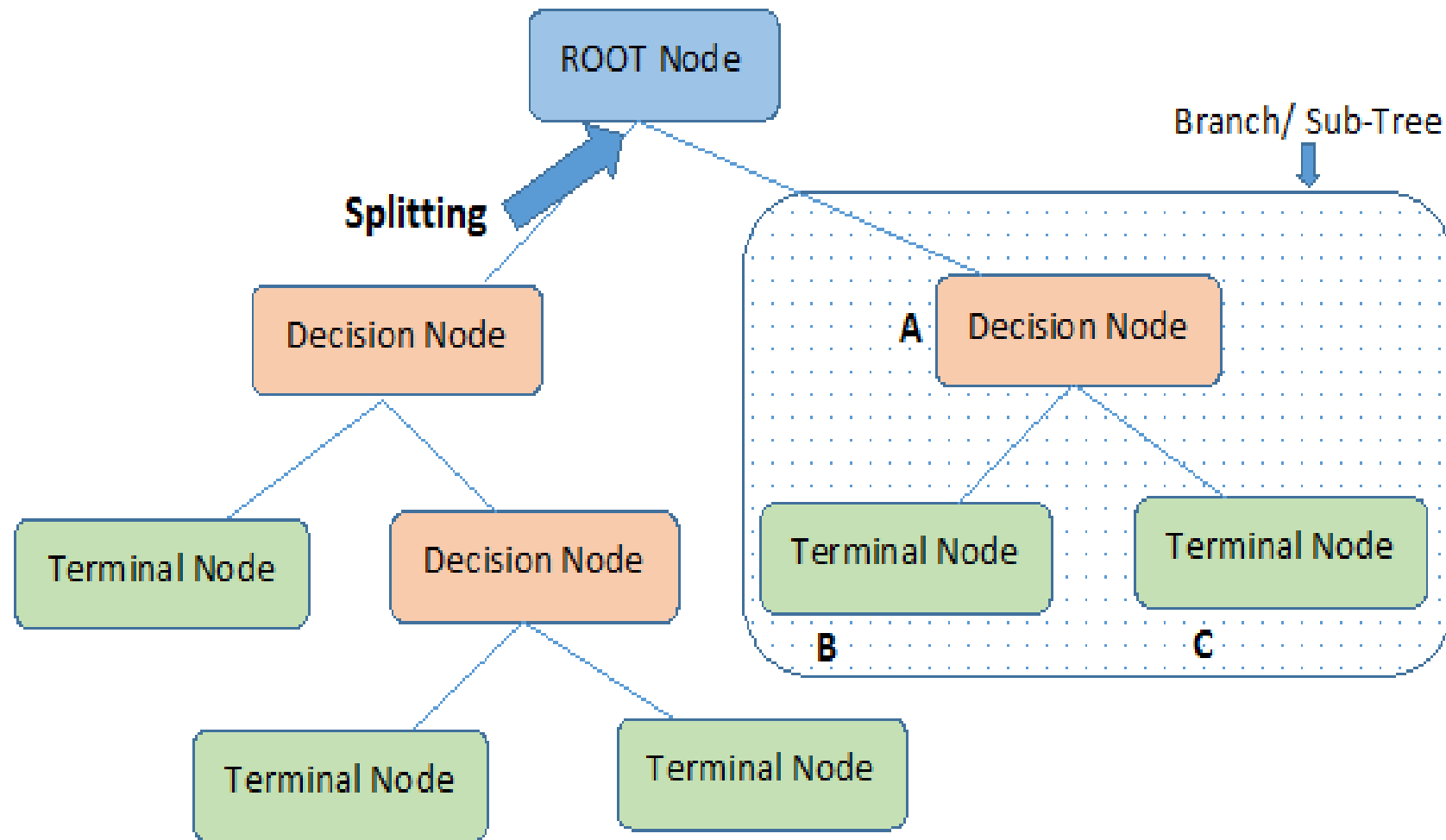


Decision Tree

Major Assumption

- In the beginning, the whole training set is considered as the **root**.
- Feature values are preferred to be categorical.
 - If the values are continuous, they are discretised before building the model.
- Records are **distributed recursively** based on attribute values.
- Order to place attributes as root or internal node of the tree is done by using some statistical approach.
- Main challenge – (**splitting approach**) to identify which attributes need to be considered as the root node and each level.
 - How do Decision Trees select which variable to split on at decision nodes?
 - How does it decide that the tree has enough branches and that it should stop splitting?

Decision Tree



Note:- A is parent node of B and C.

Decision Tree

Node Splitting

- A decision tree makes decisions by splitting nodes into sub-nodes. It is a supervised learning algorithm.
- This process is recursively performed multiple times during the training process until only homogenous nodes are left.
 - Beware, as the process of recursive node splitting into subsets created by each sub-tree can cause overfitting
- The ways of splitting a node can be broadly divided into two categories based on the type of target variable:
 - Continuous target variable: *Reduction In Variance*
 - Categorical target variable: *Gini impurity, Information Gain, Gain Ratio, and Chi-Square*

Decision Tree Algorithms

- **ID3** (*Iterative Dichotomiser 3*) - uses entropy and information for classification problems and standard deviation reduction for regression problems.
- **C4.5** – uses the Gain Ratio approach to generate rules (for classification)
 - It is a successor of ID3 with some improvements on continuous attribute value ranges, pruning of decision trees, and rule derivation.
- **CART** (*Classification and Regression Tree*) - a binary tree based on the Gini index used for both classification and regression.
- **CHAID** (*Chi-square Automatic Interaction Detection*)- uses χ^2 to perform multi-level splits when computing classification trees.

Decision Tree Algorithms

Properties \ Algorithms	ID3	C4.5	CART	CHAID
Types of data	Categorical	Categorical / Continuous	Categorical / Continuous	Categorical
Speed	Fast (but creating deep trees)	Slow compared to ID3	Fast	Fast
Handling missing values	Generally, does not handle missing data well.	Can handle missing data (either ignoring instances with missing values / or using surrogate splits).	Can handle missing data through surrogate splits.	Typically handles missing data by treating missing values as a separate category.
Pruning	Not incorporated	Incorporated	Incorporated	Not incorporated
Feature Selection	Information Gain	Gain Ratio	Gini index	Chi-Square
Tree structure	Polytree	Polytree	Binary tree	Polytree
Types of model	Classification	Classification / Regression	Classification / Regression	Classification / Regression

Decision Tree (Hyperparameter Tuning)

Types of Hyperparameters

- **Hyper-parameters are non-learnable** parameters that cannot be inferred from the training data (instead, they must be set outside of the training process).
- Often these parameters define a specific architecture for a given model
- In Decision Trees, the parameters consist of the selected features (f), and their associated split points s , that define how data propagate through the nodes in a tree.
- Some of the most common hyperparameters include:
 - Choice of **splitting loss function**, used to determine (f, s) at a given node
 - **Maximum depth of the tree**, which sets a hard limit on how much branching can occur
 - **Minimum number of samples for a split**, which places a lower bound on how much data must enter a node for it not to be considered a leaf node during training
 - **Maximum number of leaf nodes**, a value used to set an upper bound on the number of terminal points in the tree

Decision Tree (Hyperparameter Tuning)

- Hyper-parameters are the variables that you specify while building a machine-learning model.
 - This includes, for example, how the algorithm splits the data (either by entropy or Gini impurity).
- We can tune hyperparameters in Decision Trees by comparing models trained with different parameter configurations, on the same data.
 - An optimal model can then be selected from various attempts using relevant metrics.
- Three of the most popular approaches for hyperparameter tuning include *Baby Sitting*, *Grid Search*, *Randomized Search*, and *Bayesian Search*
- Optimizing the values of the hyperparameters, for a specific task, will often lead to superior model performance

Decision Tree (Hyperparameter Tuning)

- **Baby Sitting:** Purely trial & error and 100% manual.
- **Grid Search:** This involves an exhaustive search through a list of all possible hyperparameter values to try.
 - Every possible combination is tried, and then the best-performing model can be selected (computationally expensive)
- **Randomized Search:** In this approach, a random set of samples is selected from the hyperparameter space ~ (more computationally efficient, but can yield lower quality results if we are not careful with the parameter space definition)
- **Bayesian Search:** Build a probabilistic model to identify the best set of hyperparameters by statistical distributions and can further influence how the tuning performs through a careful selection of prior distributions.
- Regardless as to which method we use, it is best practice to evaluate each hyperparameter configuration with Cross-Validation.

Decision Tree (Hyperparameter Tuning)

Techniques	Advantages	Disadvantages
Grid Search	<ul style="list-style-type: none">• Systematic approach ~ tests all possible combinations within the defined grid.• Can find the optimal hyperparameter values if they lie within the search space	<ul style="list-style-type: none">• Computationally expensive ~ especially for large search spaces and complex models.• May not find the optimal hyperparameters if they lie outside the defined search space
Random Search	<ul style="list-style-type: none">• More efficient than Grid Search, especially for large search spaces.• Can explore a broader range of hyperparameter values.• Can find a good set of hyperparameters with a fewer number of iterations.	<ul style="list-style-type: none">• Lacks the systematic approach of Grid Search.• May require more iterations to find the optimal hyperparameters.• Performance depends on the number of iterations and the sampling strategy.

Decision Tree (Hyperparameter Tuning)

Hyperparameter Tuning using the Grid Search Approach

```
start_time = 0
# hyperparameter tuning process - using GridSearchCV
from sklearn.model_selection import GridSearchCV
```

```
params = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [None, 2, 4, 6, 8, 10],
    'max_features': [None, 'sqrt', 'log2', 0.2, 0.4, 0.6, 0.8],
    'splitter': ['best', 'random']
}
```

```
optimized_hyperparam_DTclassifier_GridSearch = GridSearchCV(
    estimator=DecisionTreeClassifier(),
    param_grid=params,
    cv=5,
    n_jobs=5,
    verbose=1,
)
```

```
start_time = time.time()
optimized_hyperparam_DTclassifier_GridSearch.fit(X_train,y_train)
duration_with_hyperparam_tuning = time.time() - start_time
```

```
print('\n ----- Optimized Hyperparameter Settings - Grid Search -----')
print(optimized_hyperparam_DTclassifier_GridSearch.best_params_)
```

The initialization of a search space in GridSearch function

----- Optimized Hyperparameter Settings - Grid Search -----
{ 'criterion': 'gini', 'max_depth': 2, 'max_features': None,
 'splitter': 'best' }

Decision Tree (Hyperparameter Tuning)

Hyperparameter Tuning using the Grid Search Approach

```
model_optimizedGridSearch = optimized_hyperparam_DTclassifier_GridSearch.best_estimator_  
y_pred_optimizedGridSearch = model_optimizedGridSearch.predict(X_test)
```

```
print("\nAccuracy Score (%):', metrics.accuracy_score(y_test,y_pred_optimizedGridSearch))
```

```
#confusion metric  
cm = confusion_matrix(y_test, y_pred_optimizedGridSearch)  
print("\nConfusion Metrics\n', cm)
```

```
#classification report  
print("\n-----Classification Reports-----\n')  
print(classification_report(y_test, y_pred_optimizedGridSearch))  
print('Computation time: %.2f' % duration_with_hyperparam_tuning)
```

Accuracy Score (%): 0.94

Confusion Metrics

[[64 4]
[2 30]]

-----Classification Reports-----

	precision	recall	f1-score	support
0	0.97	0.94	0.96	68
1	0.88	0.94	0.91	32
accuracy			0.94	100
macro avg	0.93	0.94	0.93	100
weighted avg	0.94	0.94	0.94	100

Computation time: 2.69

Decision Tree (Hyperparameter Tuning)

Hyperparameter Tuning using the Grid Search Approach

```
from sklearn import tree
text_representation_after_hyperparam_tuning = tree.export_text(model_optimizedGridSearch,
feature_names=feature_cols)
print("\n-----')
print('Generated Rules (after hyperparameter tuning - Grid Search)')
print('-----\n')
print(text_representation_after_hyperparam_tuning)
```

Generated Rules (after hyperparameter tuning - Grid Search)

```
|--- Age <= 44.50
| |--- EstimatedSalary <= 90500.00
| | |--- class: 0
| |--- EstimatedSalary > 90500.00
| | |--- class: 1
|--- Age > 44.50
| |--- Age <= 46.50
| | |--- class: 1
| |--- Age > 46.50
| | |--- class: 1
```

Decision Tree (Hyperparameter Tuning)

Hyperparameter Tuning using the Grid Search Approach

Without hyperparameter tuning, depth = 2

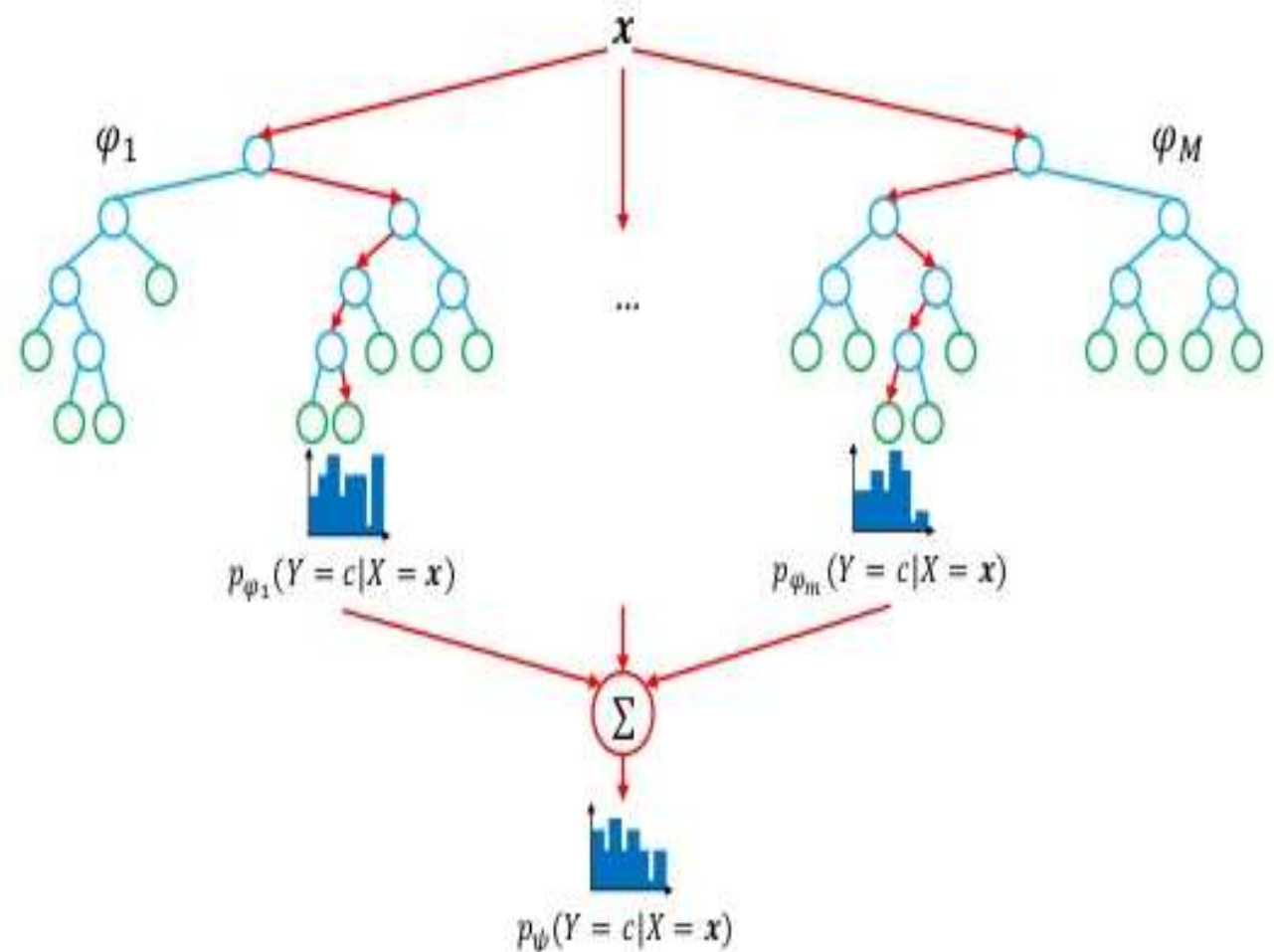
```
|--- Age <= 44.50
|   |--- EstimatedSalary <= 90500.00
|   |   |--- class: 0
|   |--- EstimatedSalary > 90500.00
|   |   |--- class: 1
|--- Age > 44.50
|   |--- EstimatedSalary <= 41500.00
|   |   |--- class: 1
|   |--- EstimatedSalary > 41500.00
|   |   |--- class: 1
```

With hyperparameter tuning

```
|--- Age <= 44.50
|   |--- EstimatedSalary <= 90500.00
|   |   |--- class: 0
|   |--- EstimatedSalary > 90500.00
|   |   |--- class: 1
|--- Age > 44.50
|   |--- Age <= 46.50
|   |   |--- class: 1
|   |--- Age > 46.50
|   |   |--- class: 1
```

Random Forest

- Random forests consist of multiple single trees each based on a random sample of the training data.
- They are typically more accurate than single decision trees as the decision boundary becomes more accurate and stable as more trees are added.

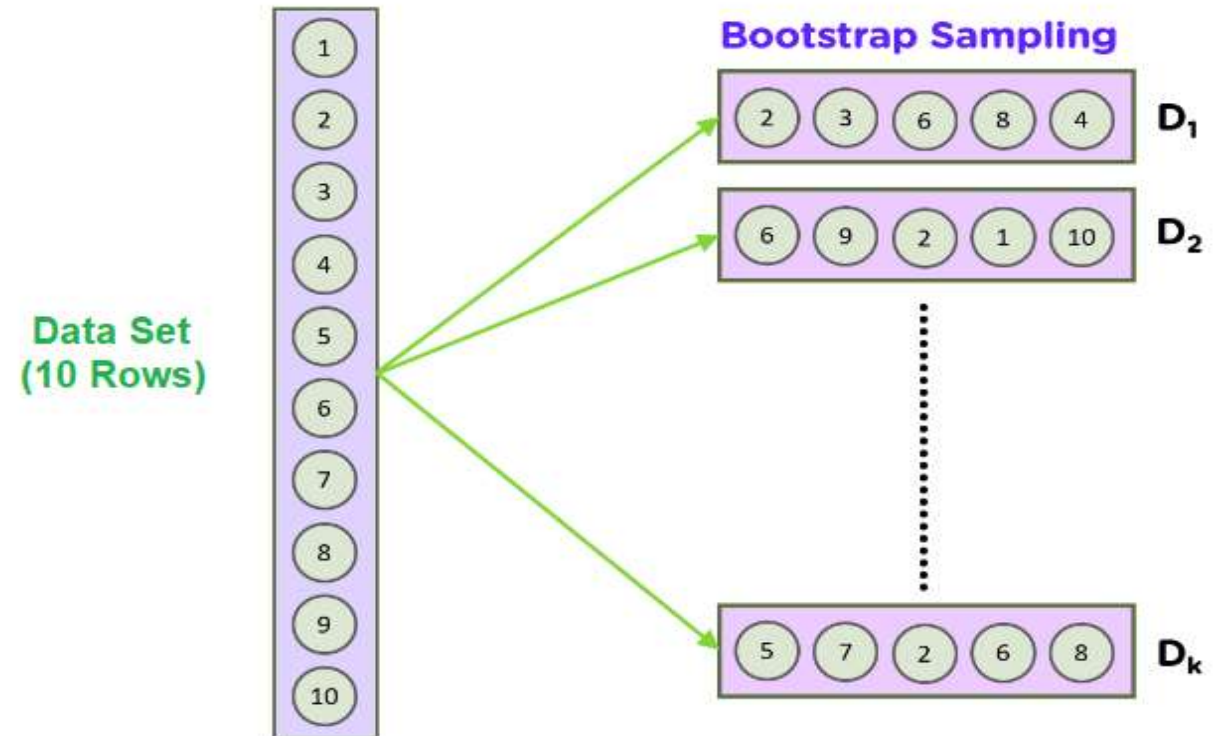


Random Forest

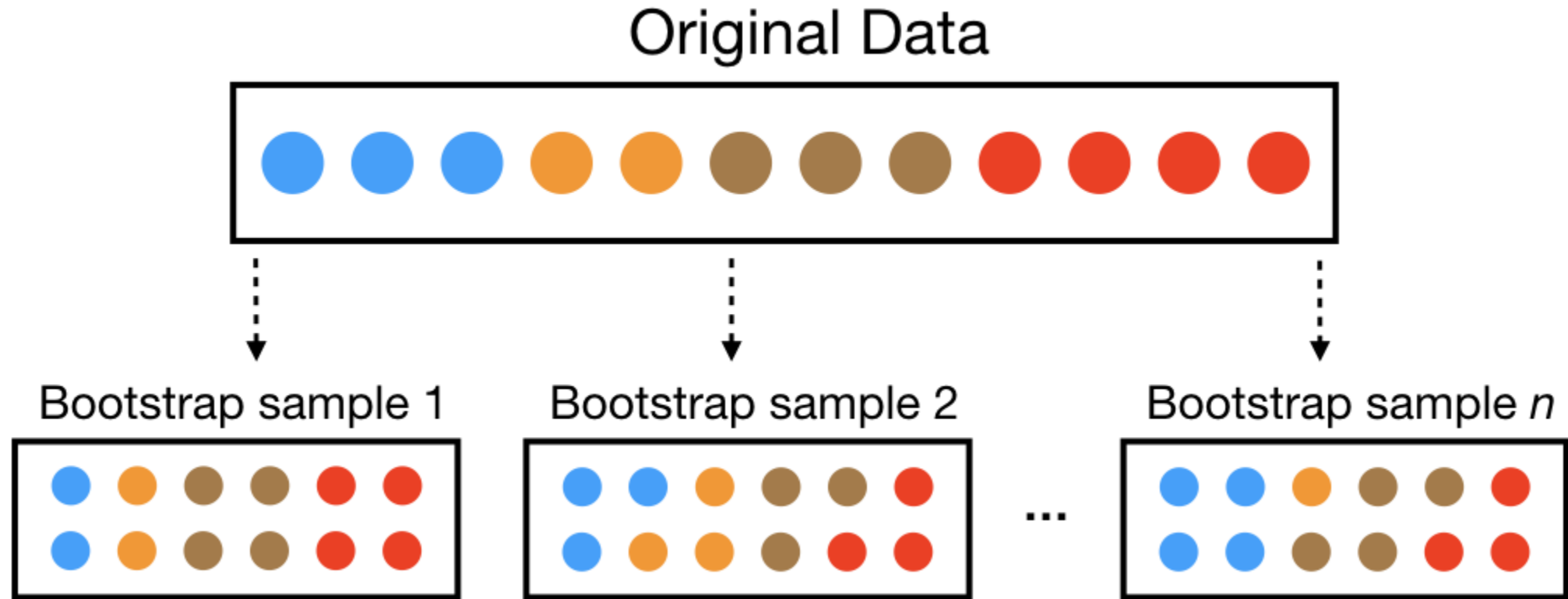
- Before understanding the workings of the random forest algorithm in machine learning, we must look into the ensemble learning technique.
 - Ensemble learning - means combining multiple models.
 - Thus a collection of models is used to make predictions rather than an individual model.
- Ensemble uses two types of methods:
 - Bagging – creates a different training subset from sample training data with replacement and the final output is based on majority voting.
 - Boosting - combines weak learners into strong learners by creating sequential models such that the final model has the highest accuracy

Random Forest (Bootstrapping)

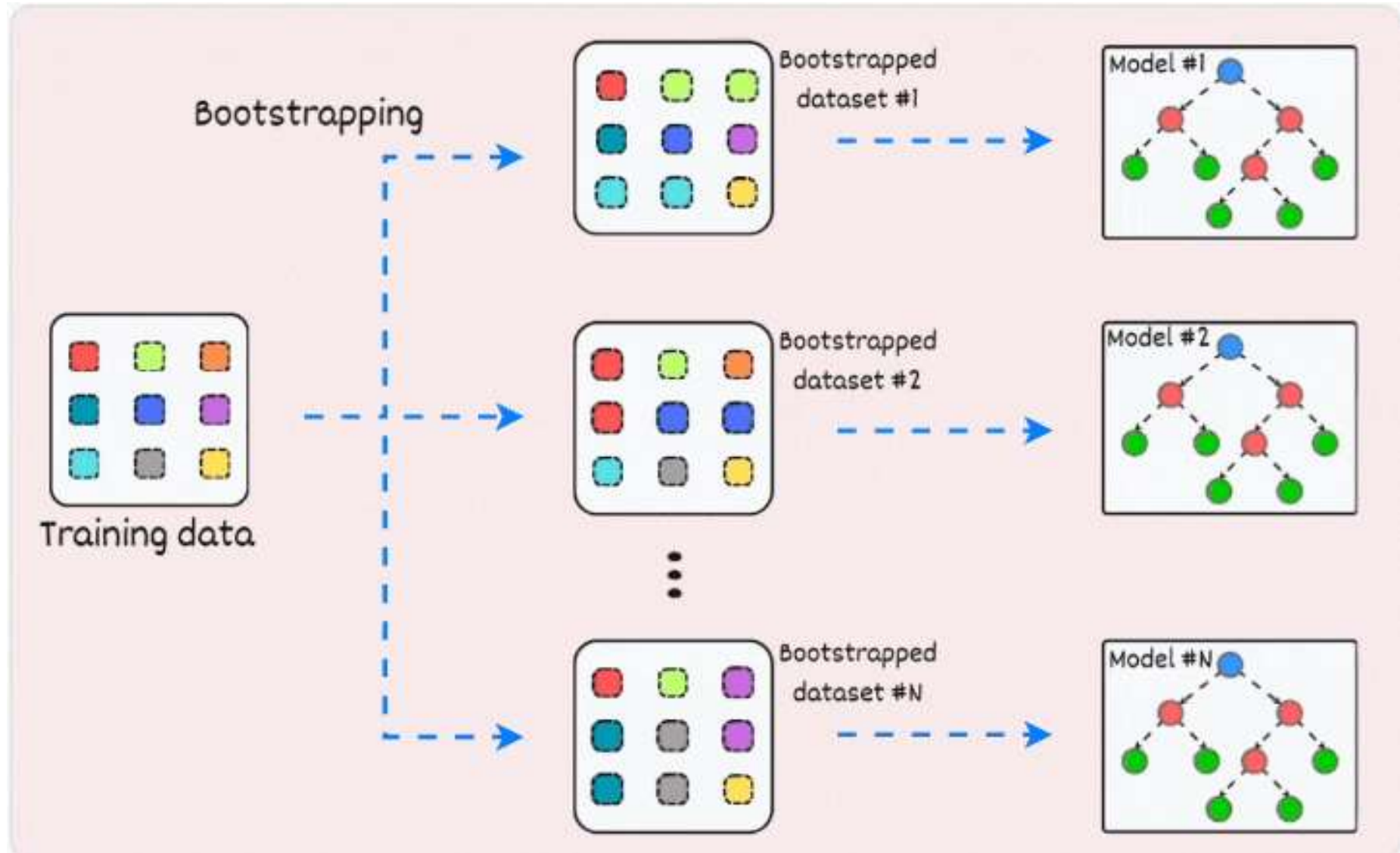
- The goal of a bootstrap approach is to estimate an unknown data distribution, given a limited dataset size.
- Design to decrease variance.
- The basic idea is that during random **data sampling**, or taking a smaller subset of the dataset, we can also **re-sample** the data:
 - This process is repeated a large number of times (1,000-10,000 times), and by taking statistics during each iteration, we can gain a sense of the data distribution.



Random Forest (Bootstrapping)

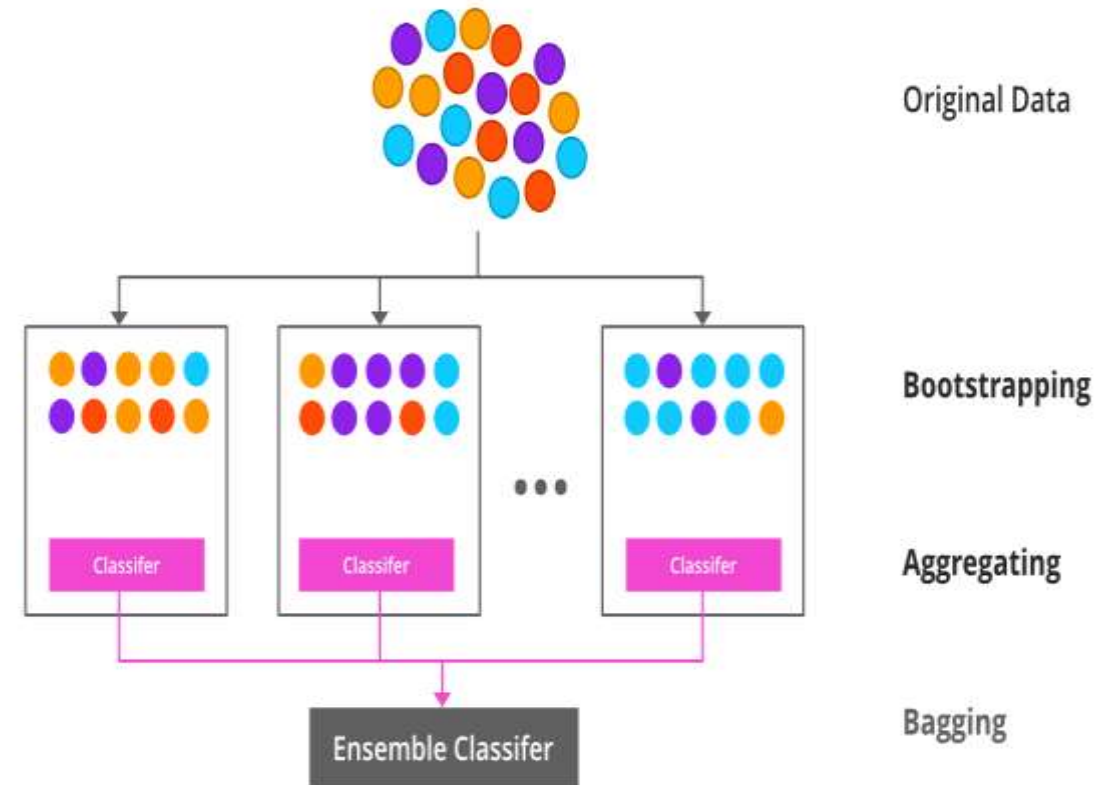


Random Forest (Bootstrapping)



Random Forest (Bagging)

- Bagging can also be called bootstrap aggregation, which is used in ensemble learning methods to increase the accuracy and performance of ensemble learning methods.
- Increasing the performance by reducing the variance in a dataset.
- Improves the model by reducing the overfitting problem and can work in high-dimension data.



Random Forest (Bagging)

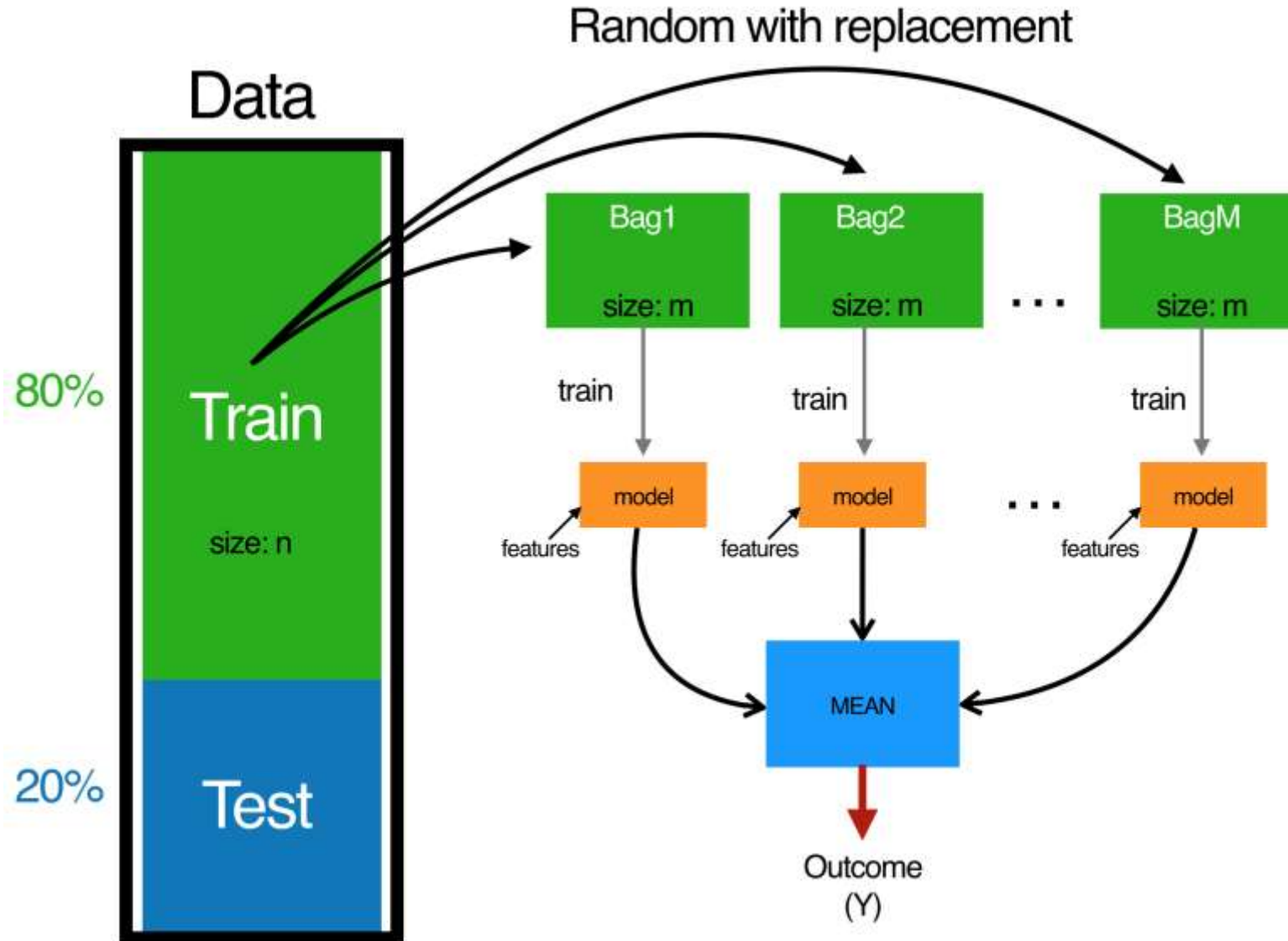
Procedures:

1. Selection of Subset - starts by choosing a random sample, or subset, from the entire dataset.
2. Bootstrapping - each model is then created from these samples (bootstrap samples), which are taken from the original data with replacement (row sampling).
3. Independent Model Training - each model is trained independently on its corresponding Bootstrap Sample.
4. Majority Voting - the final output is determined by combining the results of all models through majority voting.
5. Aggregation – involves combining all the results and generating the final output based on majority voting, which is known as aggregation.

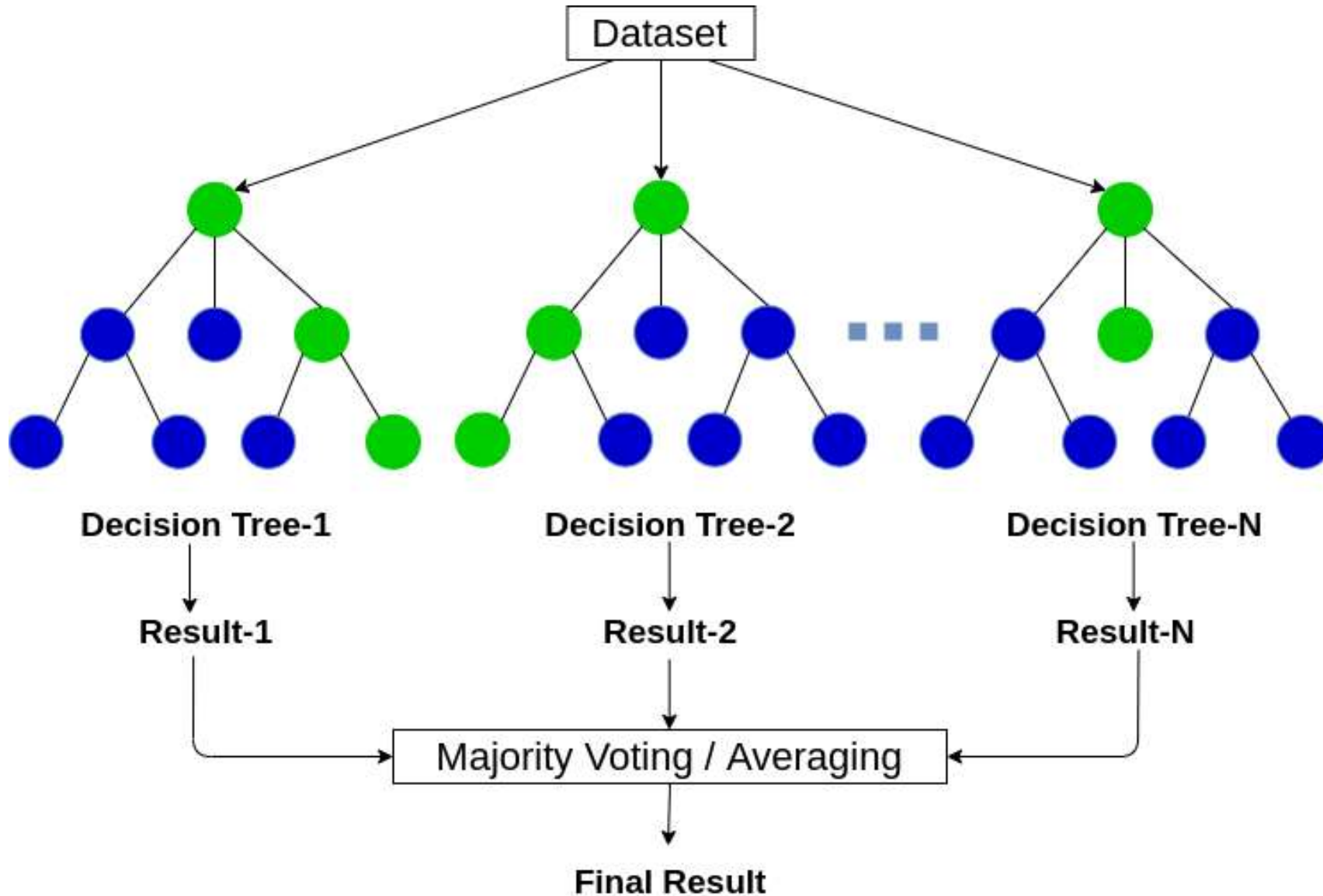
Random Forest

- Basically, a decision tree has low bias and high variance, thus by averaging the result of many decision trees reduces the variance while maintaining that low bias.
- Combining trees is known as an 'ensemble method'.
 - 1) The method starts with creating two or more separate models with the same dataset.
 - 2) Then a Voting based Ensemble model can be used to wrap the previous models and aggregate the predictions of those models.
 - 3) After the Voting based Ensemble model is constructed, it can be used to make a prediction on new data

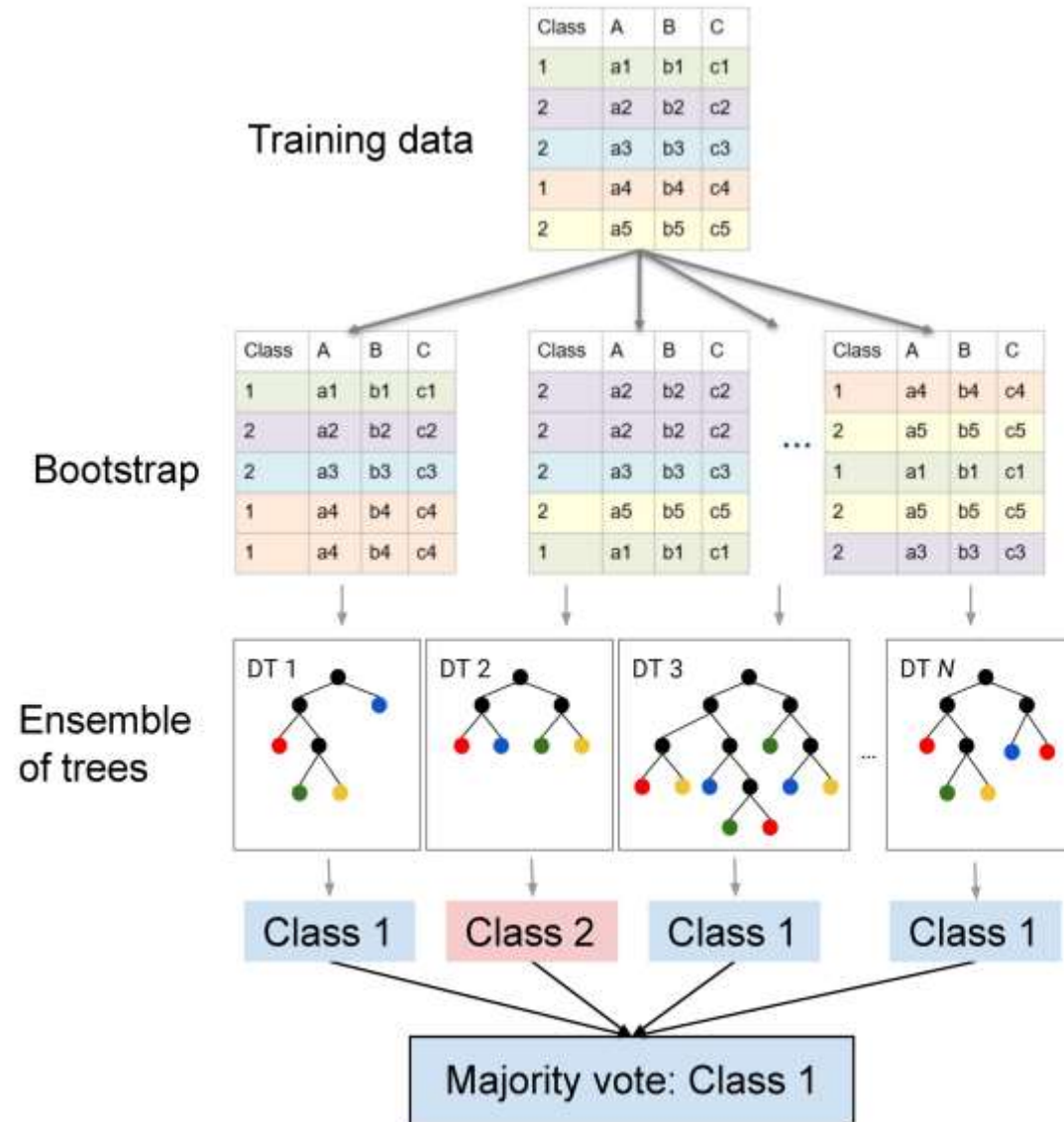
Random Forest (Bagging)



Random Forest (Bagging)

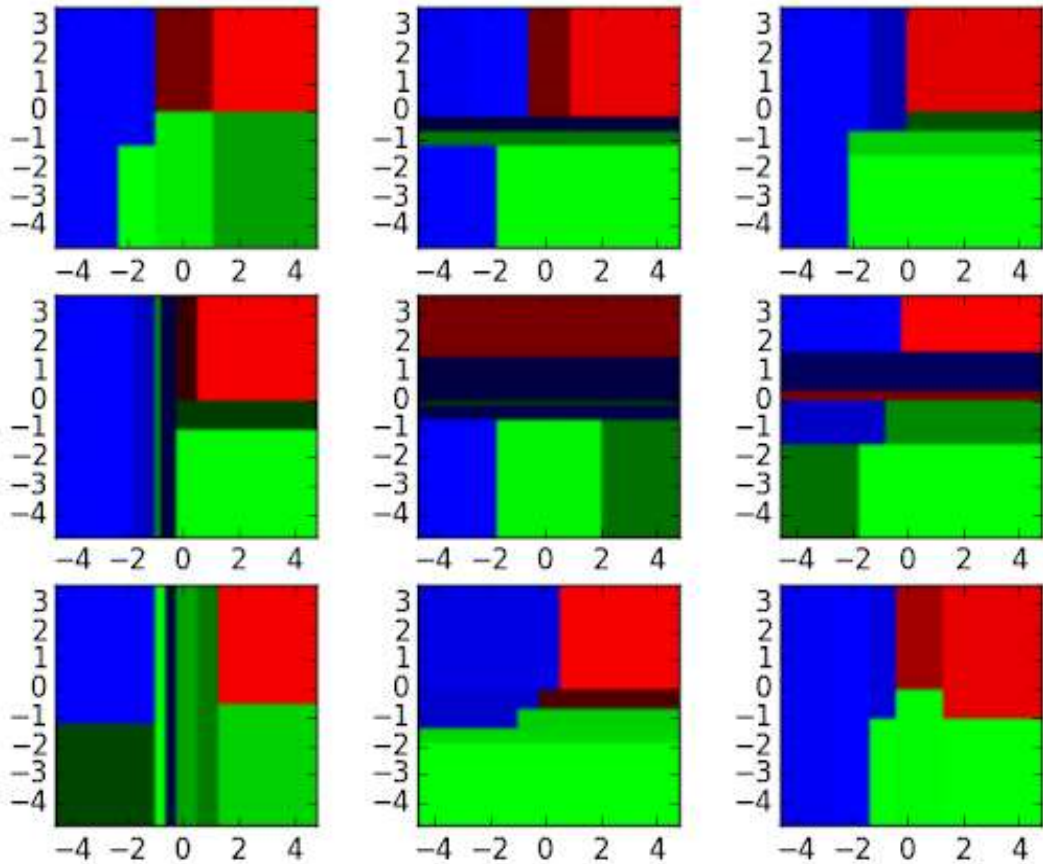


Random Forest (Bagging)

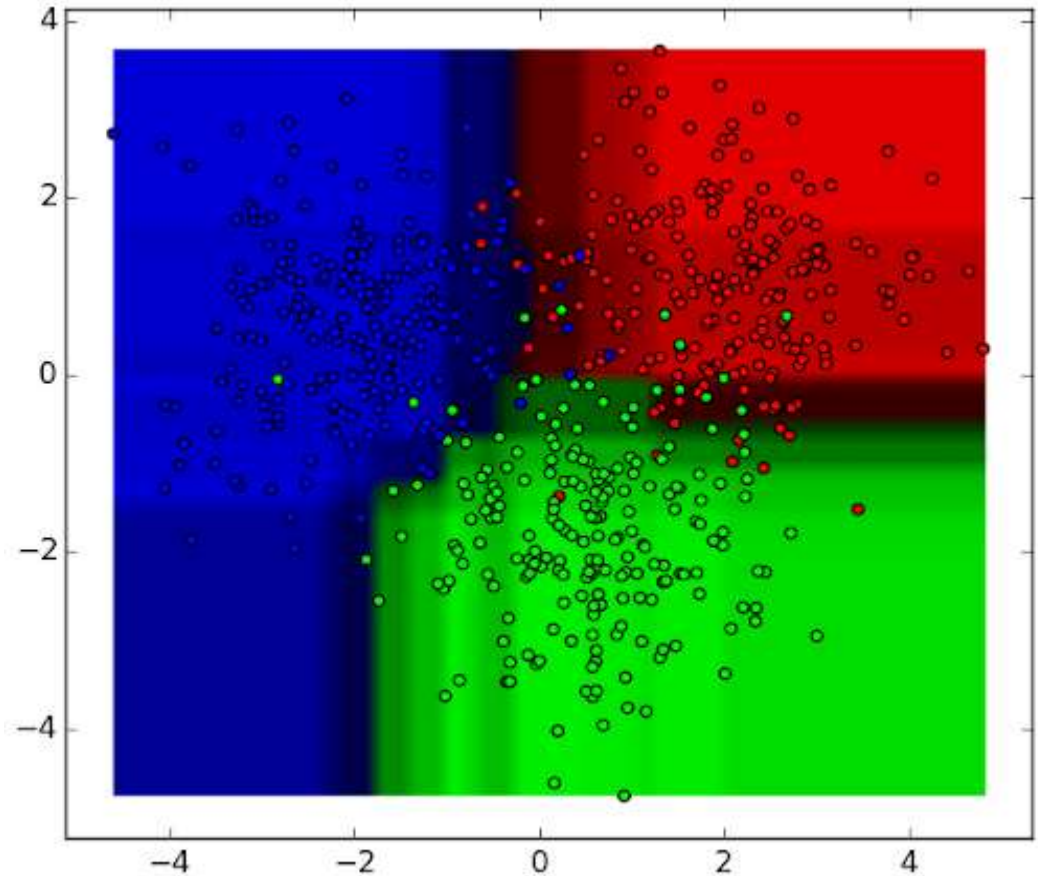


Random Forest (Bagging)

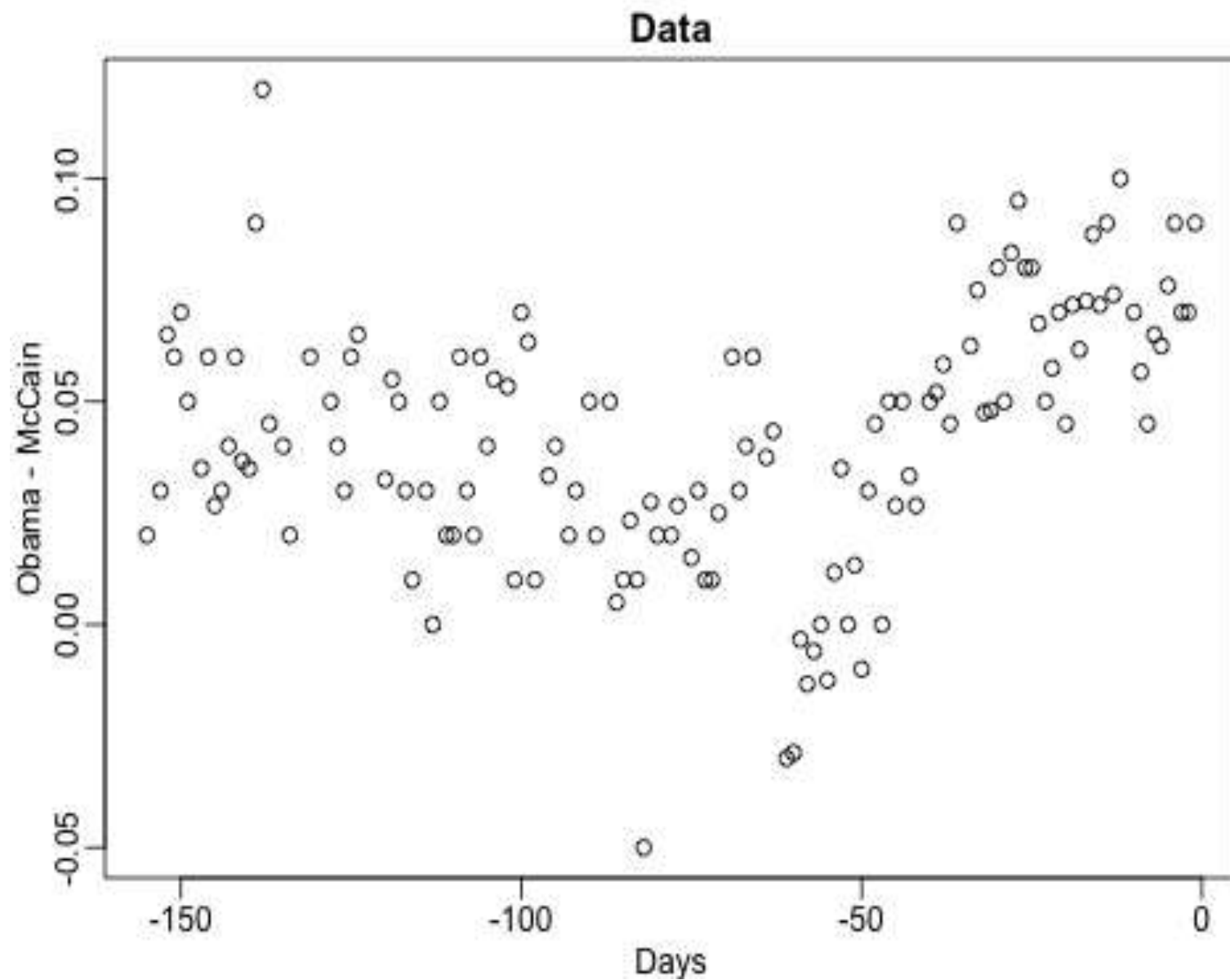
Results from different “tree classifiers”



Final Result for a random forest classifier



Random Forest



Random Forest

Important Features:

- **Diversity:** Not all attributes/variables/features are considered while making an individual tree; each tree is different.
- **Immune to the curse of dimensionality:** Since each tree does not consider all the features, the feature space is reduced.
- **Parallelization:** Each tree is created independently out of different data and attributes. This means we can fully use the CPU to build random forests.
- **Train-Test split:** In a random forest, we don't have to segregate the data for train and test as there will always be 30% of the data which is not seen by the decision tree.
- **Stability:** Stability arises because the result is based on majority voting/ averaging.

Random Forest

Hyperparameters:

Hyperparameters to Increase the Predictive Power

- **n_estimators:** Number of trees the algorithm builds before averaging the predictions.
- **max_features:** Maximum number of features random forest considers splitting a node.
- **mini_sample_leaf:** Determines the minimum number of leaves required to split an internal node.
- **criterion:** How to split the node in each tree? (Entropy/Gini impurity/Log Loss)
- **max_leaf_nodes:** Maximum leaf nodes in each tree

Random Forest

Hyperparameters:

Hyperparameters to Increase the Speed

- ***n_jobs***: it tells the engine how many processors it is allowed to use.
 - If the value is 1, it can use only one processor, but if the value is -1, there is no limit.
- ***random_state***: controls randomness of the sample.
 - The model will always produce the same results if it has a definite value of random state and has been given the same hyperparameters and training data.
- ***oob_score***: *OOB* means out of the bag (random forest cross-validation method).
 - In this, one-third of the sample is not used to train the data; instead used to evaluate its performance. These samples are called out-of-bag samples.

Random Forest

Case Study (Random Forest for Classification): Bank Marketing Campaign

- The data is related to direct marketing campaigns of a Portuguese banking institution.
- The marketing campaigns were based on phone calls.
- Often, more than one contact with the same client was required, to access if the product (bank term deposit) would be ('yes') or not ('no') subscribed
- Comparison – Linear Regression, Naïve Bayesian, Decision Tree, Random Forest.

age	job	marital	education	default	balance	housing	loan	contact	day	month	duration	campaign	pdays	previous	poutcome	Target
58	management	married	tertiary	no	2143	yes	no	unknown	5	may	261	1	-1	0	unknown	no
44	technician	single	secondary	no	29	yes	no	unknown	5	may	151	1	-1	0	unknown	no
33	entrepreneur	married	secondary	no	2	yes	yes	unknown	5	may	76	1	-1	0	unknown	no
47	blue-collar	married	unknown	no	1506	yes	no	unknown	5	may	92	1	-1	0	unknown	no

Random Forest

```
# Data Processing
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
```

```
my_data = pd.read_csv('bank-full.csv')
```

```
#checking any possible missing values
```

```
val=my_data.isnull().values.any()
```

```
if val==True:
```

```
    print("Missing values present : ", my_data.isnull().values.sum())
```

```
    my_data=my_data.dropna()
```

```
else:
```

```
    print("No missing values present")
```

```
print(my_data.describe().T)
```

```
# Data visualization - histogram
```

```
my_data.hist(figsize=(10,10),color="blueviolet",grid=False)
```

```
plt.show()
```

```
# Correlation
```

```
cor=my_data.corr()
```

```
plt.subplots(figsize=(10,8))
```

```
sns.heatmap(cor,annot=True)
```

```
plt.show()
```

No missing values present

	count	mean	std	...	50%	75%	max
age	45211.0	40.936210	10.618762	...	39.0	48.0	95.0
balance	45211.0	1362.272058	3044.765829	...	448.0	1428.0	102127.0
day	45211.0	15.806419	8.322476	...	16.0	21.0	31.0
duration	45211.0	258.163080	257.527812	...	180.0	319.0	4918.0
campaign	45211.0	2.763841	3.098021	...	2.0	3.0	63.0
pdays	45211.0	40.197828	100.128746	...	-1.0	-1.0	871.0
previous	45211.0	0.580323	2.303441	...	0.0	0.0	275.0

[7 rows x 8 columns]

Random Forest

Data Processing

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
```

```
my_data = pd.read_csv('bank-full.csv')
```

#checking any possible missing values

```
val=my_data.isnull().values.any()
if val==True:
    print("Missing values present : ", my_data.isnull().values.sum())
    my_data=my_data.dropna()
else:
    print("No missing values present")
```

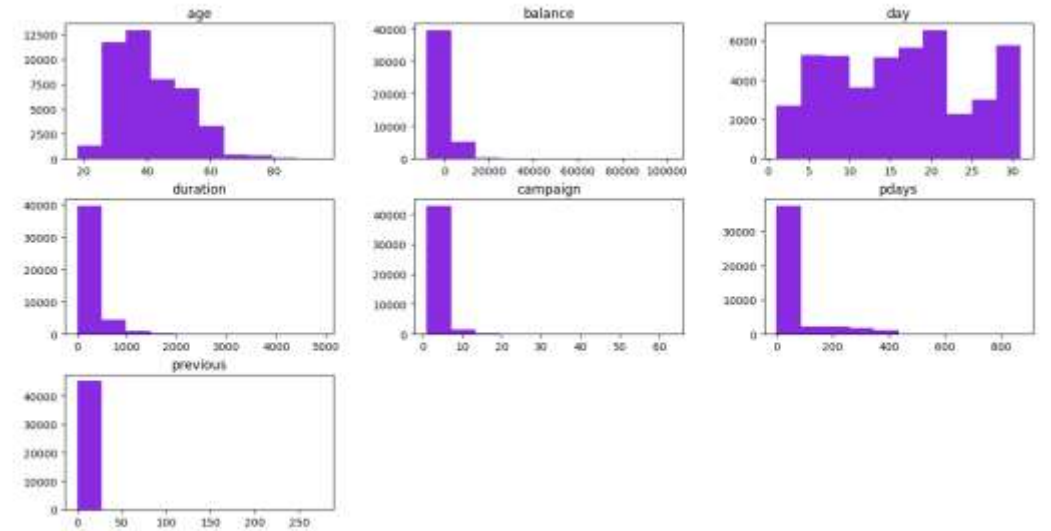
```
print(my_data.describe().T)
```

Data visualization - histogram

```
my_data.hist(figsize=(10,10),color="blueviolet",grid=False)
plt.show()
```

Correlation

```
cor=my_data.corr()
plt.subplots(figsize=(10,8))
sns.heatmap(cor,annot=True)
plt.show()
```



Random Forest

```
# Label encoder order in alphabetical
from sklearn.preprocessing import LabelEncoder
labelencoder_X = LabelEncoder()
my_data['job'] = labelencoder_X.fit_transform(my_data['job'])
my_data['marital'] = labelencoder_X.fit_transform(my_data['marital'])
my_data['education'] = labelencoder_X.fit_transform(my_data['education'])
my_data['default'] = labelencoder_X.fit_transform(my_data['default'])
my_data['housing'] = labelencoder_X.fit_transform(my_data['housing'])
my_data['loan'] = labelencoder_X.fit_transform(my_data['loan'])
my_data['contact'] = labelencoder_X.fit_transform(my_data['contact'])
my_data['month'] = labelencoder_X.fit_transform(my_data['month'])

#function to create group of ages, this helps because we have 78 different values here
def age(dataframe):
    dataframe.loc[dataframe['age'] <= 32, 'age'] = 1
    dataframe.loc[(dataframe['age'] > 32) & (dataframe['age'] <= 47), 'age'] = 2
    dataframe.loc[(dataframe['age'] > 47) & (dataframe['age'] <= 70), 'age'] = 3
    dataframe.loc[(dataframe['age'] > 70) & (dataframe['age'] <= 98), 'age'] = 4

    return dataframe
age(my_data)

#function to change range
def duration(data):
    data.loc[data['duration'] <= 102, 'duration'] = 1
    data.loc[(data['duration'] > 102) & (data['duration'] <= 180), 'duration'] = 2
    data.loc[(data['duration'] > 180) & (data['duration'] <= 319), 'duration'] = 3
    data.loc[(data['duration'] > 319) & (data['duration'] <= 644.5), 'duration'] = 4
    data.loc[data['duration'] > 644.5, 'duration'] = 5

    return data
duration(my_data)

#encoding for pdays
my_data.loc[(my_data['pdays'] == 999), 'pdays'] = 1
my_data.loc[(my_data['pdays'] > 0) & (my_data['pdays'] <= 10), 'pdays'] = 2
my_data.loc[(my_data['pdays'] > 10) & (my_data['pdays'] <= 20), 'pdays'] = 3
my_data.loc[(my_data['pdays'] > 20) & (my_data['pdays'] != 999), 'pdays'] = 4

#encoding for poutcome
my_data['poutcome'].replace(['unknown', 'failure', 'other', 'success'], [1,2,3,4], inplace = True)

final_data=my_data
print(final_data.shape)
print(final_data.head())
```

Data encoding according to range {1,2,...5}

	age	job	marital	education	...	pdays	previous	poutcome	Target
0	3	4	1	2	...	-1	0	1	no
1	2	9	2	1	...	-1	0	1	no
2	2	2	1	1	...	-1	0	1	no
3	2	1	1	3	...	-1	0	1	no
4	2	11	2	3	...	-1	0	1	no

[5 rows x 17 columns]

Random Forest

```
from sklearn import preprocessing
from sklearn.metrics import accuracy_score
from sklearn import metrics
```

```
X = final_data.values[:,0:15] #select input features
Y = final_data.values[:,16]  #select target features
```

```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.30, random_state = 7)
print("\n ***** Results *****")
```

```
# Logistic Regression
```

```
from sklearn.linear_model import LogisticRegression
LRclf = LogisticRegression()
LRclf.fit(X_train, Y_train)
y_predLR = LRclf.predict(X_test)
LR=accuracy_score(Y_test, y_predLR, normalize = True)
print('\nAccuracy score - Logistic Regression:',LR)
print('Confusion Matrix of LR:')
print(metrics.confusion_matrix(Y_test, y_predLR))
```

Accuracy score - Logistic Regression: 0.8812297257446181

Confusion Matrix of LR:

**[[11720 310]
 [1301 233]]**

```
# Naive Bayesian
```

```
from sklearn.naive_bayes import GaussianNB
NBclf = GaussianNB()
NBclf.fit(X_train, Y_train)
y_predNB = NBclf.predict(X_test)
NB=accuracy_score(Y_test, y_predNB, normalize = True)
print('\nAccuracy Score - Naive Bayes:',NB)
print('Confusion Matrix of Naive Bayes:')
print(metrics.confusion_matrix(Y_test,y_predNB))
```

Accuracy Score - Naive Bayes: 0.848938366263639

Confusion Matrix of Naive Bayes:

**[[10703 1327]
 [722 812]]**

Random Forest

```
# Decision Tree
from sklearn.tree import DecisionTreeClassifier
DTclf = DecisionTreeClassifier(criterion = 'entropy')
DTclf.fit(X_train, Y_train)
y_predDT = DTclf.predict(X_test)
DT=accuracy_score(Y_test, y_predDT, normalize = True)
print('\nAccuracy Score - Decision Tree:',DT)
print('Confusion Matrix of Decision Tree:')
print(metrics.confusion_matrix(Y_test, y_predDT))
```

Accuracy Score - Decision Tree: 0.871129460336184
Confusion Matrix of Decision Tree:
[[11120 910]
 [838 696]]

```
# Random Forest
from sklearn.ensemble import RandomForestClassifier
RFclf = RandomForestClassifier(n_estimators = 50)
RFclf.fit(X_train, Y_train)
y_predRF = RFclf.predict(X_test)
RF=accuracy_score(Y_test, y_predRF, normalize = True)
print('\nAccuracy Score - Random Forest:',RF)
print('Confusion Matrix of RF:')
print(metrics.confusion_matrix(Y_test, y_predRF))
```

Accuracy Score - Random Forest: 0.9003243880861103
Confusion Matrix of RF:
[[11669 361]
 [991 543]]

```
# Score evaluation
models = pd.DataFrame({
    'Models': ['Logistic Regression', 'Gaussian NB', 'Decision Tree', 'Random Forest Classifier'],
    'Score': [LR, NB, DT, RF]})
```

```
print("\n ----- The Best Classifiers by Score -----")
print(models.sort_values(by='Score', ascending=False))
```

----- The Best Classifiers by Score -----

	Models	Score
3	Random Forest	0.900324
0	Logistic Regression	0.881230
2	Decision Tree	0.871129
1	Gaussian NB	0.848938

Random Forest

Hyperparameters Tuning:

setting hyperparameter tuning

```
RFclf = RandomForestClassifier()  
param_dist = {'n_estimators': randint(50,500),  
              'max_depth': randint(1,20)}
```

Create a random forest classifier

Use random search to find the best hyperparameters

```
rand_searchRF = RandomizedSearchCV(RFclf,  
                                   param_distributions = param_dist,  
                                   n_iter=5,  
                                   cv=5)
```

----- Before Hyperparameters Tuning -----

{'max_depth': 30, 'n_estimators': 100}

Accuracy Score (%): 0.899955765260985

Best hyperparameters: {'max_depth': 15, 'n_estimators': 323}

----- After Hyperparameters Tuning -----

Accuracy Score (%): 0.9015777056915364

Random Forest

Case Study (Random Forest for Regression): Predicting Salaries

Position	Level	Salary
Business Analyst	1	45000
Junior Consultant	2	50000
Senior Consultant	3	60000
Manager	4	80000
Country Manager	5	110000
Region Manager	6	150000
Partner	7	200000
Senior Partner	8	300000
C-level	9	500000
CEO	10	1000000

Random Forest

```
import pandas as pd
import matplotlib.pyplot as plt
import sklearn
import warnings
```

```
warnings.filterwarnings('ignore')
```

```
df= pd.read_csv('Position_Salaries.csv')
print(df)
```

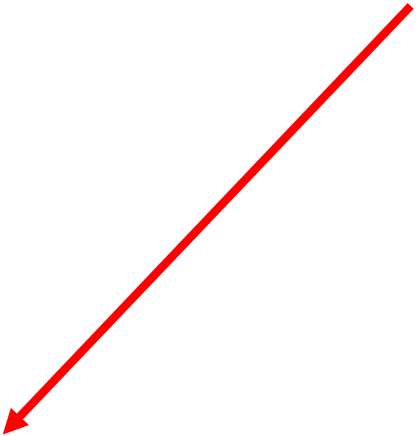
```
# Assuming df is your DataFrame
X = df.iloc[:,1:2].values #features
y = df.iloc[:,2].values  # Target variable
```

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.preprocessing import LabelEncoder
```

```
# Fitting the Regression Model to the dataset with 10 tree prediction
regressor = RandomForestRegressor(n_estimators=10, random_state=0, oob_score=True)
```

```
# Fit the regressor with x and y data
regressor.fit(X, y)
```

```
class sklearn.ensemble.RandomForestRegressor(n_estimators=100, *, criterion='squared_error',
max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_fea
tures=1.0, max_leaf_nodes=None, min_impurity_decrease=0.0, bootstrap=True, oob_score=False, n_jo
bs=None, random_state=None, verbose=0, warm_start=False, ccp_alpha=0.0, max_samples=None)
```



Random Forest

```
# Evaluating the model
from sklearn.metrics import mean_squared_error, r2_score
```

```
# Access the OOB Score
oob_score = regressor.oob_score_
print("\n ----- Evaluation -----")
print(f'Out-of-Bag Score: {oob_score}')
```

```
# Making predictions on the same data or new data
predictions = regressor.predict(X)
```

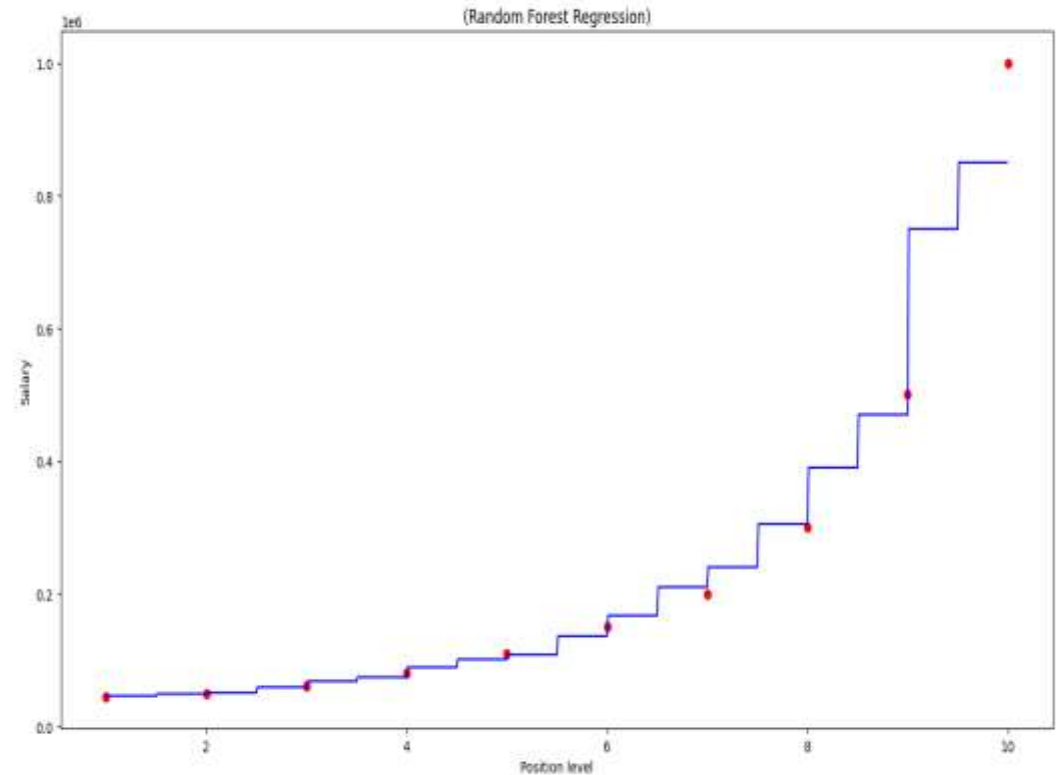
```
# Evaluating the model
mse = mean_squared_error(y, predictions)
print(f'Mean Squared Error: {mse}')
r2 = r2_score(y, predictions)
print(f'R-squared: {r2}')
```

----- Evaluation -----
Out-of-Bag Score: 0.6819214296245728
Mean Squared Error: 2384100000.0
R-squared: 0.9704434230386582



Random Forest

```
import numpy as np # linear algebra
# Visualising the Regression results (for higher resolution and smoother curve)
X_grid = np.arange(min(X), max(X), 0.01)
X_grid = X_grid.reshape((len(X_grid), 1))
plt.scatter(X, y, color = 'red')
plt.plot(X_grid, regressor.predict(X_grid), color = 'blue')
plt.title('(Random Forest Regression)')
plt.xlabel('Position level')
plt.ylabel('Salary')
plt.show()
```



Random Forest

```
from sklearn.tree import plot_tree
```

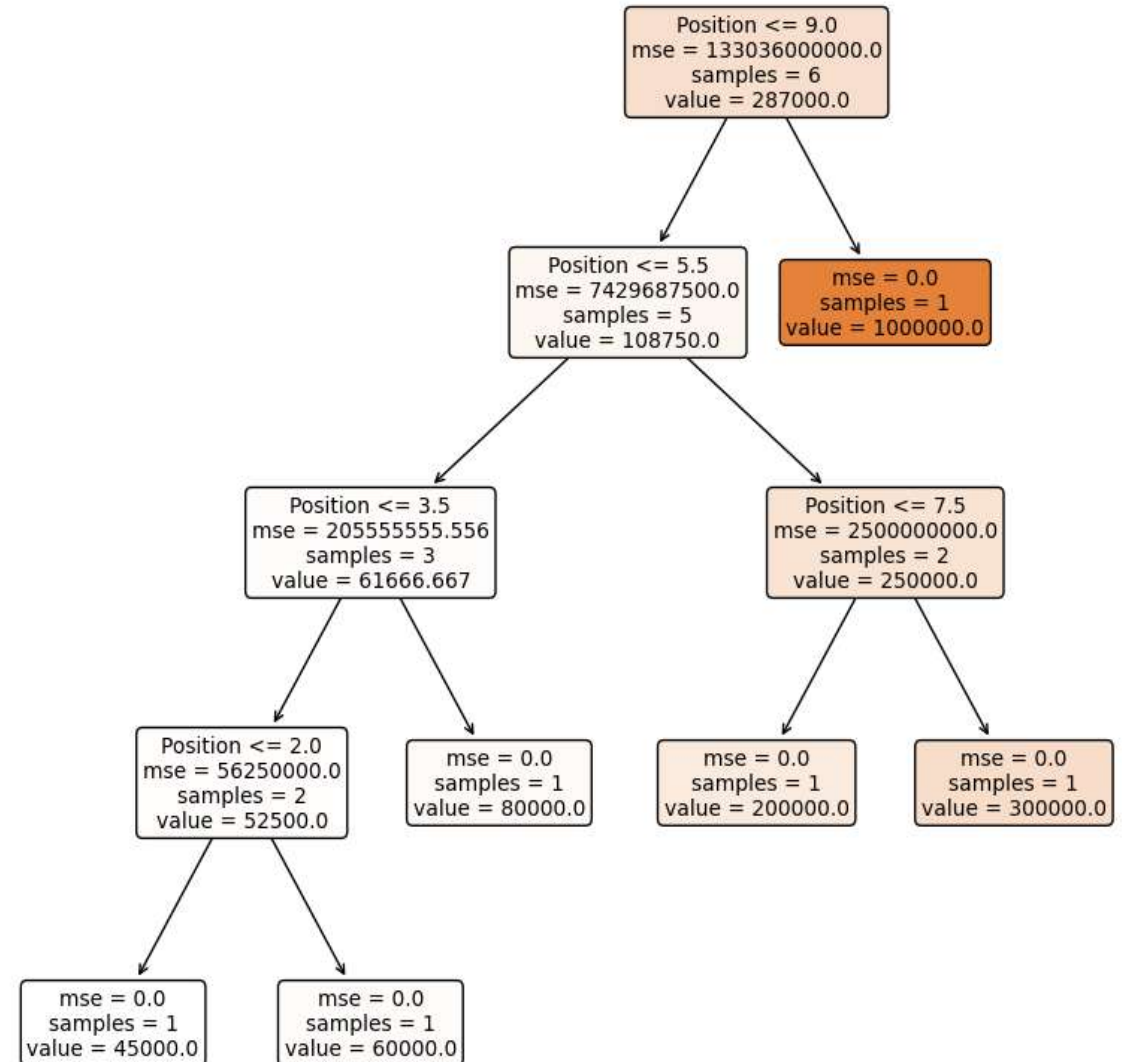
```
# Assuming regressor is your trained Random Forest model  
# Pick one tree from the forest, e.g., the first tree (index 0)  
tree_to_plot = regressor.estimators_[0]
```

```
# Plot the decision tree
```

```
plt.figure(figsize=(10, 10))
```

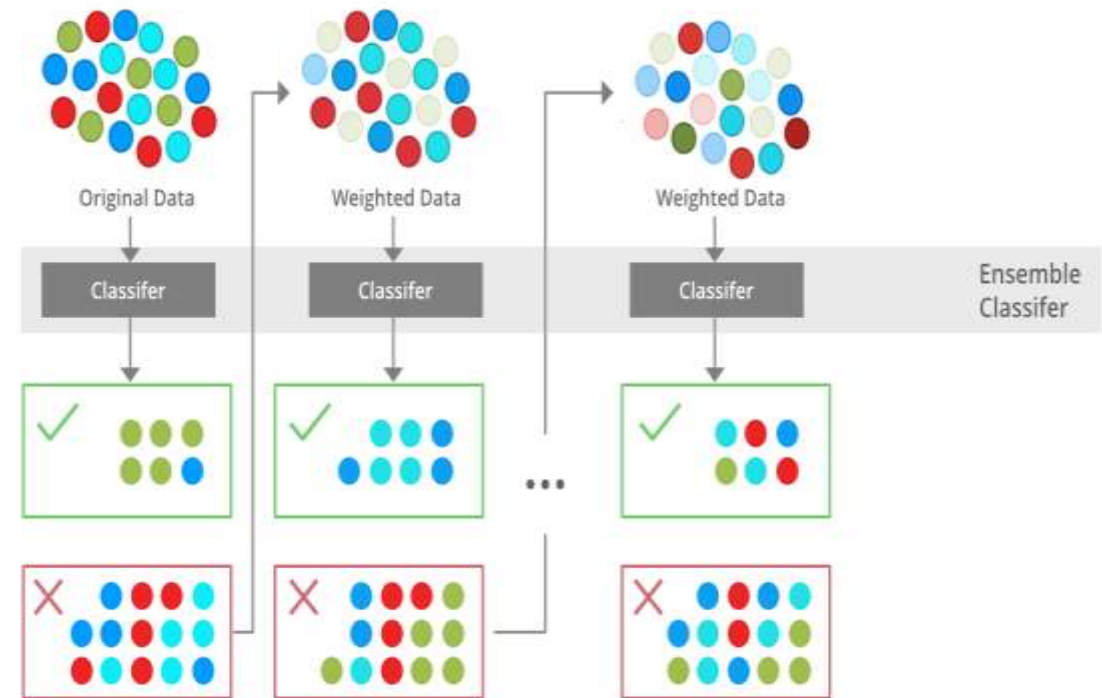
```
plot_tree(tree_to_plot, feature_names=df.columns.tolist(), filled=True,  
rounded=True, fontsize=10)
```

```
plt.show()
```

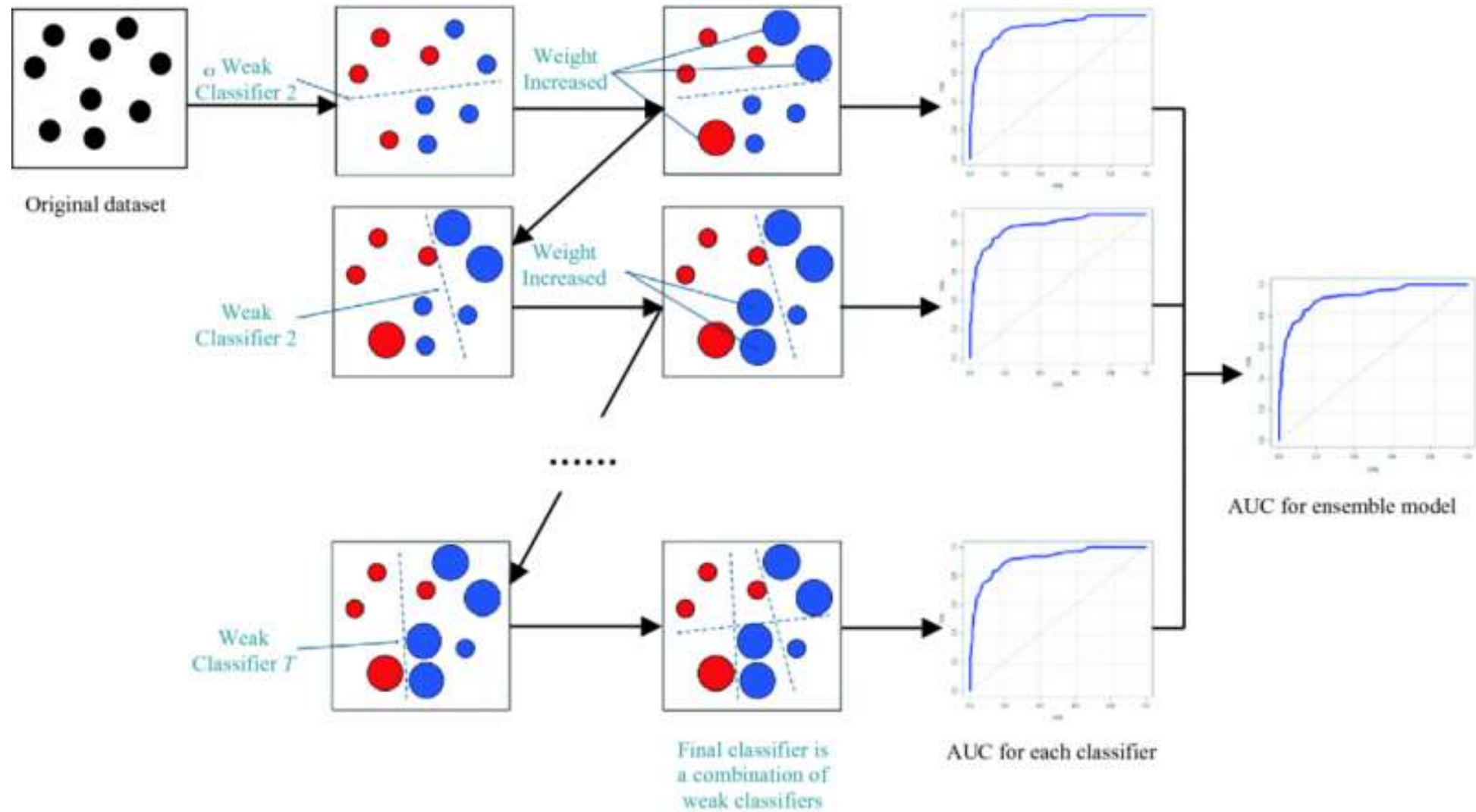


Boosting

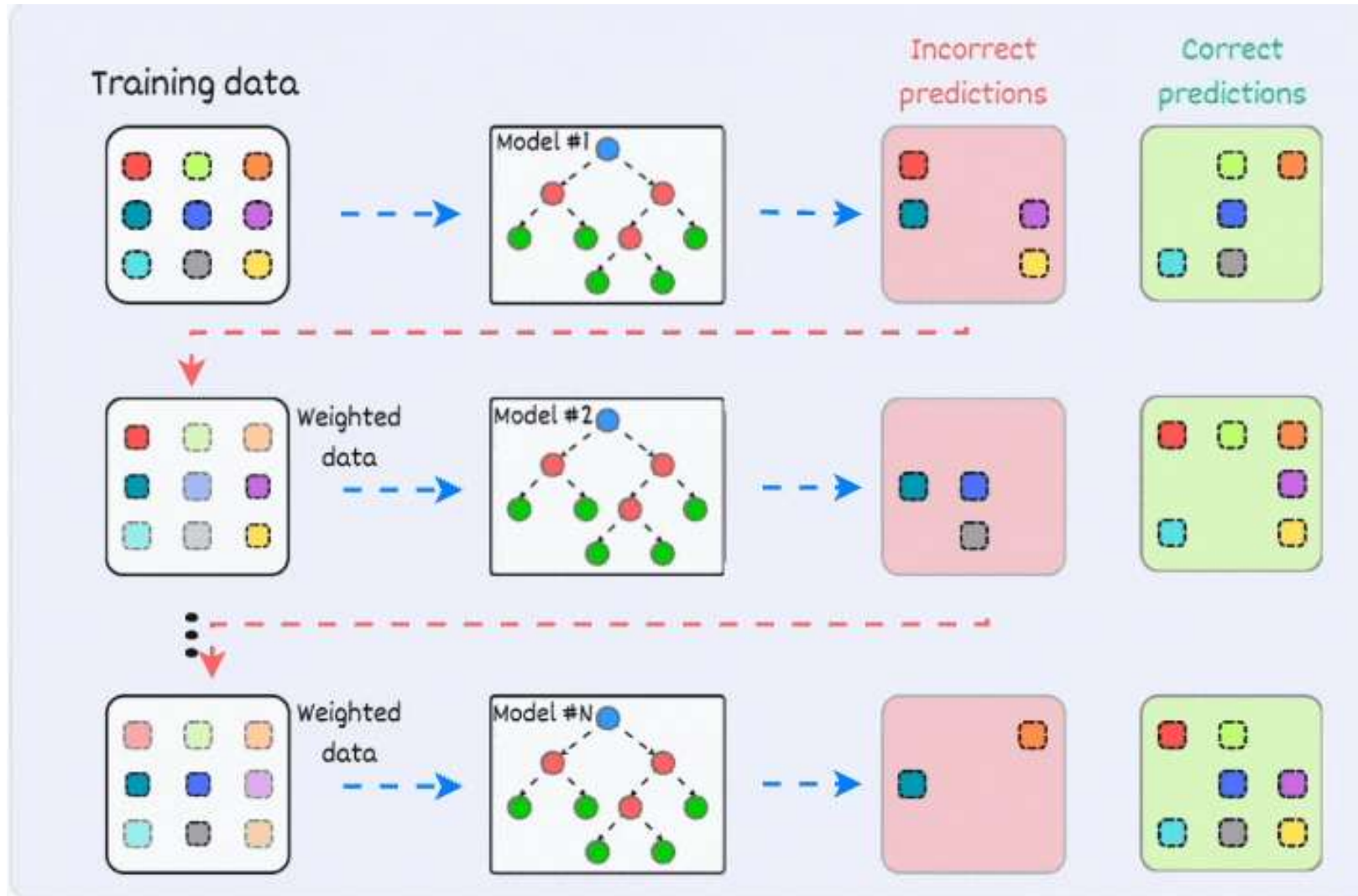
- Boosting works on the principle of improving the mistakes of the previous learner through the next learner.
- Design to decrease bias.
 - In boosting, weak learners are used which perform only slightly better than a random chance.
 - Boosting focuses on sequentially adding up these weak learners and filtering out the observations that a learner gets correct at every step.



Boosting



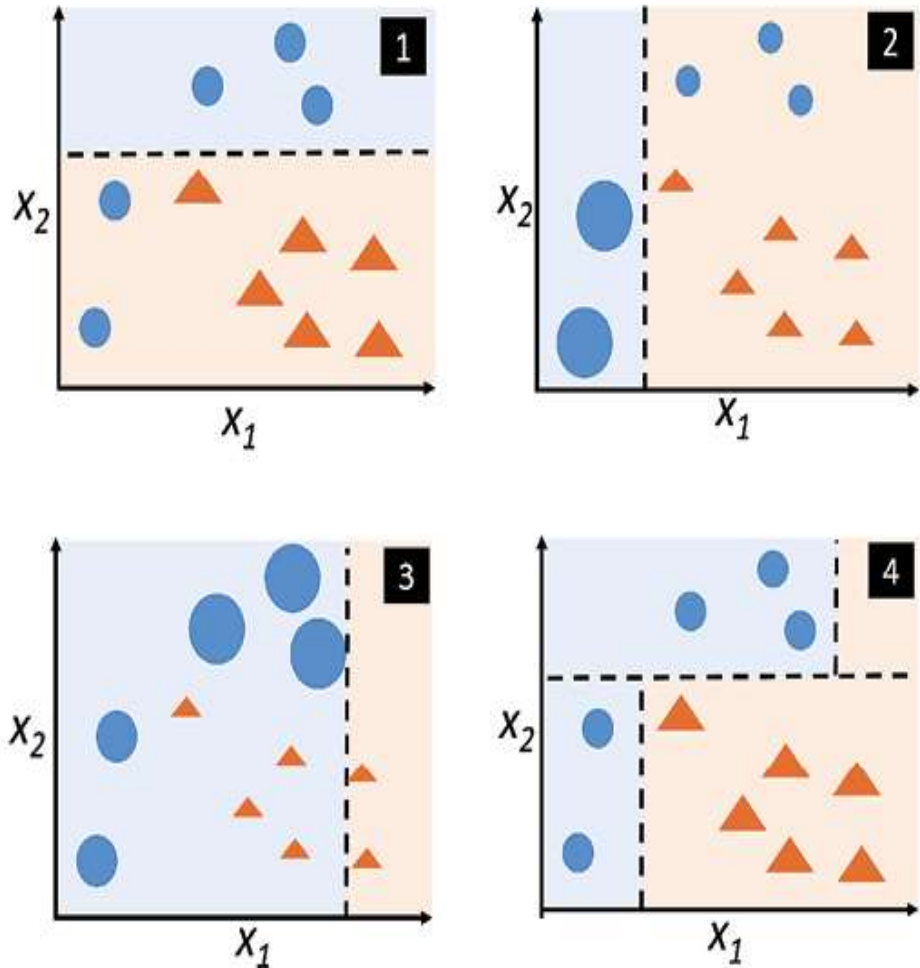
Boosting



Boosting - AdaBoost

- The main idea behind AdaBoost (Adaptive Boosting) is to combine the predictions of multiple weak learners @ stump (simple models that perform slightly better than random guessing) to create a strong classifier.
- The basic concept behind Adaboost is to set the weights of classifiers and train the data sample in each iteration such that it ensures accurate predictions of unusual observations.
- Adaboost should meet two conditions:
 - The classifier should be trained interactively on various weighed training examples.
 - In each iteration, it tries to provide an excellent fit for these examples by minimising training errors.

Boosting - AdaBoost



1. The 1st classifier classifies the points (with some misclassified points).
2. A 2nd classifier is trained with a new training data set having more weights assigned to the misclassified points and lesser weights to correctly classified points.
3. The 3rd classifier gets retrained with new training datasets with weights for each data point updated.
4. Finally, the ensemble adaptive boosting classifier is constructed by ensembling three classifiers constructed/fitted / trained using different training datasets created as a result of adaptive resampling.

Boosting - AdaBoost

```
from sklearn.model_selection import GridSearchCV
# Define parameter grid for grid search
param_grid = {
    'n_estimators': [50, 100, 150],
    'learning_rate': [0.01, 0.1, 0.3, 0.5, 0.8, 1.0],
    'base_estimator': [DecisionTreeClassifier(criterion='entropy',max_depth=1),
                       DecisionTreeClassifier(criterion='entropy',max_depth=2),
                       DecisionTreeClassifier(criterion='gini',max_depth=1),
                       DecisionTreeClassifier(criterion='gini',max_depth=2)],
}

# Create an AdaBoost classifier with 100 estimators
ada_clf = AdaBoostClassifier()

# Perform grid search with cross-validation
grid_search = GridSearchCV(ada_clf, param_grid, cv=5)
grid_search.fit(X_train, y_train)
```

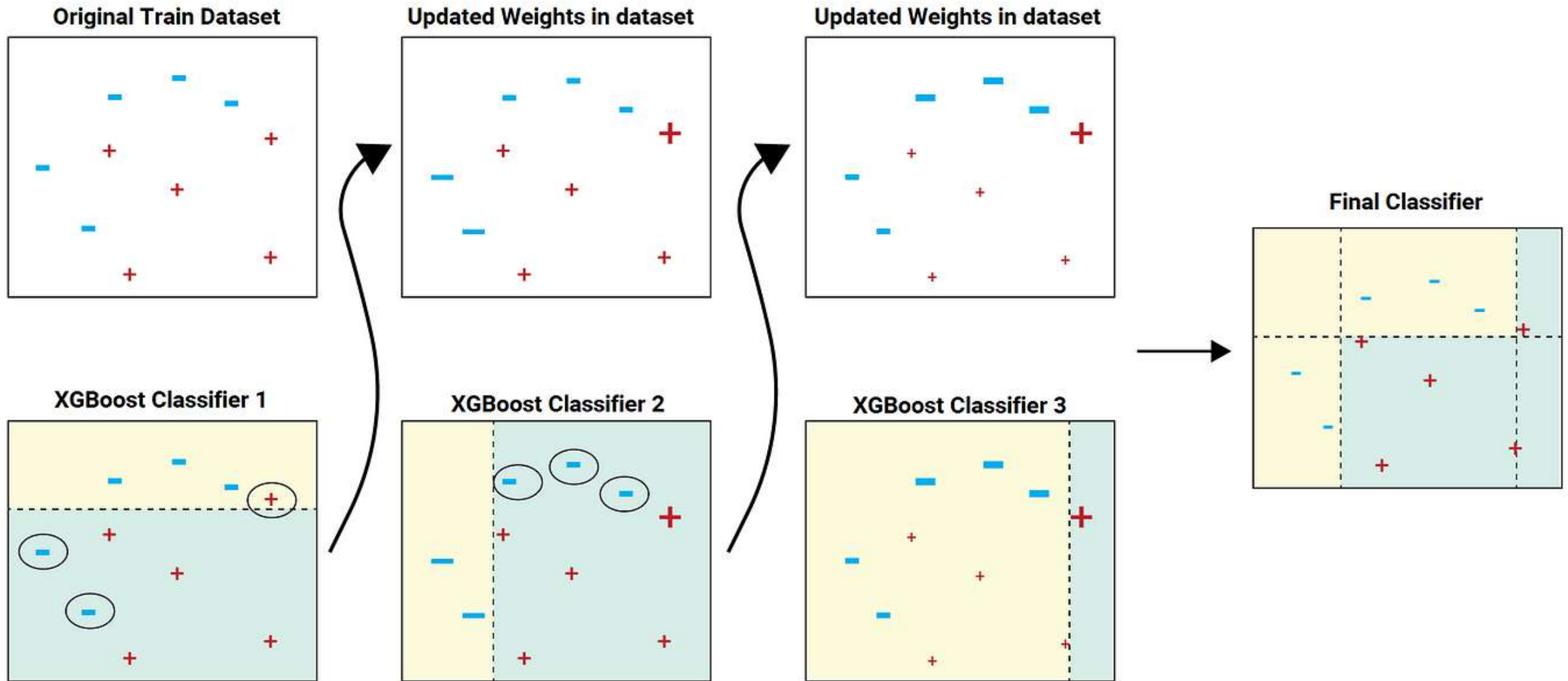
Grid Search hyperparameter ~

- *number of estimators* - The maximum number of estimators at which boosting is terminated. In case of perfect fit, the learning procedure is stopped early (init = 50)
- *learning rate* - A higher learning rate increases the contribution of each classifier (default = 1)
- *base model* - The base estimator from which the boosted ensemble is built. Default – decision tree (max_depth=1).

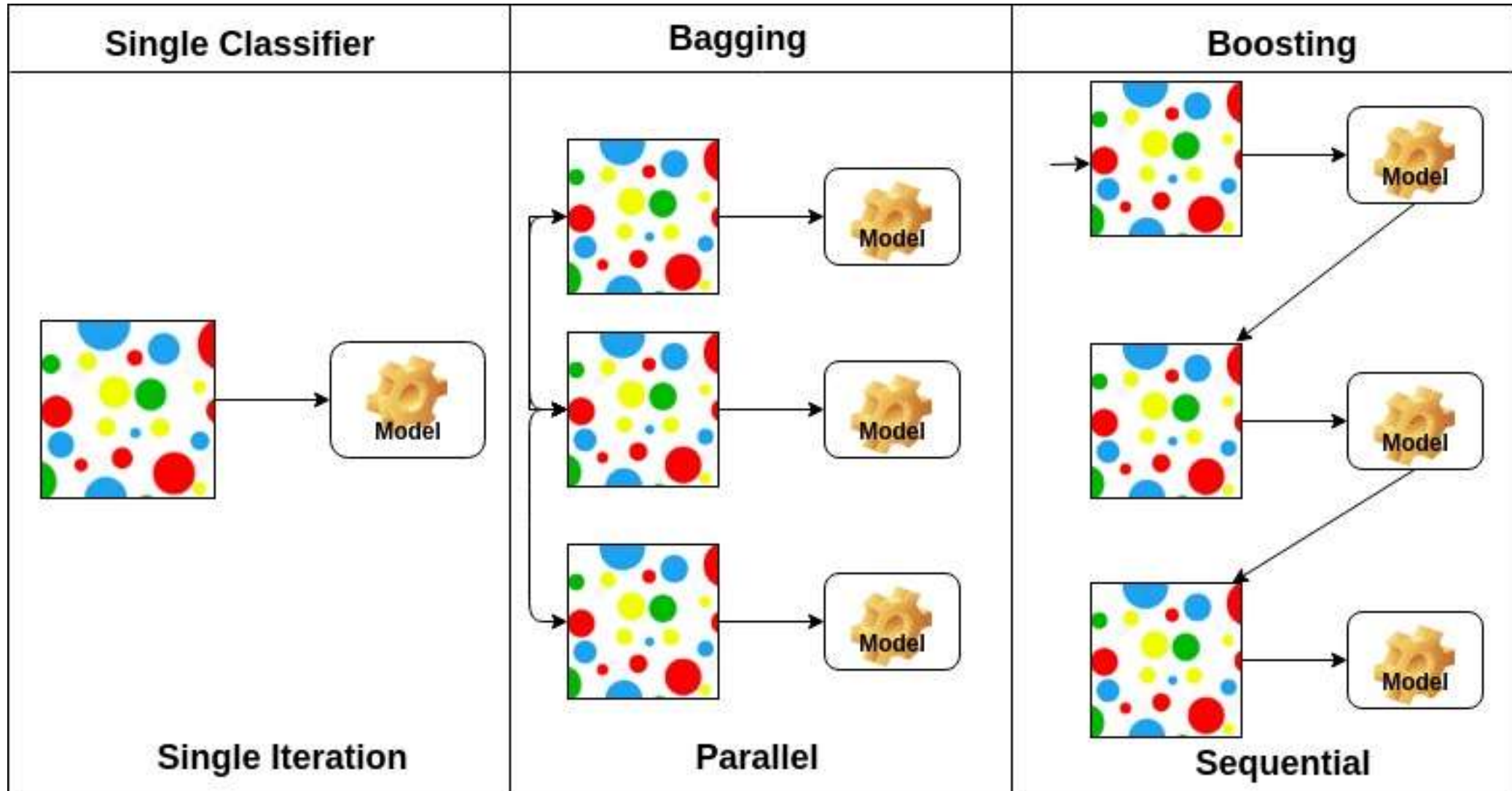
Boosting – XGBoost

- XGBoost, which stands for "eXtreme Gradient Boosting," is a powerful and efficient open-source implementation of the gradient boosting algorithm.
- Compared to the AdaBoost, the weights of the training instances are not tweaked, instead, each predictor is trained using the residual errors of the predecessor as labels.
- Features:
 - **Regularization:** XGBoost includes regularization terms in its objective function, which helps prevent overfitting.
 - **Handling Missing Data:** XGBoost can automatically handle missing values in the input data.
 - **Parallel and Distributed Computing:** It supports parallel and distributed computing, making it scalable and suitable for large datasets.
 - **Tree Pruning:** XGBoost uses a technique called "pruning" during the tree-building process.

Boosting – XGBoost



Boosting vs. Bagging

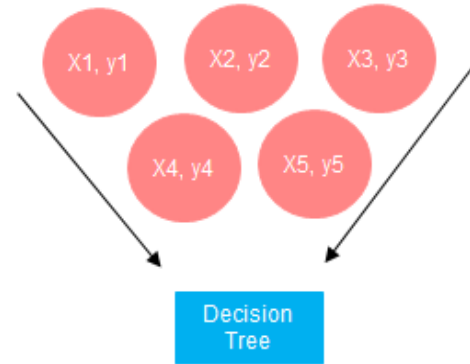


Boosting – XGBoost

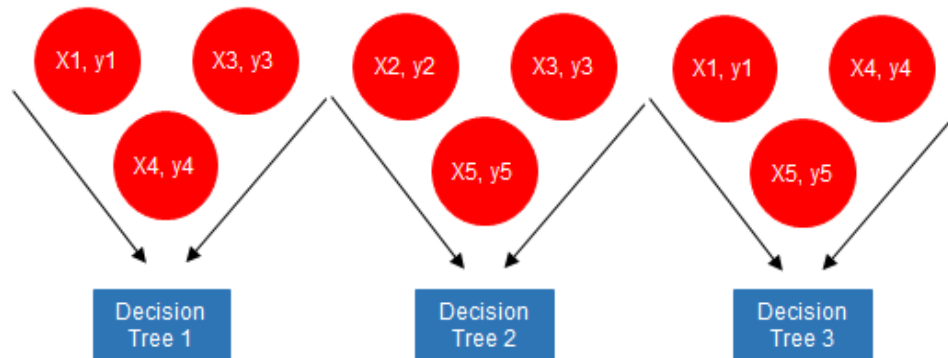
- XGBoost, which stands for "eXtreme Gradient Boosting," is a powerful and efficient open-source implementation of the gradient boosting algorithm.
- Compared to the AdaBoost, the weights of the training instances are not tweaked, instead, each predictor is trained using the residual errors of the predecessor as labels.
- Features:
 - **Regularization:** XGBoost includes regularization terms in its objective function, which helps prevent overfitting.
 - **Handling Missing Data:** XGBoost can automatically handle missing values in the input data.
 - **Parallel and Distributed Computing:** It supports parallel and distributed computing, making it scalable and suitable for large datasets.
 - **Tree Pruning:** XGBoost uses a technique called "pruning" during the tree-building process.

Boosting vs. Bagging

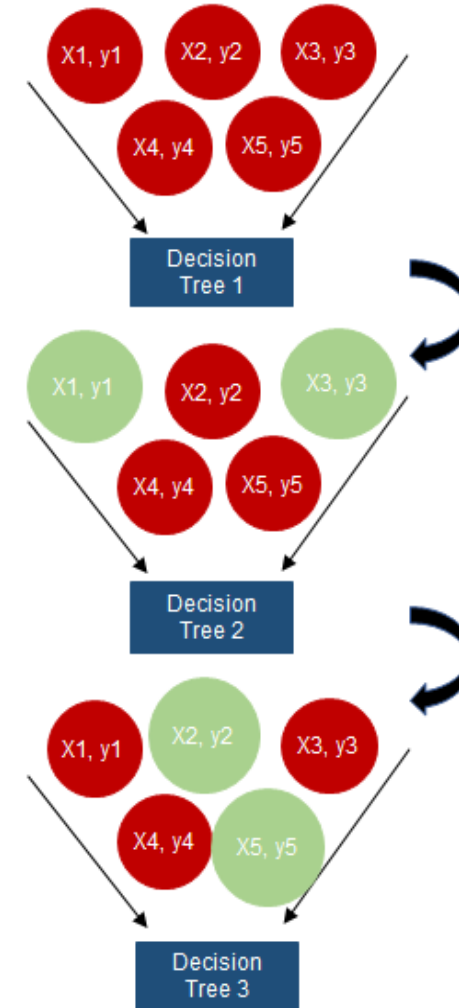
Single decision tree iteration: All samples



Bagging: Parallel tree growing with subsamples

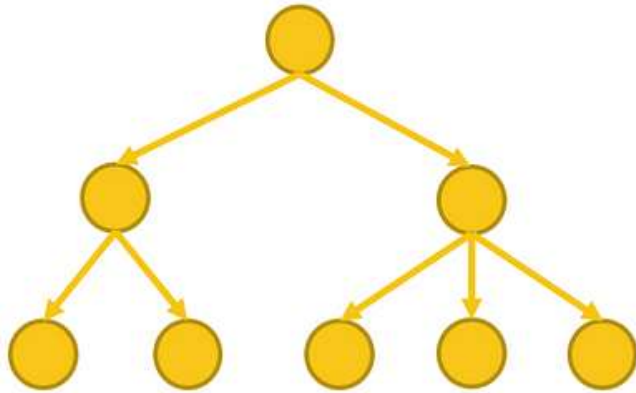


Boosting: Sequential tree growing with weighted samples

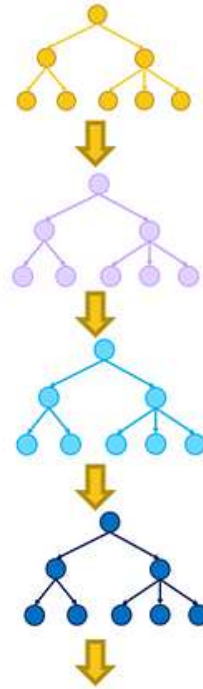


Boosting vs. Bagging

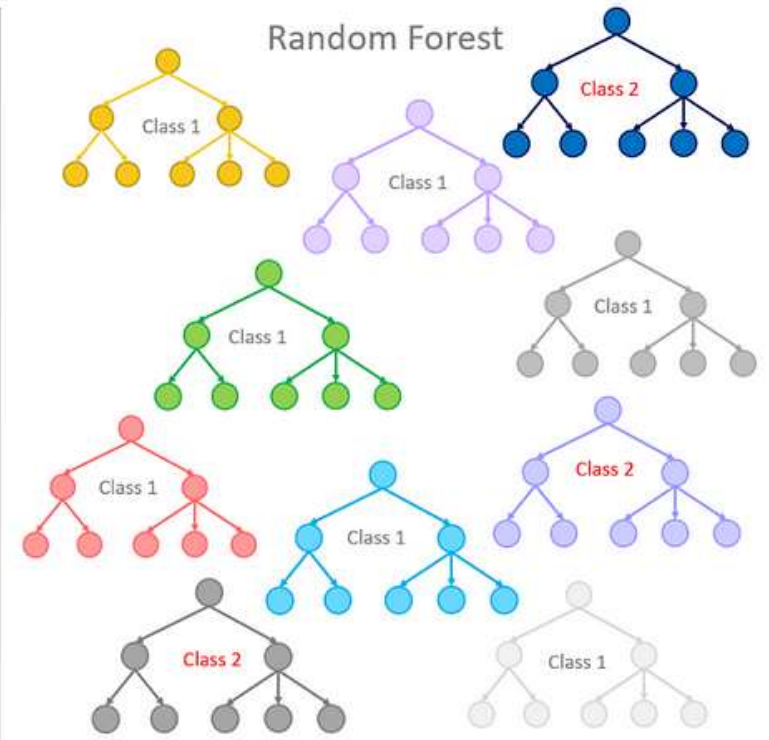
Single Decision Tree



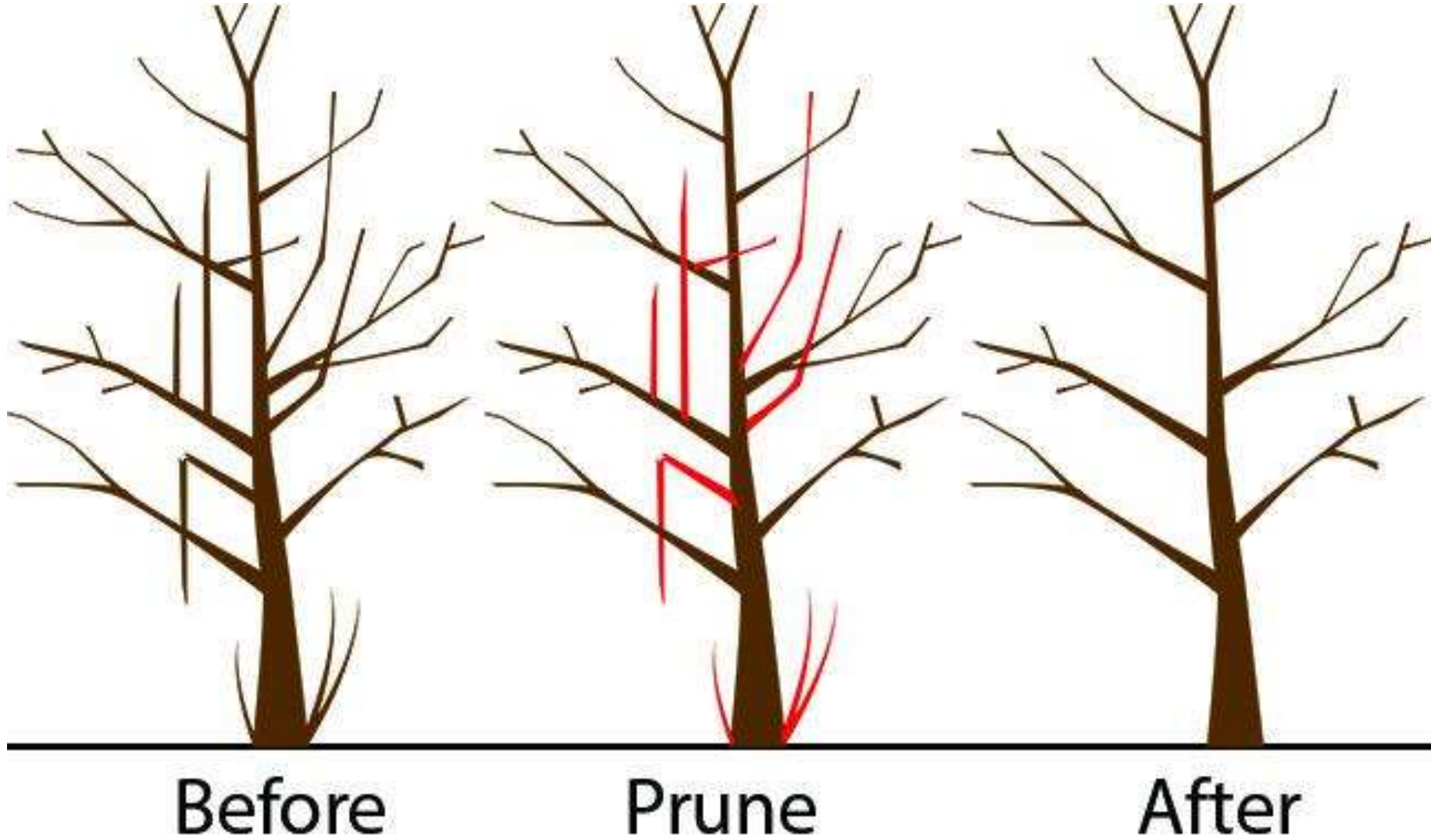
Gradient Boosted Trees



Random Forest

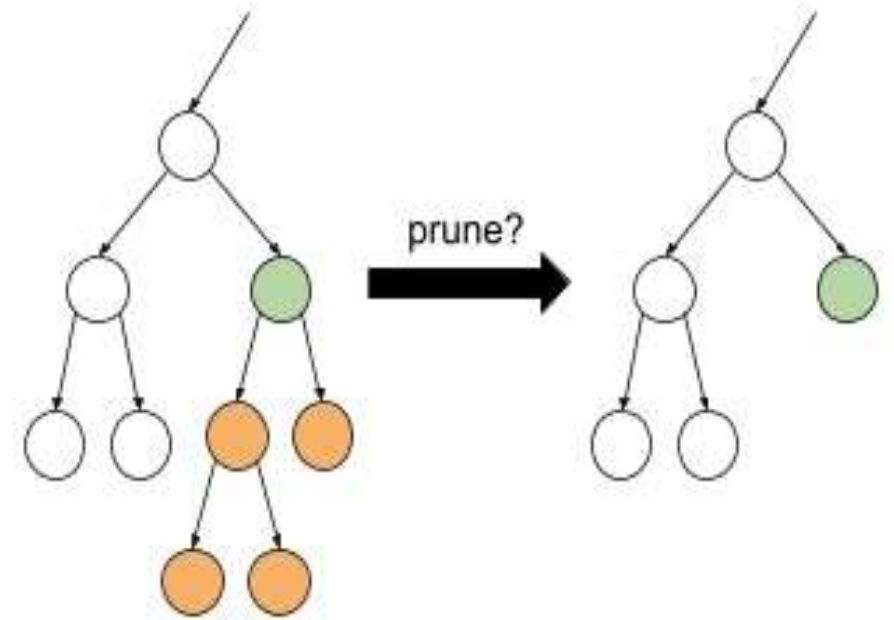


Tree Pruning



Tree Pruning

- Pruning is a technique that removes the parts of the Decision Tree which prevent it from growing to its full depth.
 - The parts that it removes from the tree are the parts that do not provide the power to classify instances.
 - A Decision tree that is trained to its full depth will highly likely lead to overfitting the training data - therefore Pruning is important.
- Aims to construct an algorithm that will perform worse on training data but will generalize better on test data.
- Methods – Pre / Post Pruning



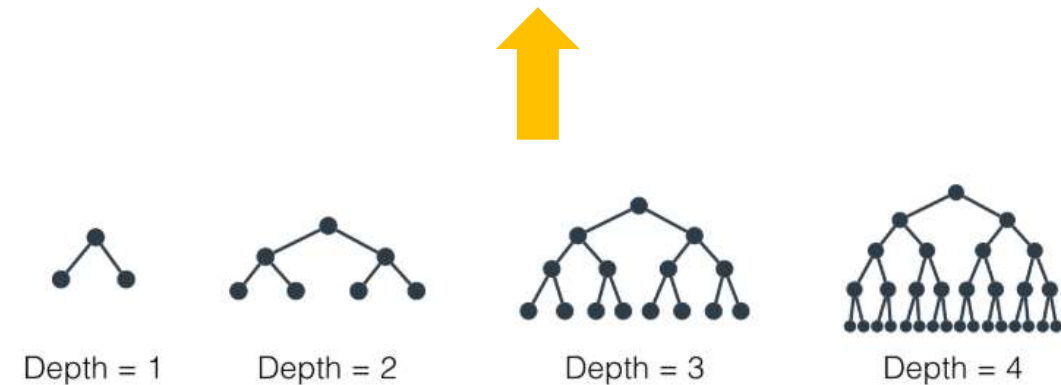
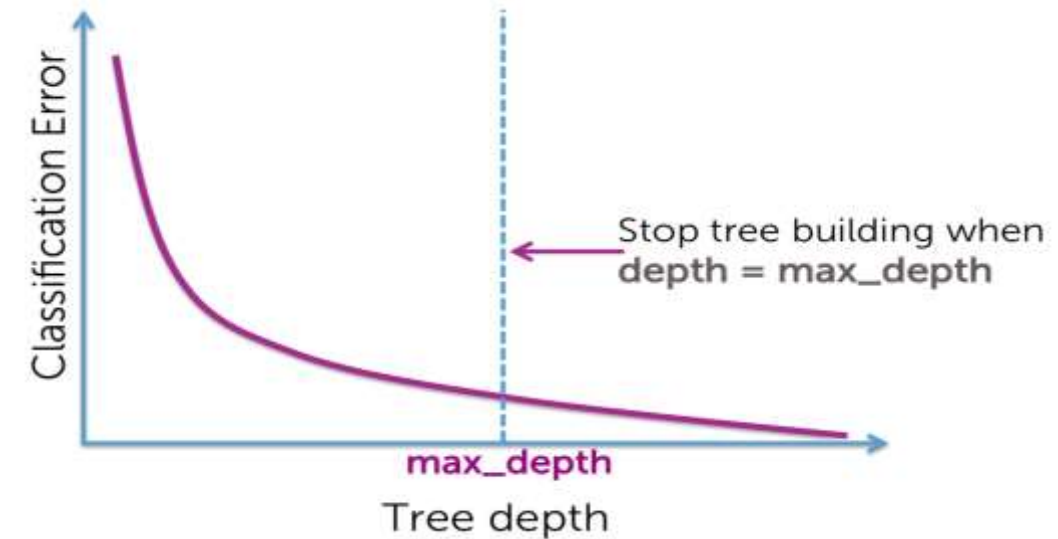
Tree Pruning (Pre-Pruning)

- Pre-pruning is a critical step in the decision tree construction process because it allows the model to find a balance between complexity and accuracy.
- By stopping the tree from growing too much, pre-pruning helps create a simpler model that is more likely to generalize well to new, unseen data.
 - **Maximum Depth (Depth Limit):** Limit the maximum depth of the decision tree. The tree stops growing once it reaches the specified depth (reducing complexity & overfitting)
 - **Minimum Samples per Leaf:** Set a threshold for the minimum number of samples required to create a leaf node.
 - **Minimum Samples per Split:** Like the minimum samples per leaf but applied to internal nodes.
- However, you should be cautious as early stopping can also lead to underfitting.

Tree Pruning (Pre-Pruning)

Maximum Depth (Depth Limit):

- Limiting the maximum depth of the decision tree restricts the number of levels or layers it can have.
- If a tree is allowed to grow without any constraints on depth, it might become overly complex and capture noise or outliers in the training data that do not generalize well to new data.
- Setting a maximum depth helps control the complexity of the tree and prevents it from becoming too specific to the training data (overfitting).

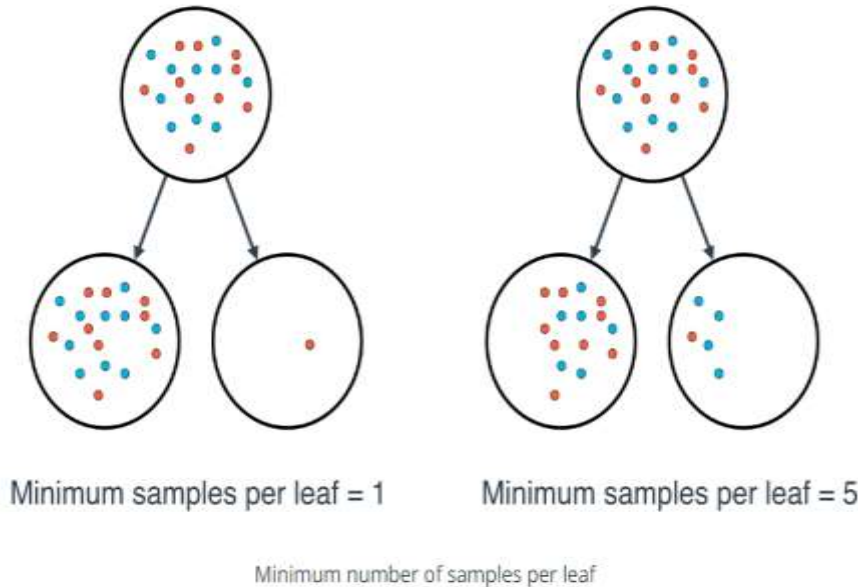


Maximum depth of a decision tree

Tree Pruning (Pre-Pruning)

Minimum Samples per Leaf:

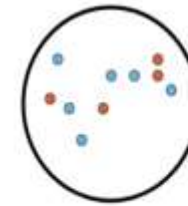
- represents the minimum number of samples needed to form a leaf node.
- Consider a node with 5 samples:
 - It can be divided into two leaf nodes with sizes of 2 and 3 respectively.
 - However, if *min. sample leaf* = 3 is greater than or equal to 3, the node will not be split since the minimum leaf size is 3 and a new node with only 2 samples cannot be created.
- Note: the lower values should be chosen for imbalanced class problems as the regions in which the minority class will be in the majority will be of small size.



Tree Pruning (Pre-Pruning)

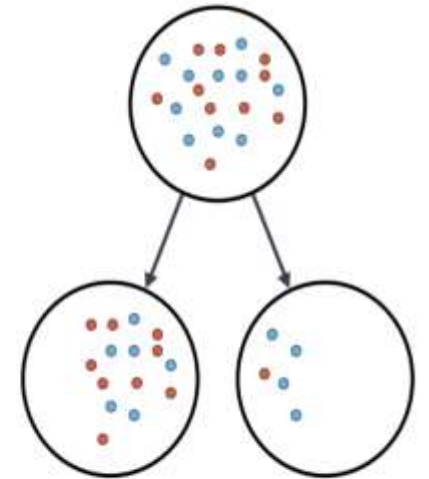
Minimum Samples per Split:

- Similar to the minimum samples per leaf, this criterion applies to internal nodes (non-leaf nodes) in the tree.
 - It sets a threshold β , for the minimum number of samples required to perform a split at a particular node.
 - If the number of samples at a node $< \beta$, the split is not attempted.
 - This helps control the granularity of the tree, preventing it from splitting based on small subsets of the data that may not be representative.



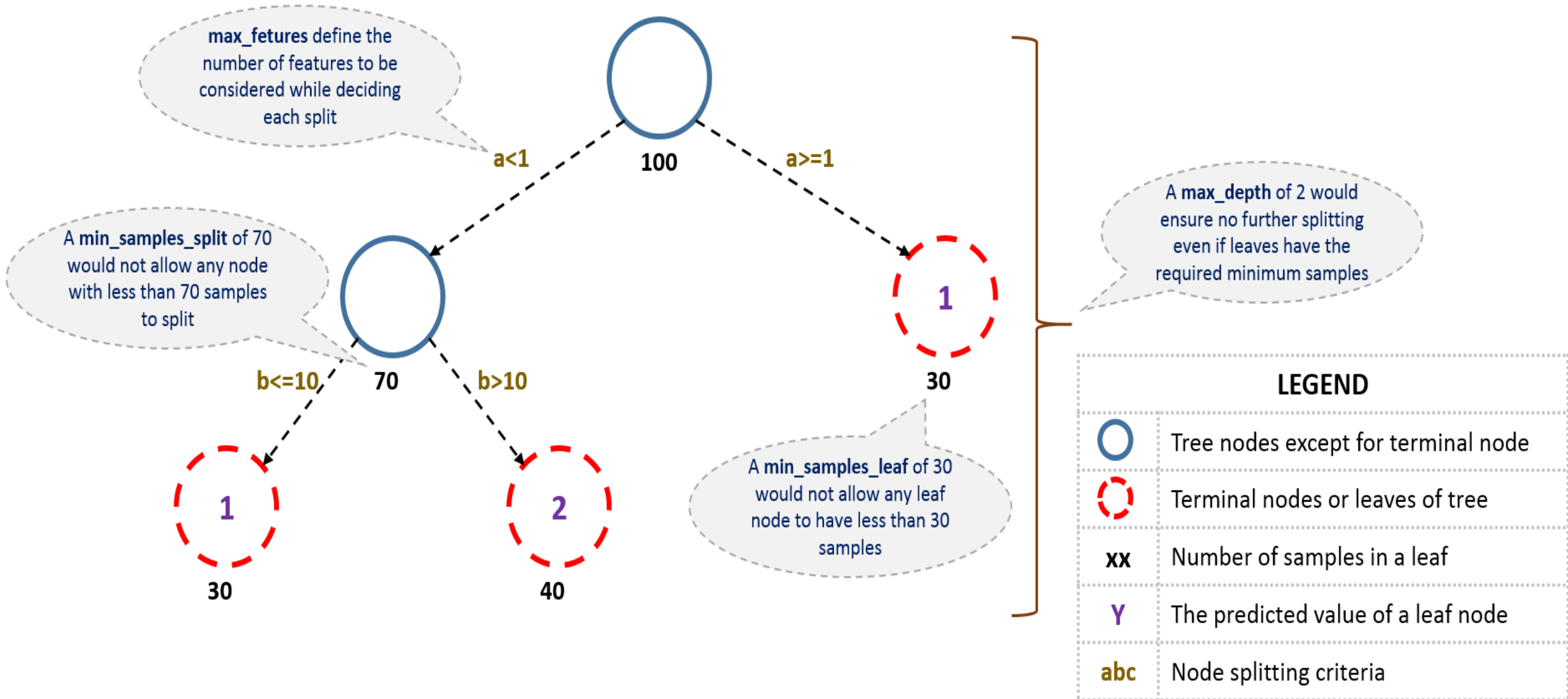
No split!

Minimum number of samples to split = 11



Minimum number of samples to split = 11

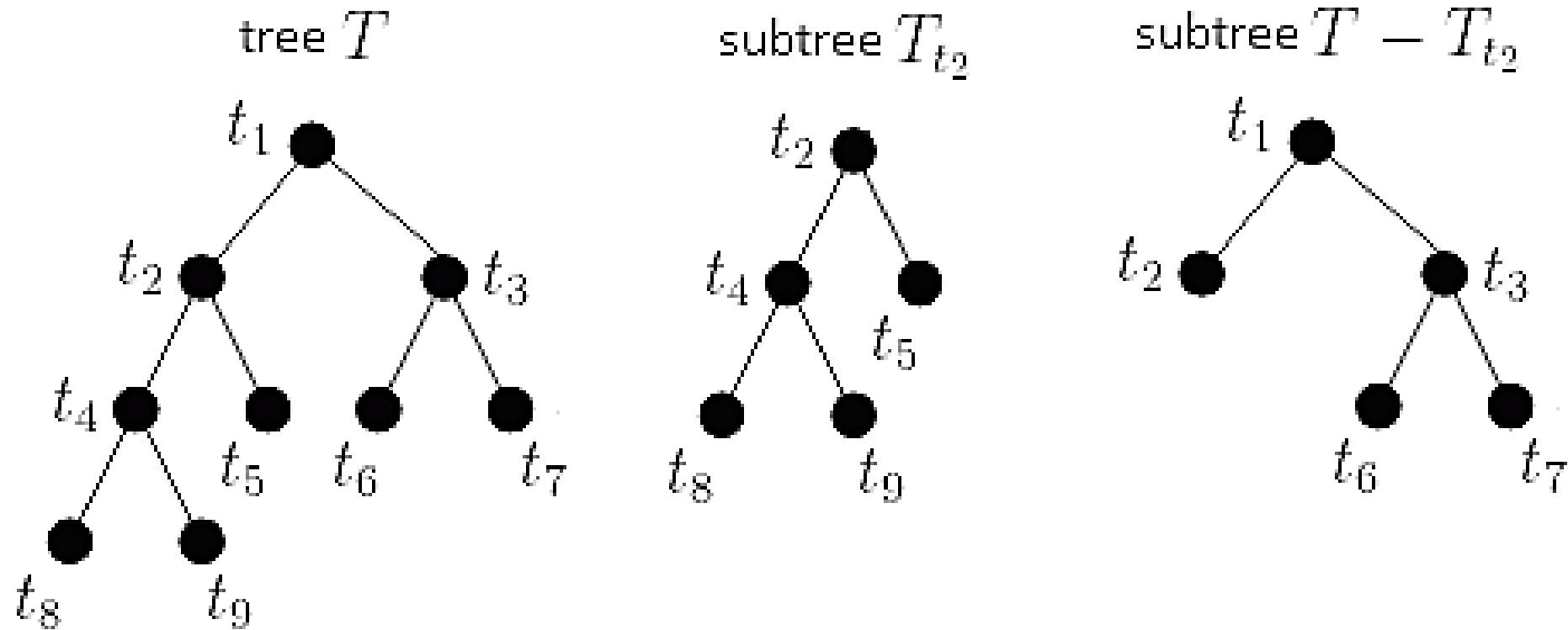
Tree Pruning (Pre-Pruning)



Tree Pruning (Post-Pruning)

- Post-pruning, as the name suggests, involves first building the tree to its entirety and then removing nodes that do not provide significant predictive power.
- Cost-complexity pruning is a common method for post-pruning decision trees.
- It involves assigning a cost to each subtree in the fully grown tree and then selecting the subtree with the smallest cost as the pruned tree.
 - The cost of a subtree is determined by a complexity parameter (often denoted as α) and the number of leaf nodes in the subtree.
- Process:
 - **Calculate Cost for Subtrees:** For each subtree, calculate the total error, E (such as misclassification rate) on the validation data and add a complexity penalty based on the number of leaf nodes and the complexity parameter, $\alpha \geq 0$ (as α increases, the more we are penalised for having larger trees.).
 - $\text{Cost} = E + \alpha * (\text{Number of Leaf Nodes})$
 - **Select the Best Subtree:** Choose the subtree with the smallest cost as the pruned tree.

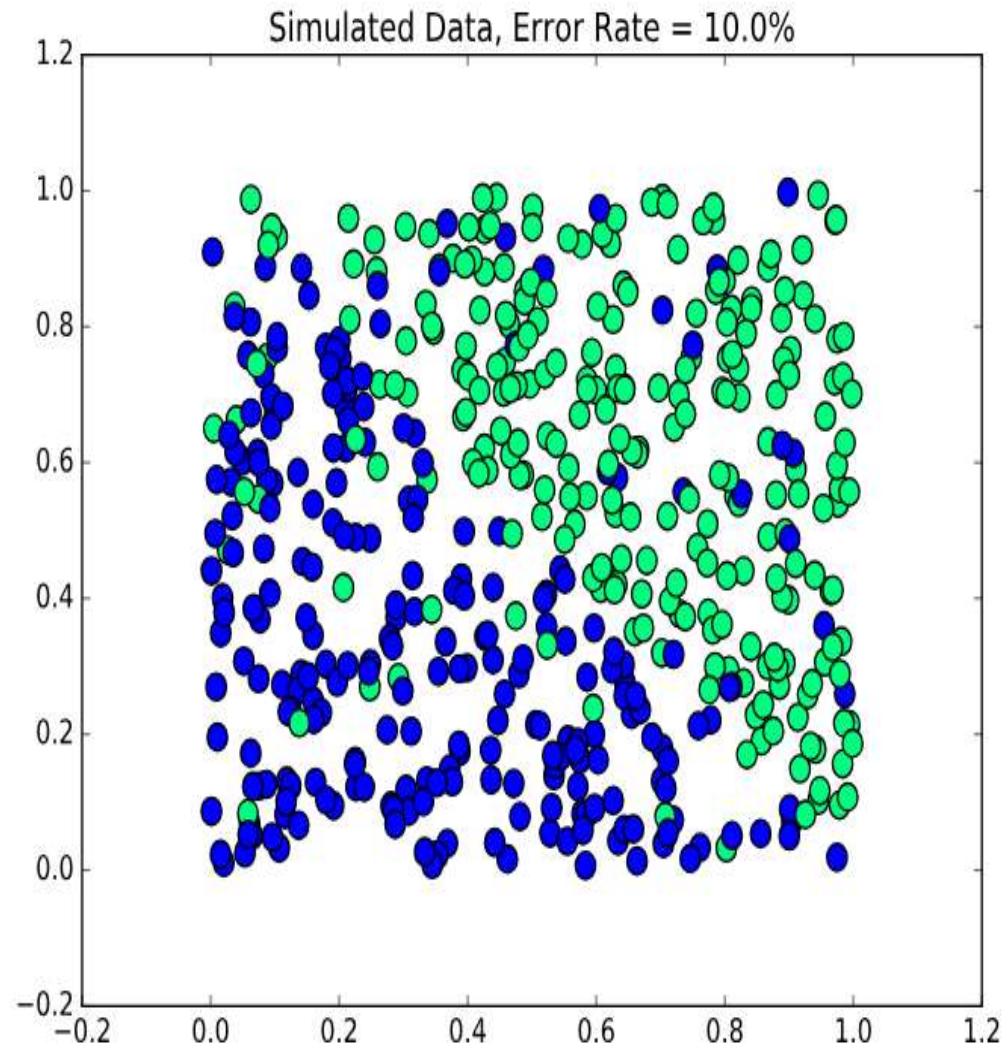
Tree Pruning (Post-Pruning)



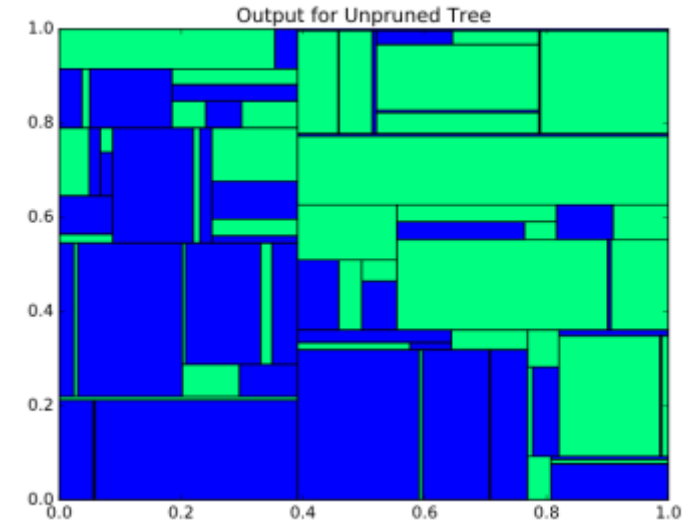
Tree Pruning (Post-Pruning)

- The more leaf nodes that the tree contains the higher complexity of the tree because we have more flexibility in partitioning the space into smaller pieces, and therefore more possibilities for fitting the training data.
- There's also the issue of how much importance to put on the size of the tree.
 - The complexity parameter α adjusts that.
- In the end, the cost complexity measure comes as a penalized version of the resubstituting error rate. This is the function to be minimised when pruning the tree.
- Which subtree is selected eventually depends on α .
 - If $\alpha = 0$ then the biggest tree will be chosen because the complexity penalty term is essentially dropped.
 - As α approaches infinity, the tree of size 1, i.e., a single root node, will be selected.

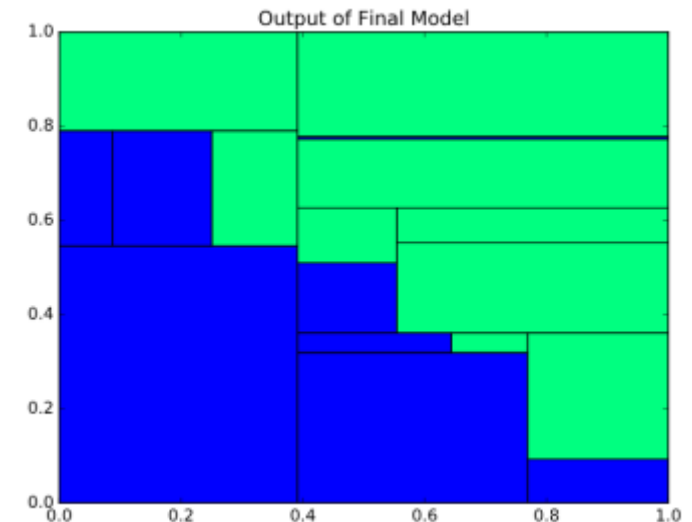
Tree Pruning (Post-Pruning)



*Without
pruning*



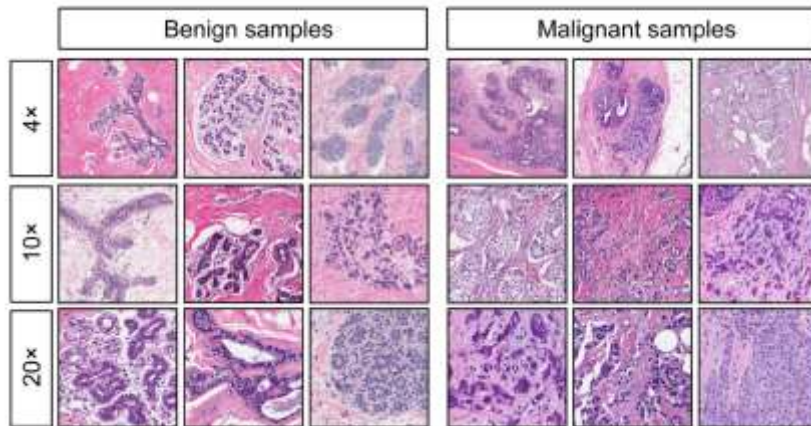
*With
pruning*



Tree Pruning (Post-Pruning)

Case Study – Breast Cancer Dataset

- Breast cancer is one of the most common diseases in women aged 40 to 59 years with multiple associated risk factors: genetic, environmental and behavioural factors, characterized by the disordered proliferation and constant growth of cells in this organ.



#	Attribute Name	Possible Value
1	Case ID	Id Number
2	Clump Thickness	1 - 10
3	Uniformity of Cell Size	1 - 10
4	Uniformity of Cell Shape	1 - 10
5	Marginal Adhesion	1 - 10
6	Single Epithelial Cell Size	1 - 10
7	Bare Nuclei	1 - 10
8	Bland Chromatin	1 - 10
9	Normal Nuclei	1 - 10
10	Mitoses	1 - 10
11	Class	(2 for Benign, 4 for Malignant)

Tree Pruning (Post-Pruning)

```
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
```

```
X, y = load_breast_cancer(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

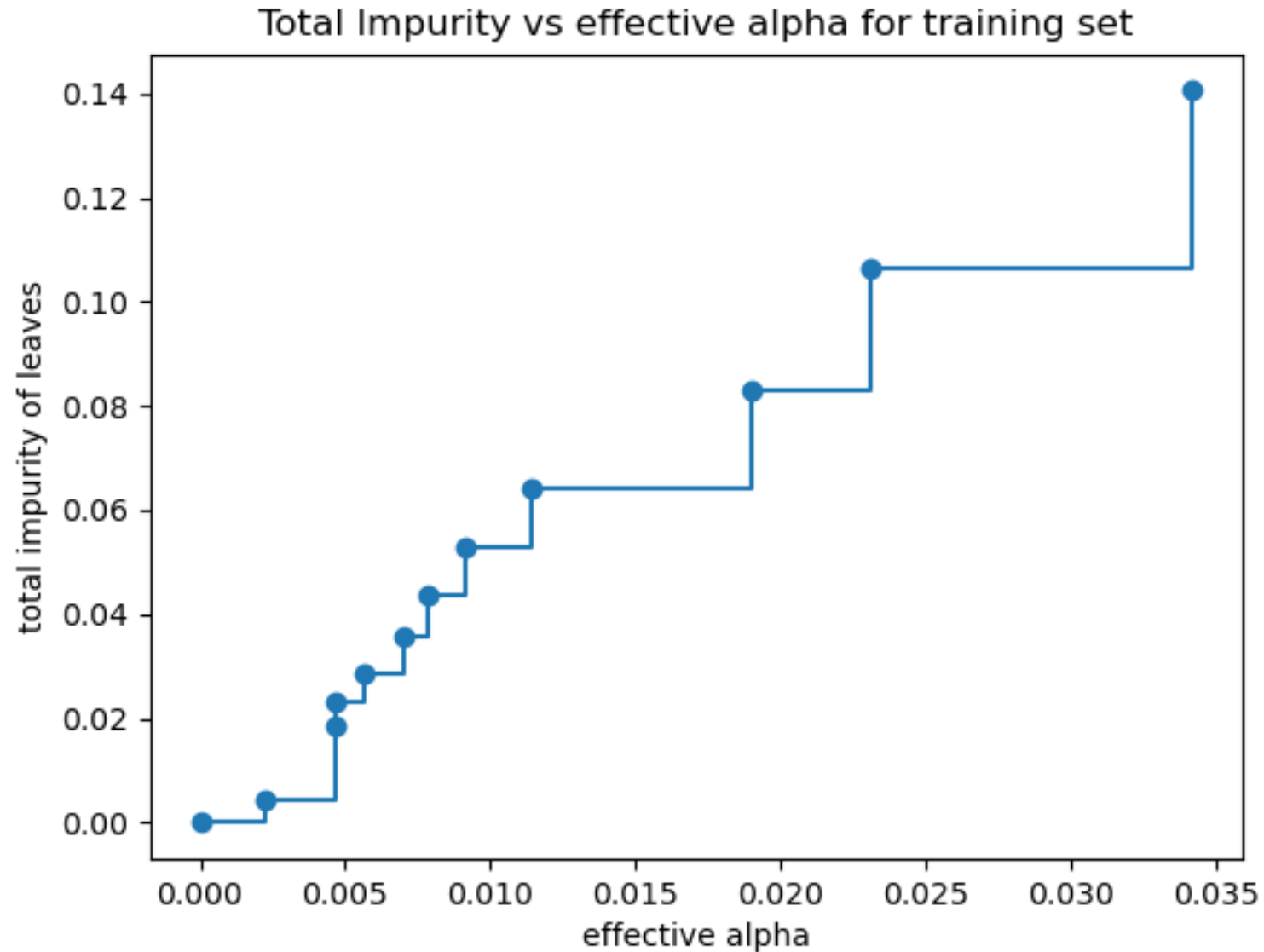
```
clf = DecisionTreeClassifier(random_state=0)
path = clf.cost_complexity_pruning_path(X_train, y_train)
ccp_alphas, impurities = path.ccp_alphas, path.impurities
```



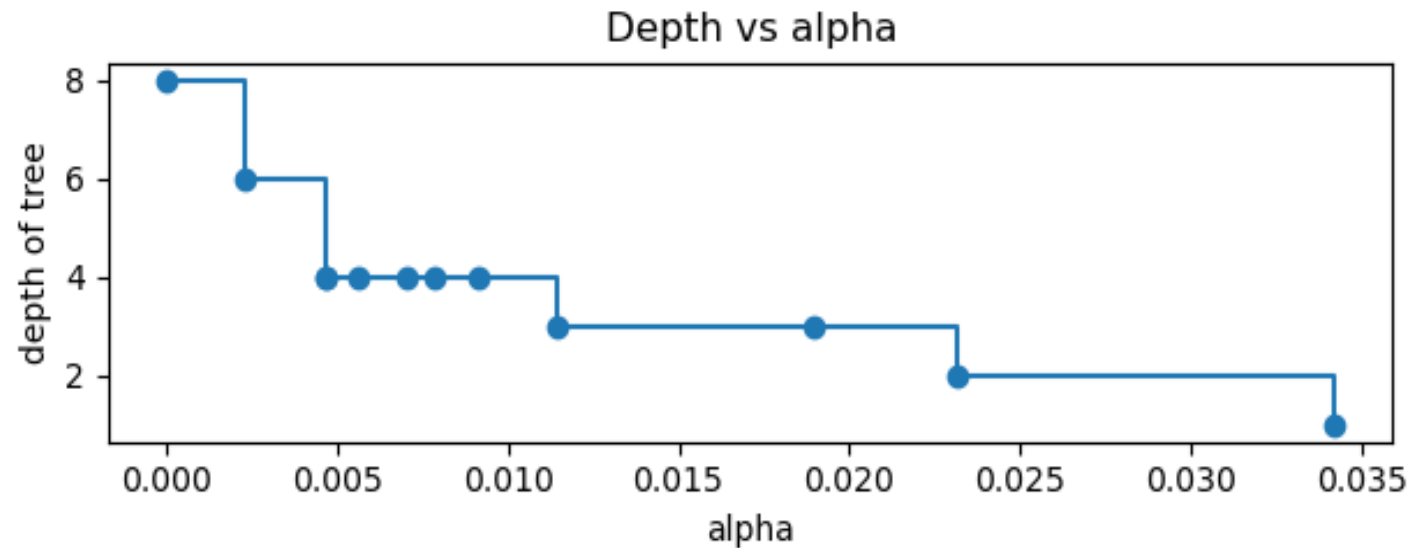
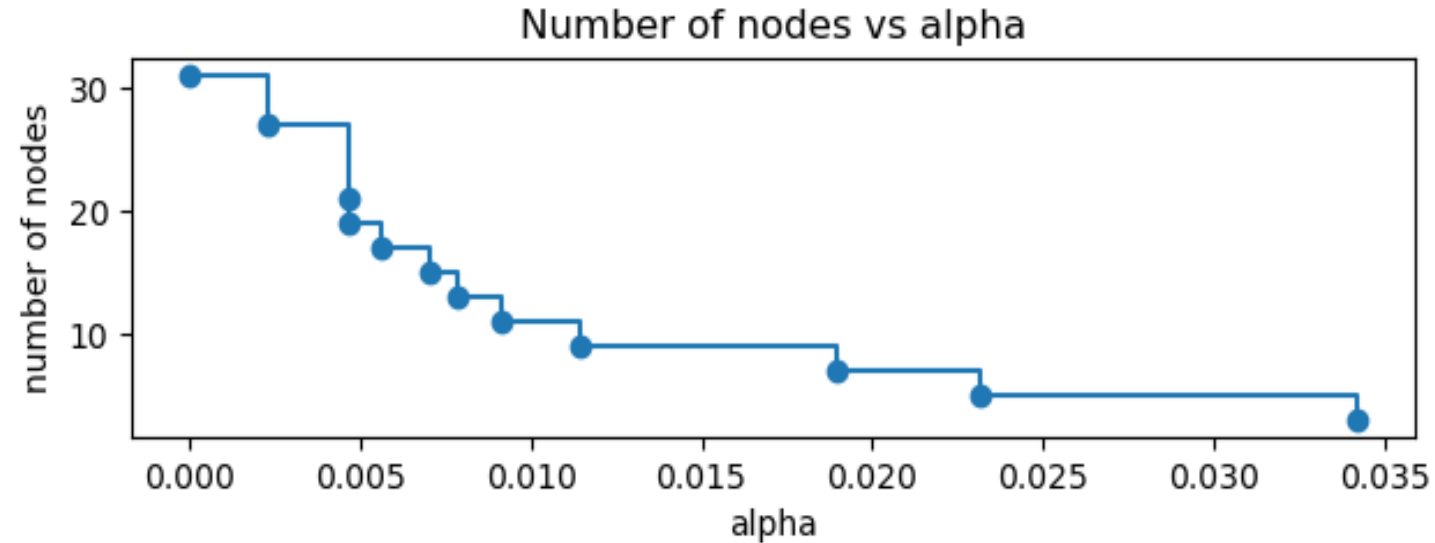
<classifier>.

cost_complexity_pruning_path(X_train, y_train)

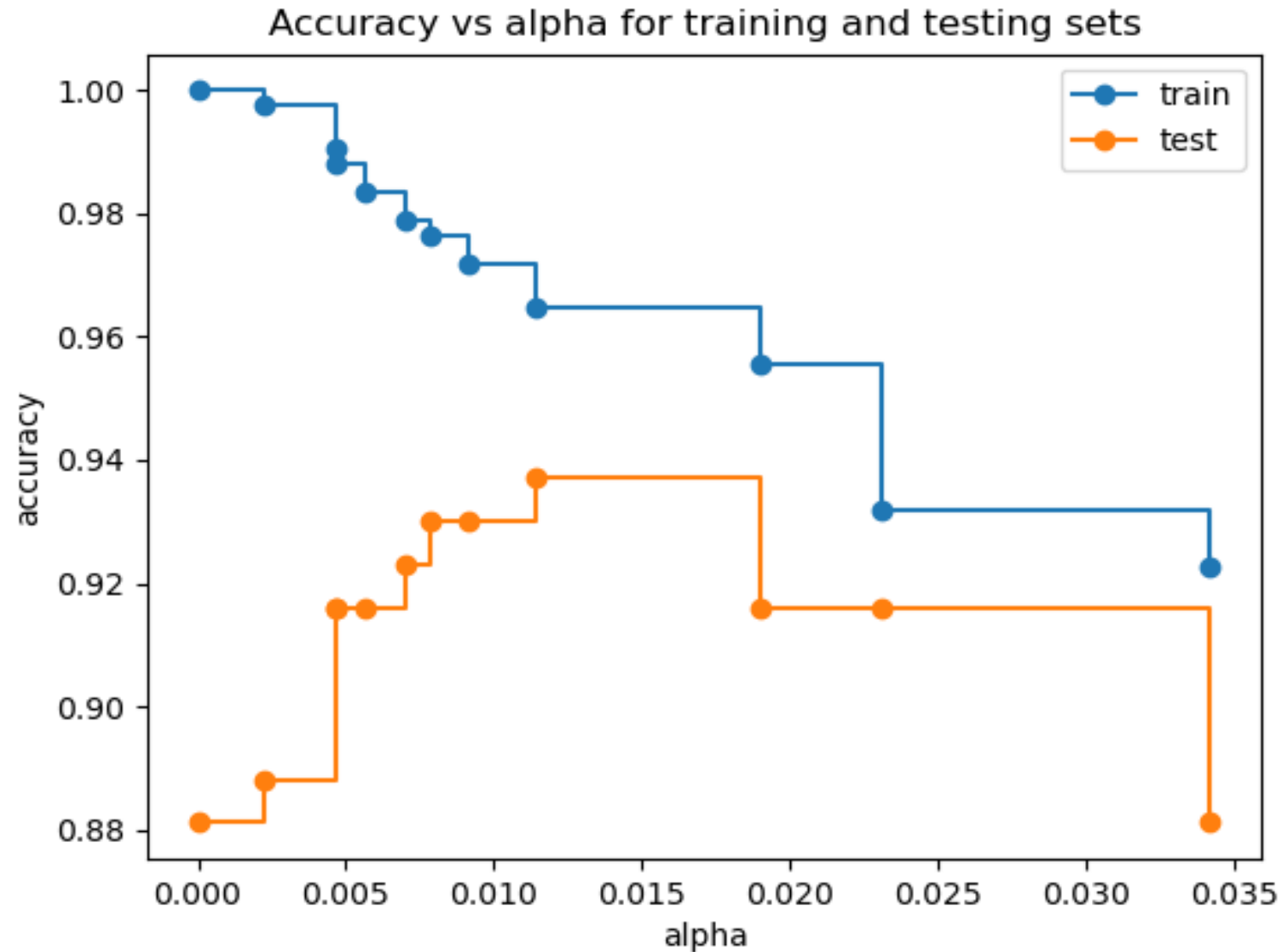
Tree Pruning (Post-Pruning)



Tree Pruning (Post-Pruning)



Tree Pruning (Post-Pruning)



Next

**Adaptive Pattern
Recognition Techniques**