KUMARAGURU
college of technology
character is life

ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

LABORATORY RECORD

U18AII3204T - Applied Machine Learning

2022-2023 (ODD)

KUMARAGURU
college of technology
character is life

Certificate

This is to certify that it is a Bonafide record of practical work done by Sri/Kum. _____ bearing the Register No._____of_____year_____ _____branch in the _____ Laboratory during the academic year _____ under our supervision.

Faculty in charge

INTERNAL ASSESSMENT

Ex. No	Title	Data Visualization and preprocessing (5)	Model Building & Evaluation (10)	Report (5)	Total (20)	Viva (10)
1	Regression					
2	Classification					
				Total		

Laboratory Mark

Internal Assessment	Viva (10)	Record (20)	Model Exam (20)	Total (50)

Staff in Charge

Ex No:1	Multivariable Regression and Gradient Descent

Aim:

To implement multivariable regression to the amazon dataset to predict the price of the books based on the user rating, reviews and year and optimize using gradient descent.

Multivariable linear regression:

Multivariable regression models are used to establish the relationship between a dependent variable and more than 1 independent variable. Multivariable regression can be used for a variety of different purposes in research studies.

The general model for multivariable regression is as follows:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots \beta_n x_n$$

y is the dependent variable and $x_1, x_2 \ldots x_n$ are multiple independent variables.

Multivariable Regression is an extension of Simple Linear regression as it takes more than one predictor variable to predict the response variable. For Multivariable regression, the dependent or target variable(Y) must be the continuous/real, but the predictor or independent variable may be of continuous or categorical. Multivariable regression tries to fit a regression line through a multidimensional space of data-points.

Feature Importance: The Feature is selected based on p-value and adjusted R-squared value.

Cost function: The cost function is defined as the measurement of difference or error between actual values and expected values at the current position and present in the form of a single real number. Mean Squared error is used as the cost function.

Mean Squared Error

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

Gradient Descent:

Gradient Descent is known as one of the most commonly used optimization algorithms to train machine learning models by means of minimizing errors between actual and expected results. Further, gradient descent is also used to train Neural Networks. In mathematical terminology, Optimization algorithm refers to the task of minimizing/maximizing an objective function f(x) parameterized by x. Similarly, in machine learning, optimization is the task of minimizing the cost function parameterized by the model's parameters. The main objective of gradient descent is to minimize the convex function using iteration of parameter updates. Once these machine learning models are optimized, these models can be used as powerful tools for Artificial Intelligence and various computer science applications.

Dataset:

Amazon bookselling dataset

Source: Kaggle

Independent variables	Dependent variable
▪ User Rating ▪ Year ▪ Reviews	▪ Price

Train data: 70%

Test data: 30%

Implementation of Multivariable regression:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
dataset=pd.read_csv("amazon.csv")
```

```
import statsmodels.api as sm
import statsmodels.formula.api as smf
dataset.shape
dataset.drop(['Name', 'Author', 'Genre'], axis=1)
import statsmodels.formula.api as smf
dataset.rename(columns={'User Rating':'Userrating'},inplace=True)
dataset.head()
model1 = smf.ols(formula='Price ~ Userrating', data=dataset).fit()
print(model1.summary())
model2 = smf.ols(formula='Price ~ Reviews', data=dataset).fit()
print(model2.summary())
model3 = smf.ols(formula='Price ~ Year', data=dataset).fit()
print(model3.summary())
model4 = smf.ols(formula='Price ~ Reviews+Userrating', data=dataset).fit()
print(model4.summary())
model5 = smf.ols(formula='Price ~ Reviews+Year', data=dataset).fit()
print(model5.summary())
model6 = smf.ols(formula='Price ~ Userrating+Year', data=dataset).fit()
print(model6.summary())
model7 = smf.ols(formula='Price ~ Reviews + Userrating + Year', data=dataset).fit()
print(model7.summary())
 from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
X=dataset.loc[0:700,['Userrating','Year']]
Y=dataset.loc[0:700,['Price']]
X_train,X_test,Y_train,Y_test=train_test_split(X,Y,train_size=0.70,random_state=0)
regressor=LinearRegression()
regressor.fit(X_train,Y_train)
Y_pred=regressor.predict(X_test)
from sklearn.metrics import mean_squared_error
print(mean_squared_error(Y_pred,Y_test))
sns.regplot(x='Price',y='Userrating',data=dataset,color='red')
plt.scatter(Y_test,Y_pred)
plt.xlabel('Actual')
plt.ylabel('Predicted')
```

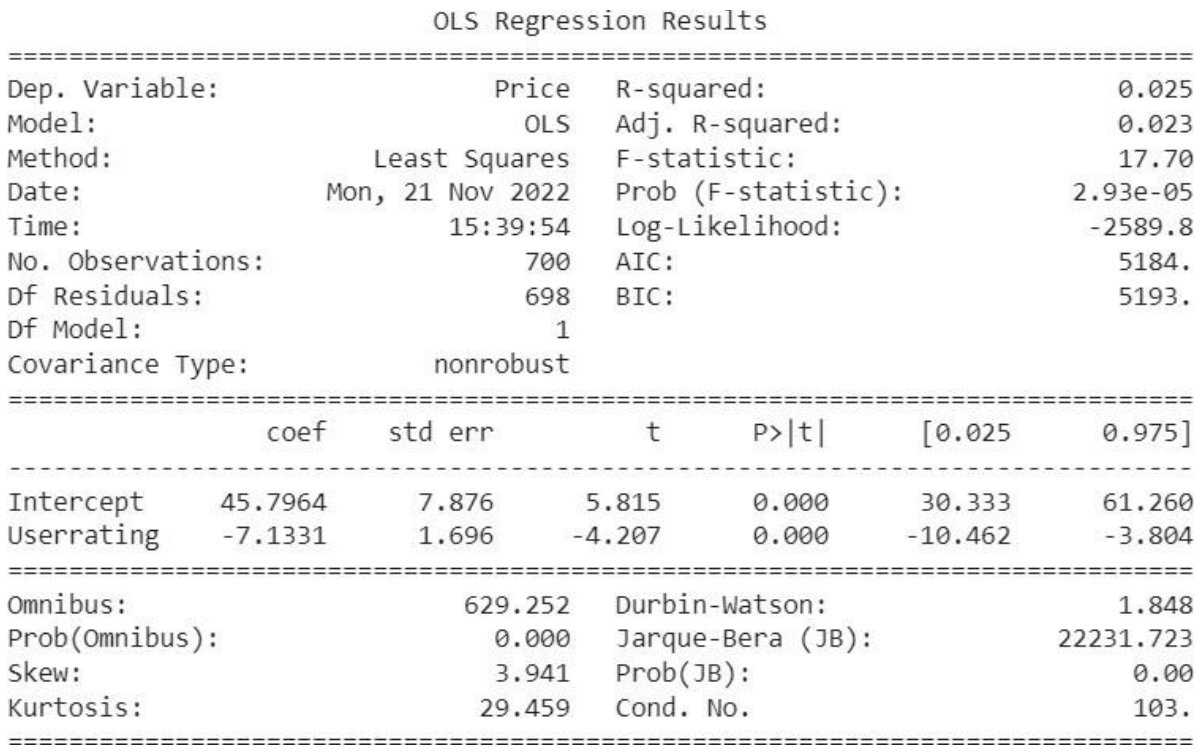
Implementation of Gradient Descent

```
m,n = X.shape
from mpl_toolkits.mplot3d import Axes3D
def cost_function(X, y, theta):
    m = y.size
    error = np.dot(X, theta.T) - y
    cost = 1/(2*m) * np.dot(error.T, error)
    return cost, error
```

```
def gradient_descent(X, y, theta, alpha, iters):
    cost_array = np.zeros(iters)
    m = y.size
    for i in range(iters):
        cost, error = cost_function(X, y, theta)
        theta = theta - (alpha * (1/m) * np.dot(X.T, error))
        cost_array[i] = cost
    return theta, cost_array

def plotChart(iterations, cost_num):
    fig, ax = plt.subplots()
    ax.plot(np.arange(iterations), cost_num, 'r')
    ax.set_xlabel('Iterations')
    ax.set_ylabel('Cost')
    ax.set_title('Error vs Iterations')
    plt.style.use('fivethirtyeight')
    plt.show()
def run():

    X = dataset[['Userrating', 'Year']]
    y = dataset['Price']
    X = (X - X.mean()) / X.std()
    X = np.c_[np.ones(X.shape[0]), X]
    alpha = 0.003
    iterations = 10000
    theta = np.array(([626.30,-5.977,-0.2905]))
    initial_cost, _ = cost_function(X, y, theta)
    print('With initial theta values of {0}, cost error is {1}'.format(theta, initial_cost))
    theta, cost_num = gradient_descent(X, y, theta, alpha, iterations)
    plotChart(iterations, cost_num)
    final_cost, _ = cost_function(X, y, theta)
    print('With final theta values of {0}, cost error is {1}'.format(theta, final_cost))

if__name__== "_main_":
    run()
```

Results and Discussion:

Regression models have been built using possible predictor combination in order to identify the best predictors. The sample output for the model built using userrating as predictor is shown in Figure 1.



OLS Regression Results

Dep. Variable:	Price	R-squared:	0.025
Model:	OLS	Adj. R-squared:	0.023
Method:	Least Squares	F-statistic:	17.70
Date:	Mon, 21 Nov 2022	Prob (F-statistic):	2.93e-05
Time:	15:39:54	Log-Likelihood:	-2589.8
No. Observations:	700	AIC:	5184.
Df Residuals:	698	BIC:	5193.
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	45.7964	7.876	5.815	0.000	30.333	61.260
Userrating	-7.1331	1.696	-4.207	0.000	-10.462	-3.804

Omnibus:	629.252	Durbin-Watson:	1.848
Prob(Omnibus):	0.000	Jarque-Bera (JB):	22231.723
Skew:	3.941	Prob(JB):	0.00
Kurtosis:	29.459	Cond. No.	103.

Figure 1. Regression Results

The models built with different predictors and their P-value and Adjusted R squared value are shown in Table 1.

Table 1. Models and parameters

Sl.No	Model	P-value	Adj R-squared
1	Userrating	0.000	0.023
2	Reviews	0.002	0.012
3	Year	0.000	0.023
4	Reviews +Userrating	0.001	0.032
5	Reviews +year	0.002	0.023
6	Userrating+ year	0.000	0.035
7	Reviews+Userrating+Year	0.027	0.037

From the table it has been identified that the best predictors are Userrating and year since their p-value is low and high Adjusted R-squared. The regression model built is shown below in figure 2. The residuals are plotted in figure 3.

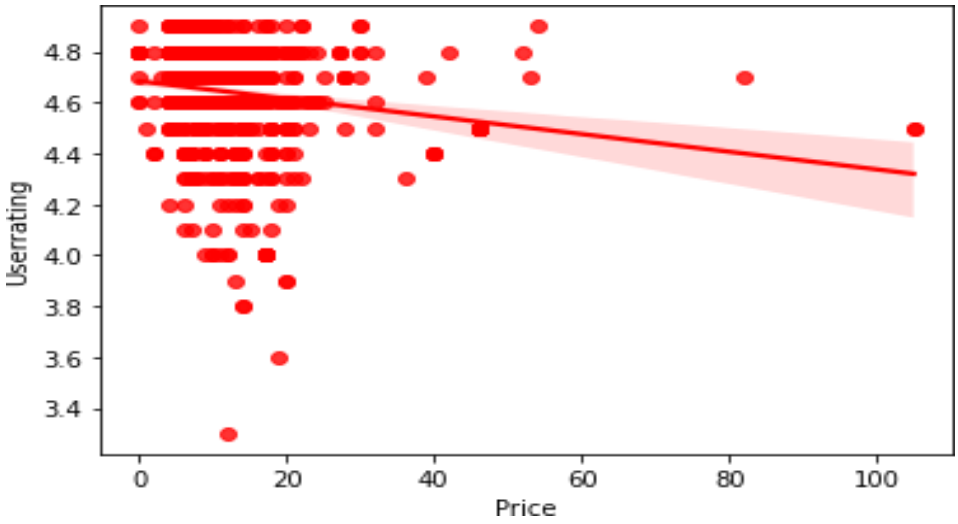


Figure 2. Regression model

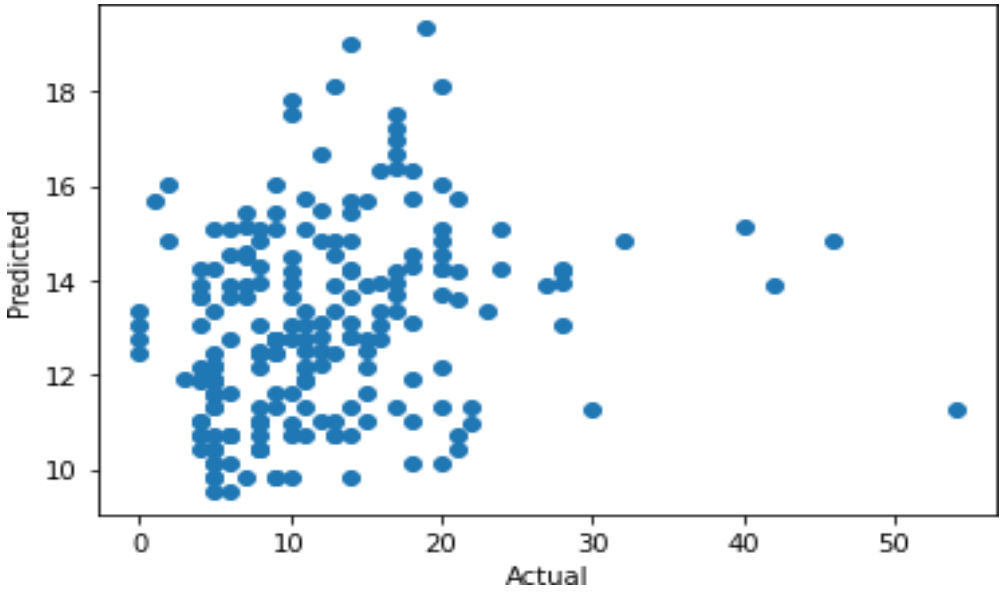


Figure 3. Residuals

The MSE for the built model is 59.109 and therefore to optimize the cost function gradient descent is applied. After optimization, the MSE is reduced to 47.22 with new coefficients β_0(12.7), β_1(-1.218), β_2(-1.190) with learning rate 0.003 and the results before and after optimization are shown in table 2 and the plot cost versus iterations is depicted in Figure 4.

Table 2. Gradient Descent

	Before Optimization	**After Optimization**
MSE	59.109	47.22
β_0	626.30	12.7
β_1	-5.977	-1.218
β_2	-0.2905	-1.190

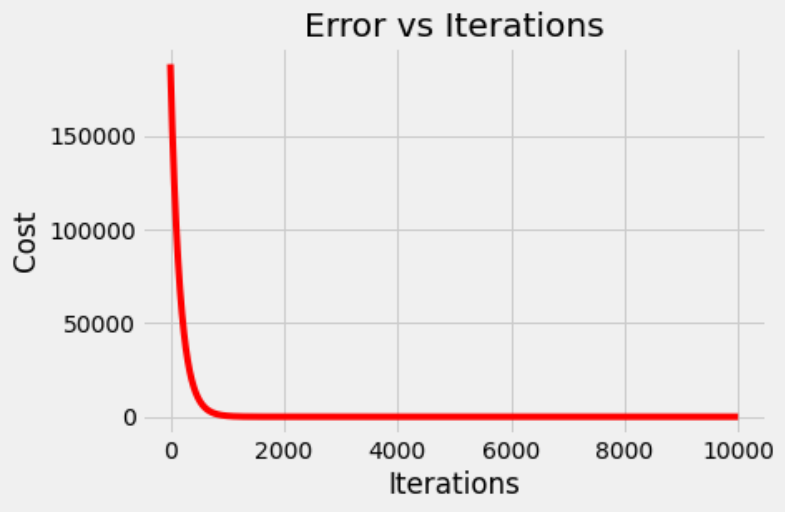


Figure 4. Cost Vs Iterations

Conclusion:

Thus, Multivariable Regression model is built, and the cost function (MSE) is reduced through optimization using gradient descent algorithm.

Ex No. 2	Classification

Aim:

To classify iris species using predictive models and to estimate the models.

Predictive analytics

Predictive modeling is a mathematical process used to predict future events or outcomes by analyzing patterns in each set of input data. A classification predictive model uses historical data to produce a broad analysis of a query. Following is few classification models used in this work.

Dataset:

IRIS dataset includes three iris species with 50 samples each as well as some properties about each flower. One flower species is linearly separable from the other two, but the other two are not linearly separable from each other. The columns in this dataset are: Id, SepalLengthCm, SepalWidthCm, PetalLengthCm, PetalWidthCm, Species.

K-Nearest Neighbor:

K-NN algorithm can be used for Regression as well as for Classification but mostly it is used for the Classification problems. K-NN is a non-parametric algorithm, which means it does not make any assumption on underlying data. It is also called a lazy learner algorithm because it does not learn from the training set immediately instead it stores the dataset and at the time of classification, it performs an action on the dataset.

KNN algorithm at the training phase just stores the dataset and when it gets new data, then it classifies that data into a category that is much similiar to the new data. There is no particular way to determine the best value for "K", so we need to try some values to find the best out of them. The most preferred value for K is 5.A very low value for K such as K=1 or K=2, can be noisy and lead to the effects of outliers in the model. Large values for K are good, but it may find some difficulties.

Nearest-neighbor algorithm

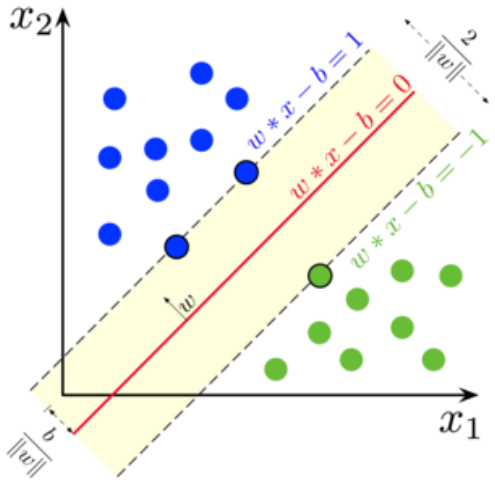
a) A pseudo code for the nearest neighbor algorithm is

ALGORITHM *Nearest-neighbor*$(D[1...n,1...n],s)$
//Input: A $n\times n$ distance matrix $D[1...n,1...n]$ and an index s of the starting city.
//Output: A list Path of the vertices containing the tour is obtained.
for $i \leftarrow 1$ to n do Visited $[i] \leftarrow$ false
Initialize the list Path with s
Visited $[s] \leftarrow$ true
Current $\leftarrow s$
for $i \leftarrow 2$ to n do
    Find the lowest element in row current and unmarked column j containing the element.
    Current $\leftarrow j$
    Visited$[j] \leftarrow$ true
    Add j to the end of list Path
Add s to the end of list Path
return Path

Support Vector Machine:

Support Vector Machine or SVM is one of the most popular Supervised Learning algorithms, which is used for Classification as well as Regression problems. The goal of the SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best decision boundary is called a hyperplane. SVM chooses the extreme points/vectors that help in creating the hyperplane. These extreme cases are

called as support vectors, and hence algorithm is termed as Support Vector Machine. In the SVM algorithm, we plot each data item as a point in n-dimensional space with the value of each feature being the value of a particular coordinate. In the SVM classifier, it is easy to have a linear hyper-plane between these two classes. The distance between the vectors and the hyperplane is called as margin. And the goal of SVM is to maximize this margin. The hyperplane with maximum margin is called the optimal hyperplane.



A Kernel Trick is a simple method where a Non-linear data is projected onto a higher dimension space so as to take it easier to classify the data where it could be linearly divided by a plane. The SVM kernel is a function that takes low dimensional input space and transforms it to a higher dimensional space i.e., it converts not separable problem to separable problem. It is mostly useful in non-linear separation problem.

SVM can be of two types:

Linear SVM: Linear SVM is used for linearly separable data, which means if a dataset can be classified into two classes by using a single straight line, then such data is termed as linearly separable data, and classifier is used called as Linear SVM classifier.

Non-linear SVM: Non-Linear SVM is used for non-linearly separated data, which means if a dataset cannot be classified by using a straight line, then such data is termed as non-linear data and classifier used is called as Non-linear SVM classifier

Major Kernel Functions:

Linear Kernel: Linear Kernel is used when the data is Linearly separable, that is, it can be separated using a single Line. It is one of the most common kernels to be used. It is mostly used when there are a Large number of Features in a particular Data Set.

RBF Kernel: RBF is the default kernel used within the sklearn's SVM classification algorithm and can be described with the following formula: where gamma can be set manually and has to be >0.

Polynomial Kernel: It represents the similarity of vectors in the training set of data in a feature space over polynomials of the original variables used in the kernel

Sigmoid Kernel: It is function is equivalent to a two-layer**,** perceptron model of the neural network, which is used as an activation function for artificial neurons.

Algorithm 1: SVM

1. Set $Input = (x_i, y_i)$, where $i = 1, 2, \cdots, N, x_i = R^n$ and $y_i = \{+1, -1\}$.
2. Assign $f(X) = \omega^T x_i + b = \sum_{i=1}^{N} \omega^T x_i + b = 0$
3. Minimize the QP problem as, $min\ \varphi(\omega, \xi) = \frac{1}{2}\|\omega\|^2 + C.(\sum_{i=1}^{N} \xi_i)$.
4. Calculate the dual Lagrangian multipliers as $min\ L_P = \frac{1}{2}\|\omega\|^2 - \sum_{i=1}^{N} x_i y_i(\omega x_i + b) + \sum_{i=1}^{N} x_i$.
5. Calculate the dual quadratic optimization (QP) problem as $max\ L_D = \sum_{i=1}^{N} x_i - \frac{1}{2}\sum_{i,j=1}^{N} x_i x_j\ y_i\ y_j\ (x_i, x_j)$.
6. Solve dual optimization problem as $\sum_{i=1}^{N} y_i x_i = 0$.
7. Output the classifier as $f(X) = sgn(\sum_{i=1}^{N} x_i y_i(x \cdot x_i) + j$.

Decision Tree:

The goal of using a Decision Tree is to create a training model that can be used to predict the class or value of the target variable by learning simple decision rules inferred from prior data (training data).

Types of decision trees are based on the type of target variable:

1. Categorical Variable Decision Tree:
2. Continuous Variable Decision Tree:

Entropy:

Entropy is an information theory metric that measures the impurity or uncertainty in a group of observations. It determines how a decision tree chooses to split data. The image below gives a better description of the purity of a set. ID3 algorithm uses entropy to calculate the homogeneity of a sample. If the sample is completely homogeneous the entropy is zero and if the sample is an equally divided it has entropy of one. It is computed between 0 and 1, however, heavily relying on the number of groups or classes present in the data set it can be more than 1 while depicting the same significance i.e. extreme level of disorder.

Gini Index:

The Gini Index or Gini Impurity computes the degree of probability of a specific variable that is wrongly being classified when chosen randomly and a variation of Gini coefficient. It works on categorical variables; provides outcomes either be “successful” or “failure” and hence conducts binary splitting only. The degree of Gini Index varies from 0 to 1, Where 0 depicts that all the elements be allied to a certain class or only one class exists there. The Gini Index of value as 1 signifies that all the elements are randomly z

distributed across various classes. A value of 0.5 denotes the elements are uniformly distributed into some classes.

$$Gini = 1 - \sum_{i=1}^{n} p^2(c_i)$$

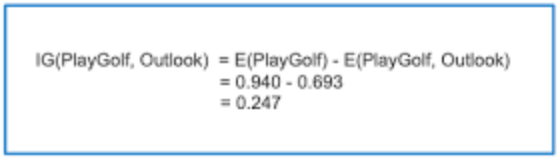
$$Entropy = \sum_{i=1}^{n} -p(c_i)log_2(p(c_i))$$

where $p(c_i)$ is the probability/percentage of class c_i in a node.

Information Gain:

Information gain is used for determining the best features/attributes that render maximum information about a class. It follows the concept of entropy while aiming at decreasing the level of entropy, beginning from the root node to the leaf nodes. Information gain computes the difference between entropy before and after split and specifies the impurity in class elements. According to the value of Information gain we split the node and build decision tree. A decision tree algorithm always tries to maximize the value of information, and node/attribute having high information split up first.

Information Gain(T,X) = Entropy(T) - Entropy(T, X)



ID3 Information Gain:

ID3 uses Information Gain or just Gain to find the best feature. Information Gain calculates the reduction in the entropy and measures how well a given feature separates or classifies the target classes. The feature with the highest Information Gain is selected as the best one. It is a classification algorithm that follows a greedy approach by selecting a best attribute that yields maximum Information Gain (IG) or minimum Entropy(H).

Algorithm: Generate_decision_tree. Generate a decision tree from the training tuples of data partition, *D*.

Input:

- Data partition, *D*, which is a set of training tuples and their associated class labels;
- *attribute_list*, the set of candidate attributes;
- *Attribute_selection_method*, a procedure to determine the splitting criterion that "best" partitions the data tuples into individual classes. This criterion consists of a *splitting_attribute* and, possibly, either a *split-point* or *splitting subset*.

Output: A decision tree.

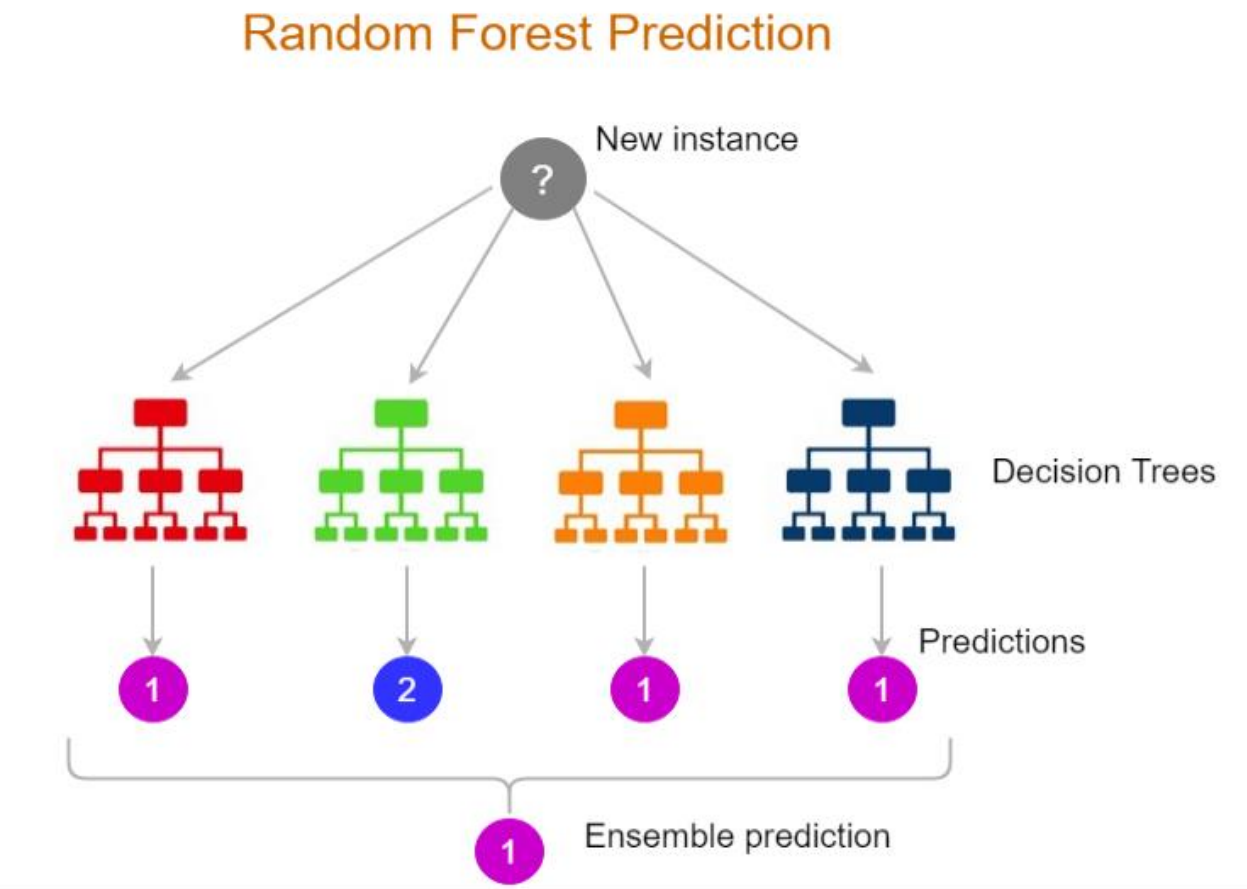
Method:

```
(1)  create a node N;
(2)  if tuples in D are all of the same class, C, then
(3)      return N as a leaf node labeled with the class C;
(4)  if attribute_list is empty then
(5)      return N as a leaf node labeled with the majority class in D; // majority voting
(6)  apply Attribute_selection_method(D, attribute_list) to find the "best" splitting_criterion;
(7)  label node N with splitting_criterion;
(8)  if splitting_attribute is discrete-valued and
         multiway splits allowed then // not restricted to binary trees
(9)      attribute_list ← attribute_list − splitting_attribute; // remove splitting_attribute
(10) for each outcome j of splitting_criterion
     // partition the tuples and grow subtrees for each partition
(11)     let Dj be the set of data tuples in D satisfying outcome j; // a partition
(12)     if Dj is empty then
(13)         attach a leaf labeled with the majority class in D to node N;
(14)     else attach the node returned by Generate_decision_tree(Dj, attribute_list) to node N;
     endfor
(15) return N;
```

Random Forest:

Random Forest 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. Random Forest is a classifier that contains a few decision trees on various subsets of the given dataset and takes the average to improve the predictive accuracy of that dataset. The random forest algorithm establishes the outcome based on the predictions of the decision trees. It predicts by taking the average or mean of the output from various trees. Increasing the number of

trees increases the precision of the outcome. Classification in random forests employs an ensemble methodology to attain the outcome. The training data is fed to train various decision trees. This dataset consists of observations and features that will be selected randomly during the splitting of nodes.



ROC CURVE:

ROC curves describe the trade-off between the true positive rate (TPR) and false positive (FPR) rate along different probability thresholds for a classifier. This curve plots two parameters:

True Positive Rate

False Positive Rate

An ROC curve plots TPR vs. FPR at different classification threshold

Lowering the classification threshold classifies more items as positive, thus increasing both False Positives and True Positives. The following figure shows a typical ROC curve.

AUC (Area under the ROC Curve).

AUC provides an aggregate measure of performance across all possible classification thresholds. One way of interpreting AUC is as the probability that the model ranks a random positive example more highly than a random negative example. The higher the AUC, the better the performance of the model at distinguishing between the positive and negative classes.

Algorithm 1: Pseudo code for the random forest algorithm

To generate c classifiers:
for i = 1 to c **do**
  Randomly sample the training data D with replacement to produce D_i
  Create a root node, N_i containing D_i
  Call BuildTree(N_i)
end for

BuildTree(N):
if N contains instances of only one class **then**
  return
else
  Randomly select x% of the possible splitting features in N
  Select the feature F with the highest information gain to split on
  Create f child nodes of N, N_1,..., N_f, where F has f possible values (F_1, … , F_f)
  for i = 1 to f **do**
    Set the contents of N_i to D_i, where D_i is all instances in N that match
    F_i
    Call BuildTree(N_i)
  end for
end if

AdaBoost:

AdaBoost algorithm, short for Adaptive Boosting, is a Boosting technique used as an Ensemble Method in Machine Learning. It is called Adaptive Boosting as the weights are re-assigned to each instance, with higher weights assigned to incorrectly classified instances. Boosting is used to reduce bias as well as variance for supervised learning. The most common algorithm used with AdaBoost is decision trees with one level that means with Decision trees with only 1 split. These trees are also called **Decision Stumps.**

AdaBoost can be used **to boost the performance of any machine learning algorithm**. It is best used with weak learners. These are models that achieve accuracy just above random chance on a classification problem. The most suited and therefore most common algorithm used with AdaBoost are decision trees with one level.

Given: $(x_1, y_1), \ldots, (x_m, y_m)$ where $x_i \in \mathcal{X}$, $y_i \in \{-1, +1\}$.
Initialize: $D_1(i) = 1/m$ for $i = 1, \ldots, m$.
For $t = 1, \ldots, T$:

- Train weak learner using distribution D_t.
- Get weak hypothesis $h_t : \mathcal{X} \to \{-1, +1\}$.
- Aim: select h_t with low weighted error:

$$\varepsilon_t = \Pr_{i \sim D_t} [h_t(x_i) \neq y_i].$$

- Choose $\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \varepsilon_t}{\varepsilon_t} \right)$.
- Update, for $i = 1, \ldots, m$:

$$D_{t+1}(i) = \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t}$$

where Z_t is a normalization factor (chosen so that D_{t+1} will be a distribution).

Output the final hypothesis:

$$H(x) = \text{sign} \left(\sum_{t=1}^{T} \alpha_t h_t(x) \right).$$

Fig. 1 The boosting algorithm AdaBoost.

Gradient Boosting:

Gradient boosting algorithm can be used for predicting not only continuous target variable (as a Regressor) but also categorical target variable (as a Classifier). When it is used as a regressor, the cost function is **Mean Square Error (MSE)** and when it is used as a classifier then the cost function is **Log loss.**

This method creates the model in a stage-wise fashion. It infers the model by enabling the optimization of an absolute differentiable loss function. As we add each weak learner, a new model is created that gives a more precise estimation of the response variable. Gradient boosting is a highly robust technique for developing predictive models. It applies to several risk functions and optimizes the accuracy of the model's prediction. It also resolves multicollinearity problems where the correlations among the predictor variables are high.

Gradient Boosting Algorithm

1. Initialize model with a constant value:

$$F_0(x) = \underset{\gamma}{argmin} \sum_{i=1}^{n} L(y_i, \gamma)$$

2. for $m = 1$ *to* M:

2-1. Compute residuals $r_{im} = -\left[\frac{\partial L(y_i,\ F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}$ $for\ i = 1,\ldots, n$

2-2. Train regression tree with features x against r and create terminal node reasions R_{jm} for $j = 1,\ldots, J_m$

2-3. Compute $\gamma_{jm} = \underset{\gamma}{argmin} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma)$ $for\ j = 1,\ldots, J_m$

2-4. Update the model:

$$F_m(x) = F_{m-1}(x) + \nu \sum_{j=1}^{J_m} \gamma_{jm} 1(x \in R_{jm})$$

Implementation: - KNN

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import Normalizer
from collections import Counter
from sklearn.metrics import accuracy_scorea
from sklearn.neighbors import KNeighborsClassifier
 from sklearn import datasets
iris=datasets.load_iris()
 Iris
predictors = iris.data[:, 0:2]
outcomes = iris.target
 plt.plot(predictors[outcomes == 0][:, 0], predictors[outcomes == 0][:, 1], "ro")
 plt.plot(predictors[outcomes == 1][:, 0], predictors[outcomes == 1][:, 1], "go")
  plt.plot(predictors[outcomes == 2][:, 0], predictors[outcomes == 2][:, 1], "bo")
 iris_df=pd.DataFrame(data=np.c_[iris['data'],iris['target']],columns= iris['feature_names'] + ['target'])
iris_df.head()
iris_df.describe()
x= iris_df.iloc[:, :-1]
y= iris_df.iloc[:, -1]
x_train, x_test, y_train, y_test= train_test_split(x, y,test_size= 0.2,shuffle= True,random_state= 0)
x_train= np.asarray(x_train)
y_train= np.asarray(y_train)
x_test= np.asarray(x_test)
y_test= np.asarray(y_test)
scaler= Normalizer().fit(x_train) # the scaler is fitted to the training set
normalized_x_train= scaler.transform(x_train) # the scaler is applied to the training set
normalized_x_test= scaler.transform(x_test) def distance_ecu(x_train, x_test_point):
  distances= []  ## create empty list called distances
  for row in range(len(x_train)):
      current_train_point= x_train[row]
      current_distance= 0
      for col in range(len(current_train_point)):

          current_distance += (current_train_point[col] - x_test_point[col]) **2
          ## Or current_distance = current_distance + (x_train[i] - x_test_point[i])**2
      current_distance= np.sqrt(current_distance)


      distances.append(current_distance)
  distances= pd.DataFrame(data=distances,columns=['dist'])
  return distances


def nearest_neighbors(distance_point, K):
    df_nearest= distance_point.sort_values(by=['dist'], axis=0)
    df_nearest= df_nearest[:K]
    return df_nearest def voting(df_nearest, y_train):
    counter_vote= Counter(y_train[df_nearest.index])


    y_pred= counter_vote.most_common()[0][0]
```

```
    return y_pred
def KNN(x_train, y_train, x_test, K):
    y_pred=[]
    for x_test_point in x_test:
      distance_point  = distance_ecu(x_train, x_test_point)
      df_nearest_point= nearest_neighbors(distance_point, K)
      y_pred_point    = voting(df_nearest_point, y_train)
      y_pred.append(y_pred_point)
    return y_pred
K=3
y_pred_scratch= KNN(normalized_x_train, y_train, normalized_x_test, K)
print(y_pred_scratch)
knn=KNeighborsClassifier(K)
knn.fit(normalized_x_train, y_train)
y_pred_sklearn= knn.predict(normalized_x_test)
print(y_pred_sklearn)
print(f'The accuracy of our implementation is {accuracy score(y_test, y_pred_scratch)}')
print(f'The accuracy of sklearn implementation is {accuracy_score(y_test, y_pred_sklearn)}')
precision = precision_score(y_test, y_pred_scratch, labels=[1,2], average='micro')
print('Precision: %.3f' % precision)
recall = recall_score(y_test, y_pred_scratch, labels=[1,2], average='micro')
print('Recall: %.3f' % recall)
```

Implementation - SVM

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets
import pandas as pd
import seaborn as sns
iris=datasets.load_iris() X = iris.data[:, :2]
y = iris.target
C = 1.0
```

Linear kernal

```
x_train,x_test,y_train,y_test=train_test_split(X,y,random_state=0,test_size=0.25)
svc = svm.SVC(kernel ='linear', C = 1).fit(X, y) classifier_predictions=svc.predict(x_test)
print(accuracy_score(y_test,classifier_predictions)*100)
print(precision_score(y_test,classifier_predictions, pos_label='positive'
                                ,average='micro')*100)
print(recall_score(y_test,classifier_predictions, pos_label='positive'
                                ,average='micro')*100) x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
h = (x_max / x_min)/100
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
        np.arange(y_min, y_max, h))
plt.subplot(1, 1, 1)
Z = svc.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap = plt.cm.Paired, alpha = 0.8)

plt.scatter(X[:, 0], X[:, 1], c = y, cmap = plt.cm.Paired)
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')
plt.xlim(xx.min(), xx.max())
```

```
plt.title('SVC with linear kernel')
plt.show()
classifier_predictions=svc.predict(x_test)
print(accuracy_score(y_test,classifier_predictions)*100)
print(precision_score(y_test,classifier_predictions, pos_label='positive'
                      ,average='micro')*100)
print(recall_score(y_test,classifier_predictions, pos_label='positive'
                      ,average='micro')*100)
```

Decision tree

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn import tree
from sklearn.datasets import load_iris
iris=load_iris()
iris
print(iris.feature_names)
print(iris.target_names)
#Spilitting the dataset
removed =[0,50,100]
new_target = np.delete(iris.target,removed)
new_data = np.delete(iris.data,removed, axis=0)
#train classifier
clf = tree.DecisionTreeClassifier()
clf=clf.fit(new_data,new_target)
prediction = clf.predict(iris.data[removed])
print("Original Labels",iris.target[removed])
print("Labels Predicted",prediction)

import numpy as np
import pandas as pd
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
x=iris.data
y=iris.target
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test=train_test_split(x,y,random_state=0,train_size=0.75)
from sklearn.tree import DecisionTreeClassifier
decision = DecisionTreeClassifier(criterion='gini',random_state = 0)
plt.figure(figsize=(12,8))
from sklearn import tree
tree.plot_tree(decision.fit(x_train, y_train))
plt.figure(figsize=(12,8))
from sklearn import tree
tree.plot_tree(decision.fit(x_train, y_train))
#Entropy
def entropy(p):
    return - p * np.log2(p) - (1 - p) * np.log2(1 - p)
# change default figure and font size
plt.rcParams['figure.figsize'] = 8, 6
plt.rcParams['font.size'] = 12
x = np.arange(0.0, 1.0, 0.01)
```

```
ent = [entropy(p) if p != 0 else None for p in x]
plt.plot(x, ent)
plt.axhline(y = 1.0, linewidth = 1, color = 'k', linestyle = '--')
plt.ylim([ 0, 1.1 ])
plt.xlabel('p(i=1)')
plt.ylabel('Entropy')
plt.show()
#Gini Index
def gini(p):
   return p * (1 - p) + (1 - p) * ( 1 - (1 - p) )
gi = gini(x)
# plot
for i, lab in zip([ent, gi], ['Entropy', 'Gini Index']):
   plt.plot(x, i, label = lab)
plt.legend(loc = 'upper center', bbox_to_anchor = (0.5, 1.15),
          ncol = 3, fancybox = True, shadow = False)
plt.axhline(y = 0.5, linewidth = 1, color = 'k', linestyle = '--')
plt.axhline(y = 1.0, linewidth = 1, color = 'k', linestyle = '--')
plt.ylim([ 0, 1.1 ])
plt.xlabel('p(i=1)')
plt.ylabel('Impurity')
plt.tight_layout()
plt.show()

# Information Gain
def information_gain(df, split_attribute, target_attribute,battr):
   print(" Information Gain Calculation of",split_attribute,)
   df_split = df.groupby(split_attribute)
   glist=[]
   for gname,group in df_split:
      print('Grouped Attribute Values ',group)
       glist.append(gname)
  glist.reverse()
   nobs = len(df.index) * 1.0
   df_agg1=df_split.agg({target_attribute:lambda x:entropy (x, glist.pop())})
   df_agg2=df_split.agg({target_attribute :lambda x:len(x)/nobs})
   df_agg1.columns=['Entropy']
   df_agg2.columns=['Proportion']
  # Calculate Information Gain:
   new_entropy = sum( df_agg1['Entropy'] * df_agg2['Proportion'])
   if battr !='S':
      old_entropy =entropy(df[target_attribute],'S-'+df.iloc[0][df.columns.get_loc(battr)])
   else:
      old_entropy = entropy(df[target_attribute],battr)
   return old_entropy - new_entropy
#ID3 Information Gain
def id3(df, target_attribute, attribute_names, default_class=None,default_attr='S'):

   from collections import Counter
   cnt = Counter(x for x in df[target_attribute])# class of YES /NO
   ## First check: Is this split of the dataset homogeneous?
   if len(cnt) == 1:
      return next(iter(cnt))  # next input data set, or raises StopIteration when EOF is hit.
   ## Second check: Is this split of the dataset empty? if yes, return a default value
   elif df.empty or (not attribute_names):
      return default_class  # Return None for Empty Data Set
```

```
  ## Otherwise: This dataset is ready to be devied up!
   else:
      # Get Default Value for next recursive call of this function:
      default_class = max(cnt.keys()) #No of YES and NO Class
      # Compute the Information Gain of the attributes:
      gainz=[]
      for attr in attribute_names:
         ig= information_gain(df, attr, target_attribute,default_attr)
         gainz.append(ig)
         print('\nInformation gain of',"“",attr,"”",'is ➞', ig)
         print("============================================================")
      index_of_max = gainz.index(max(gainz))            # Index of Best Attribute
      best_attr = attribute_names[index_of_max]         # Choose Best Attribute to split on
      print("\nList of Gain for arrtibutes:",attribute_names,"\nare:", gainz,"respectively.")
      print("\nAttribute with the maximum gain is ➞", best_attr)
      print("\nHence, the Root node will be ➞", best_attr)
      print("============================================================")
     # Create an empty tree, to be populated in a moment
      tree = {best_attr:{}} # Initiate the tree with best attribute as a node
      remaining_attribute_names =[i for i in attribute_names if i != best_attr]
     # Split dataset
On each split, recursively call this algorithm.Populate the empty tree with subtrees, which
are the result of the recursive call
      for attr_val, data_subset in df.groupby(best_attr):
         subtree = id3(data_subset,target_attribute, remaining_attribute_names,default_class,best_attr)
         tree[best_attr][attr_val] = subtree
      return tree
```

Random Forest

Implememtation

```
from sklearn import datasets
iris = datasets.load_iris()
print(iris.target_names)
print(iris.feature_names)
print(iris.data[0:5])
print(iris.target)
import pandas as pd
data=pd.DataFrame({
   'sepal length':iris.data[:,0],
   'sepal width':iris.data[:,1],
   'petal length':iris.data[:,2],
   'petal width':iris.data[:,3],
   'species':iris.target
})
data.head()
from sklearn.model_selection import train_test_split


X=data[['sepal length', 'sepal width', 'petal length', 'petal width']]  # Features
y=data['species']  # Labels


# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

```
from sklearn.ensemble import RandomForestClassifier
clf=RandomForestClassifier(n_estimators=100)
clf.fit(X_train,y_train)


y_pred=clf.predict(X_test)
from sklearn import metrics
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
import numpy as np
species_idx = clf.predict([[3, 5, 4, 2]])[0]
iris.target_names[species_idx]
import pandas as pd
feature_imp = pd.Series(clf.feature_importances_,index=iris.feature_names).sort_values(ascending=False)
feature_imp
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

sns.barplot(x=feature_imp, y=feature_imp.index)
plt.xlabel('Feature Importance Score')
plt.ylabel('Features')
plt.title("Visualizing Important Features")
plt.legend()
plt.show()

import numpy as np
from sklearn.model_selection import train_test_split
X=data[['petal length', 'petal width','sepal length']]
y=data['species']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
from sklearn.ensemble import RandomForestClassifier
clf=RandomForestClassifier(n_estimators=100)
clf.fit(X_train,y_train)
y_pred=clf.predict(X_test)
from sklearn.metrics import precision_score,recall_score,accuracy_score
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
print("Precision Score : ",precision_score(y_test, y_pred,pos_label='positive',average='micro'))
print("Recall Score : ",recall_score(y_test, y_pred,pos_label='positive',average='micro'))

Roc curve
from sklearn.metrics import roc_auc_score,roc_curve,auc
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import RocCurveDisplay
from sklearn.multiclass import OneVsRestClassifier
RF=OneVsRestClassifier(RandomForestClassifier(max_features=0.2))
RF.fit(X_train,y_train)

y_pred =RF.predict(X_test)
pred_prob=RF.predict_proba(X_test)
from  sklearn.preprocessing import label_binarize
classes=np.unique(y_test)
```

```
y_test_binarized=label_binarize(y_test,classes)
fpr={}
tpr={}
thresh={}
roc_auc=dict()


n_class = classes.shape
for i in range(n_class):
    fpr[i],tpr[i],thresh[i]=roc_curve(y_test_binarized[:,1],pred_prob[:,1])
    roc_auc[i]=auc(fpr[i],tpr[i])
    plt.plot(fpr[i],tpr[i],linestyle='--',
          label='%s vs Rest (AUC=%0.2f)'%(classes[i],roc_auc[i]))
plt.plot([0,1],[0,1],'b--')
plt.title("Multiclass ROC curve")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.legend(loc='lower right')
plt.show()
```

Implementation:

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn import metrics

iris = datasets.load_iris()
X = iris.data
y = iris.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

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


abc = AdaBoostClassifier(n_estimators=50,learning_rate=1)
model = abc.fit(X_train, y_train)
y_pred = model.predict(X_test)

print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
print("Precision:",metrics.precision_score(y_test, y_pred,pos_label='positive',average='micro'))
print("Recall:",metrics.recall_score(y_test, y_pred,pos_label='positive',average='micro'))

from sklearn.ensemble import AdaBoostClassifier
from sklearn.svm import SVC
from sklearn import metrics


svc=SVC(probability=True, kernel='linear')
abc =AdaBoostClassifier(n_estimators=50, base_estimator=svc,learning_rate=1)
model = abc.fit(X_train, y_train)
```

```
y_pred = model.predict(X_test)
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
print("Precision:",metrics.precision_score(y_test, y_pred,pos_label='positive',average='micro'))
print("Recall:",metrics.recall_score(y_test, y_pred,pos_label='positive',average='micro'))
```

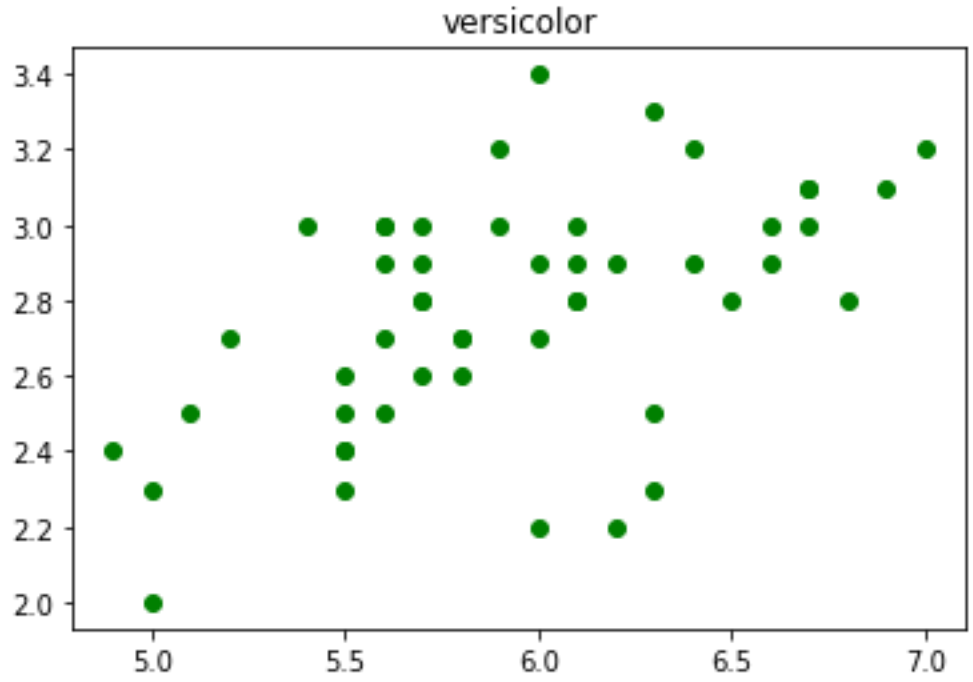
Gradient

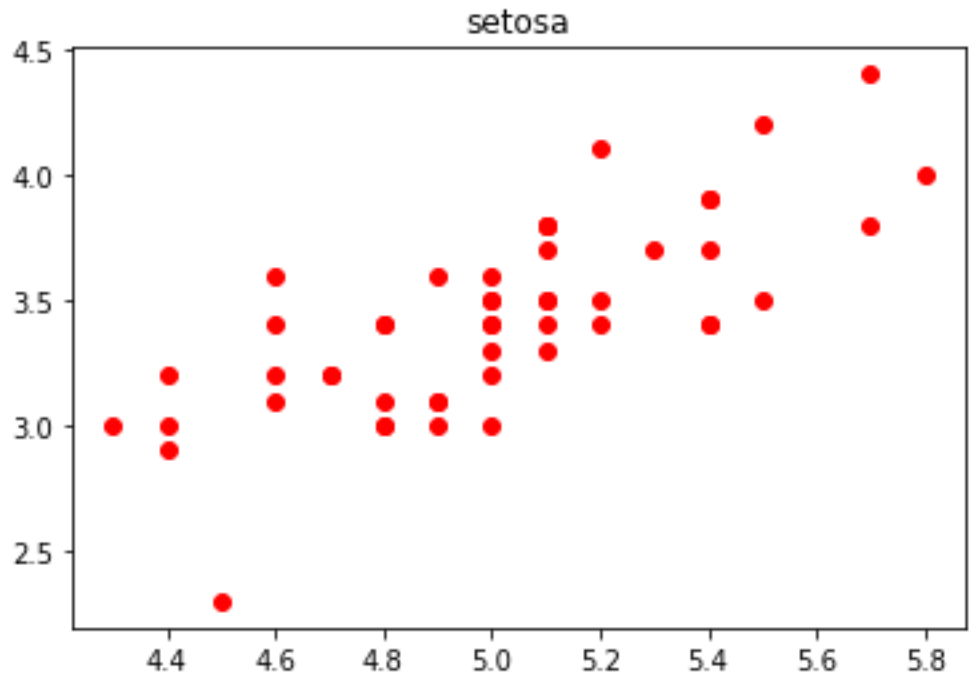
```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn import metrics
import pandas as pd
import matplotlib.pyplot as plt
import  numpy as np
iris = datasets.load_iris()
X = iris.data
y = iris.target
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report, \
f1_score, roc_auc_score, roc_curve, precision_score, recall_score
from IPython.display import display
import matplotlib.pyplot as plt
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [10,6]
import warnings
warnings.filterwarnings('ignore')
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
std = StandardScaler()
print('\033[1mStandardardization on Training set'.center(100))
X_train_std = std.fit_transform(X_train)
X_train_std = pd.DataFrame(X_train_std)
display(X_train_std.describe())
print('\n','\033[1mStandardardization on Testing set'.center(100))
X_test_std = std.transform(X_test)
X_test_std = pd.DataFrame(X_test_std)
display(X_test_std.describe())
GB_model = GradientBoostingClassifier()
GB_model.fit(X_train_std,y_train)
param_dist = {
    "n_estimators":[5,20,100,500],
    "max_depth":[1,3,5,7,9],
    "learning_rate":[0.01,0.1,1,10,100]}
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
RCV = RandomizedSearchCV(GB_model, param_dist, n_iter=50, scoring='accuracy', n_jobs=-
1, cv=5, random_state=1)
Evaluation_Results = pd.DataFrame(np.zeros((8,5)), columns=['Accuracy', 'Precision','Recall','F1-score','AUC-
ROC score'])
Evaluation_Results.index=['Logistic Regression (LR)','Decision Tree Classifier (DT)','Random Forest Classifier (RF
)','Naïve Bayes Classifier (NB)',
                          'Support Vector Machine (SVM)','K Nearest Neighbours (KNN)', 'Gradient Boosting (GB)','Extreme
 Gradient Boosting (XGB)']
Evaluation_Results
def Classification_Summary(pred,pred_prob,i):
    Evaluation_Results.iloc[i]['Accuracy']=round(accuracy_score(y_test, pred),3)*100
```

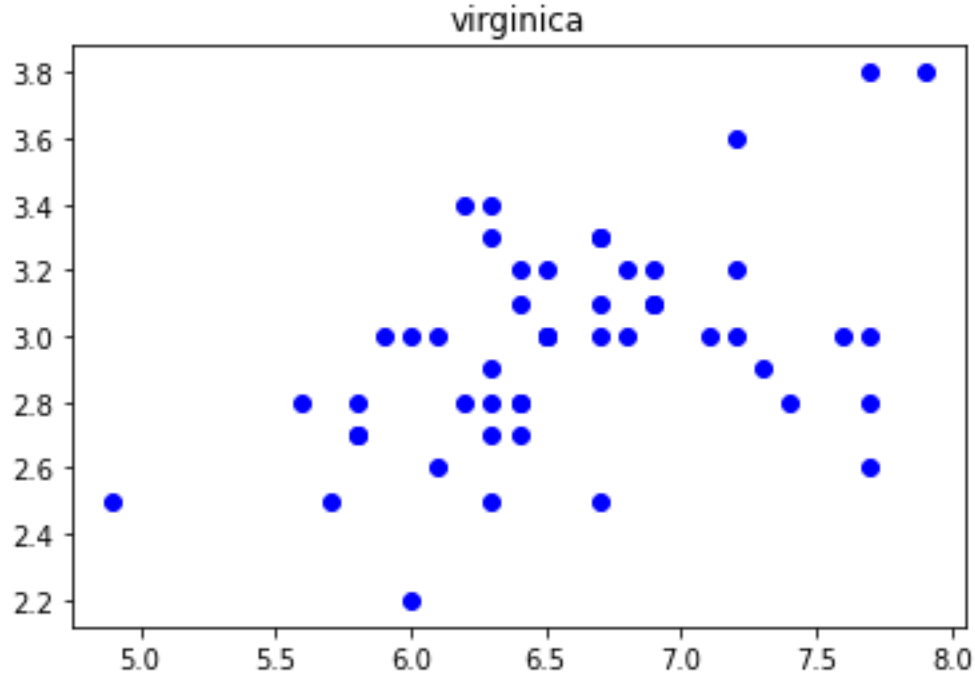
```
    Evaluation_Results.iloc[i]['Precision']=round(precision_score(y_test, pred, average='weighted'),3)*100 #
    Evaluation_Results.iloc[i]['Recall']=round(recall_score(y_test, pred, average='weighted'),3)*100 #
    Evaluation_Results.iloc[i]['F1-score']=round(f1_score(y_test, pred, average='weighted'),3)*100 #
    Evaluation_Results.iloc[i]['AUC-
ROC score']=round(roc_auc_score(y_test, pred_prob, multi_class='ovr'),3)*100 #[:, 1]
    print('{}{}\033[1m Evaluating {} \033[0m{}{}\n'.format('<'*3,'-'*35,Evaluation_Results.index[i], '-'*35,'>'*3))
    print('Accuracy = {}%'.format(round(accuracy_score(y_test, pred),3)*100))
    print('F1 Score = {}%'.format(round(f1_score(y_test, pred, average='weighted'),3)*100)) #
    print('\n \033[1mConfusiton Matrix:\033[0m\n',confusion_matrix(y_test, pred))
    print('\n\033[1mClassification Report:\033[0m\n',classification_report(y_test, pred))


GB = RCV.fit(X_train_std, y_train).best_estimator_
pred = GB.predict(X_test_std)
pred_prob = GB.predict_proba(X_test_std)
Classification_Summary(pred,pred_prob,6)

 from sklearn.metrics  import RocCurveDisplay
RocCurveDisplay.from_predictions(y_test,pred,pos_label=1)
```

Result:

The distribution of the three classes of the dataset is shown in the following figure.

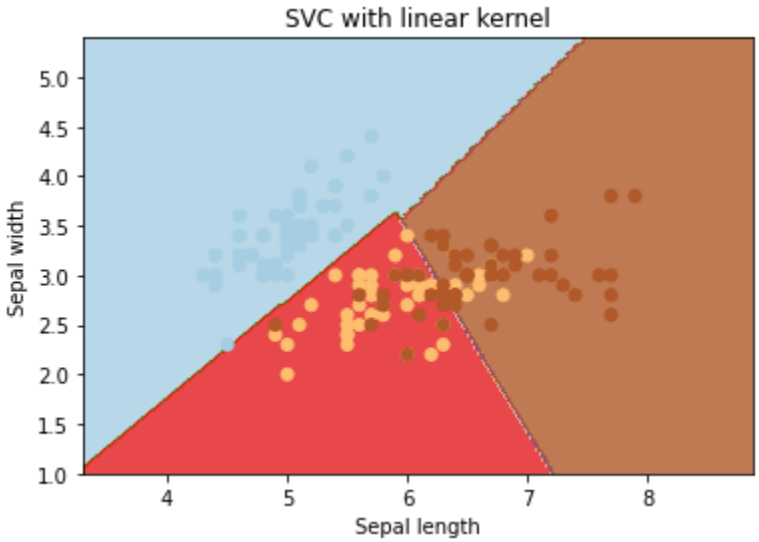



setosa
4.5
4.0
3.5
3.0
2.5
4.4
4.6
4.8
5.0
5.2
5.4
5.6
5.8

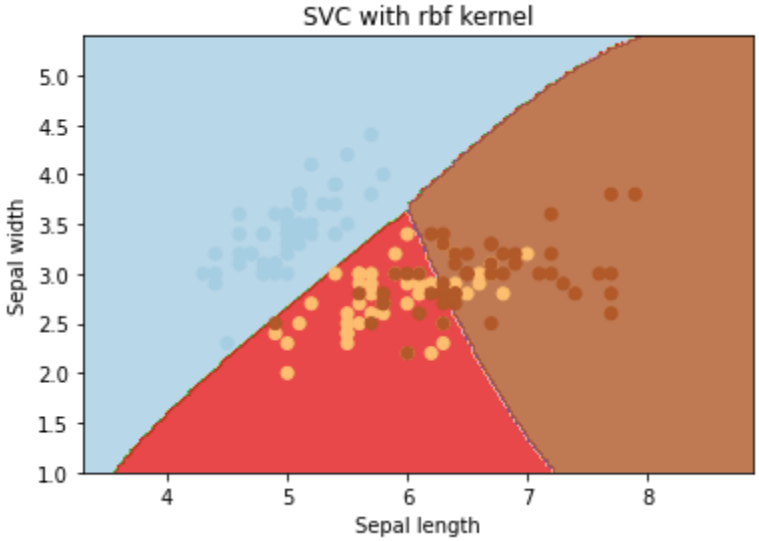

virginica
3.8
3.6
3.4
3.2
3.0
2.8
2.6
2.4
2.2
5.0
5.5
6.0
6.5
7.0
7.5
8.0

The decision boundary of each kernel is visualised as below:

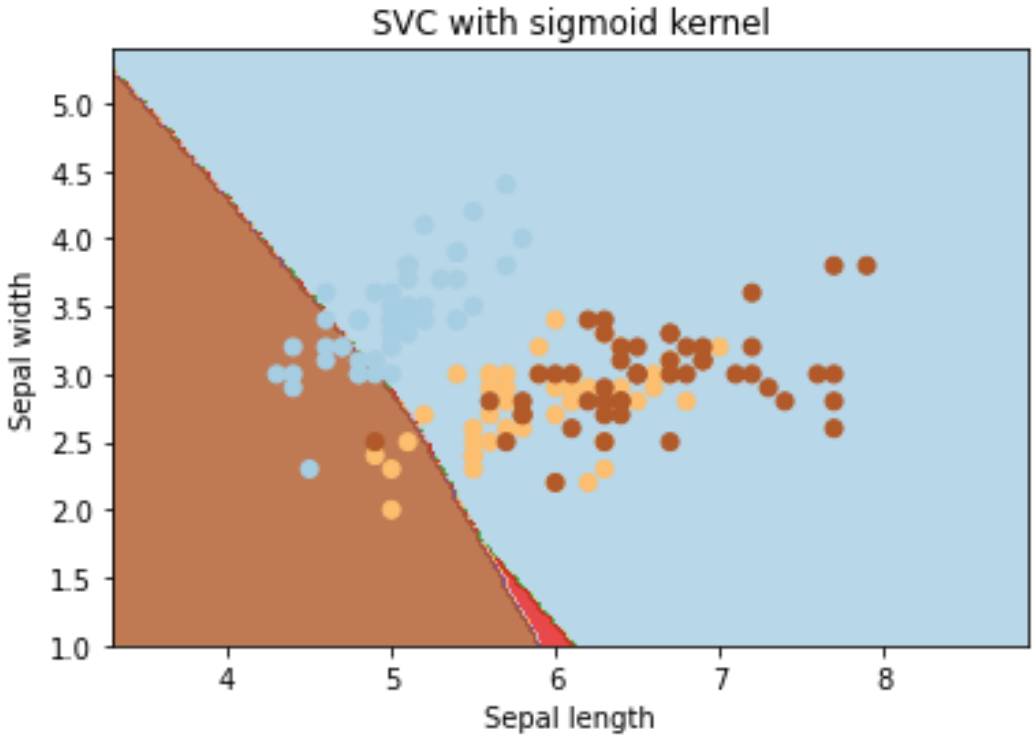
Linear kernel:



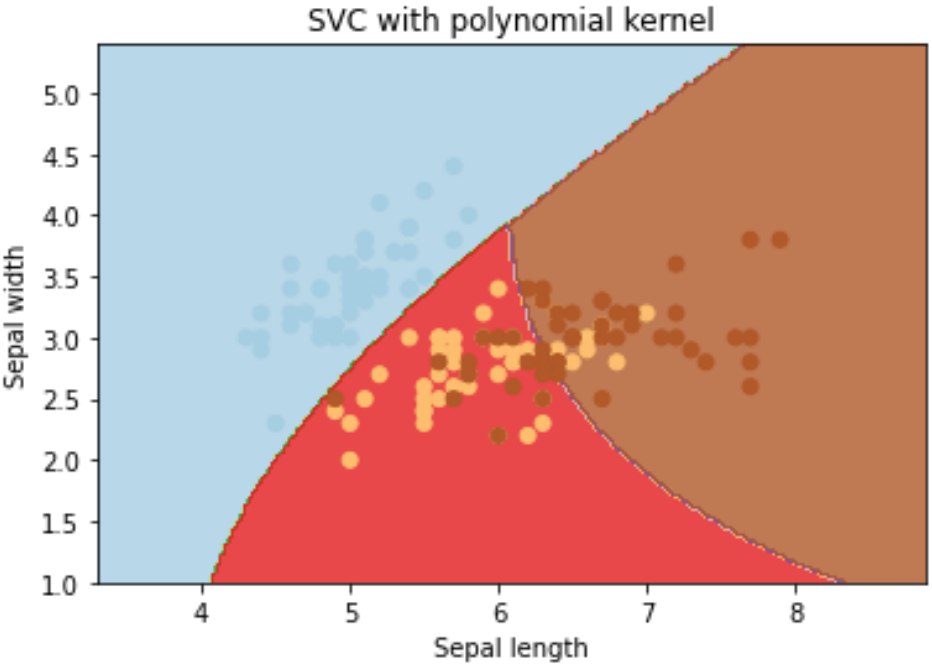
RBF kernel:



Sigmoid Kernel

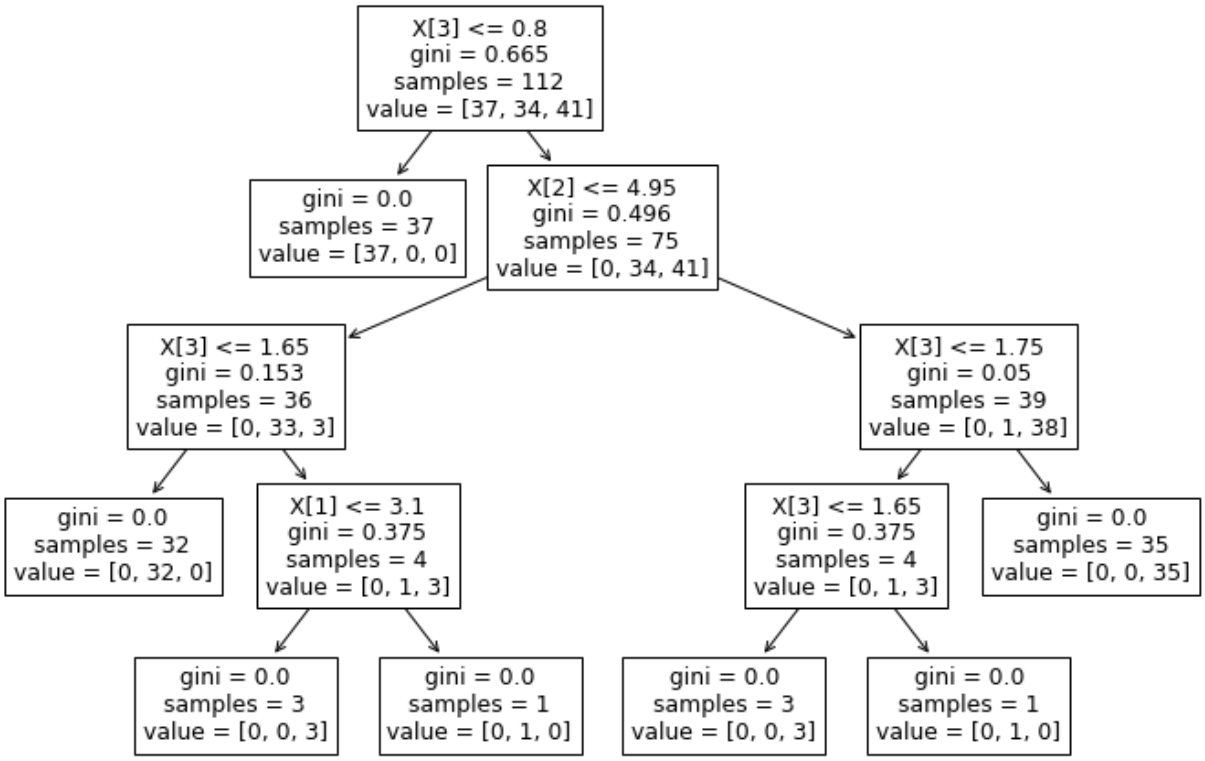


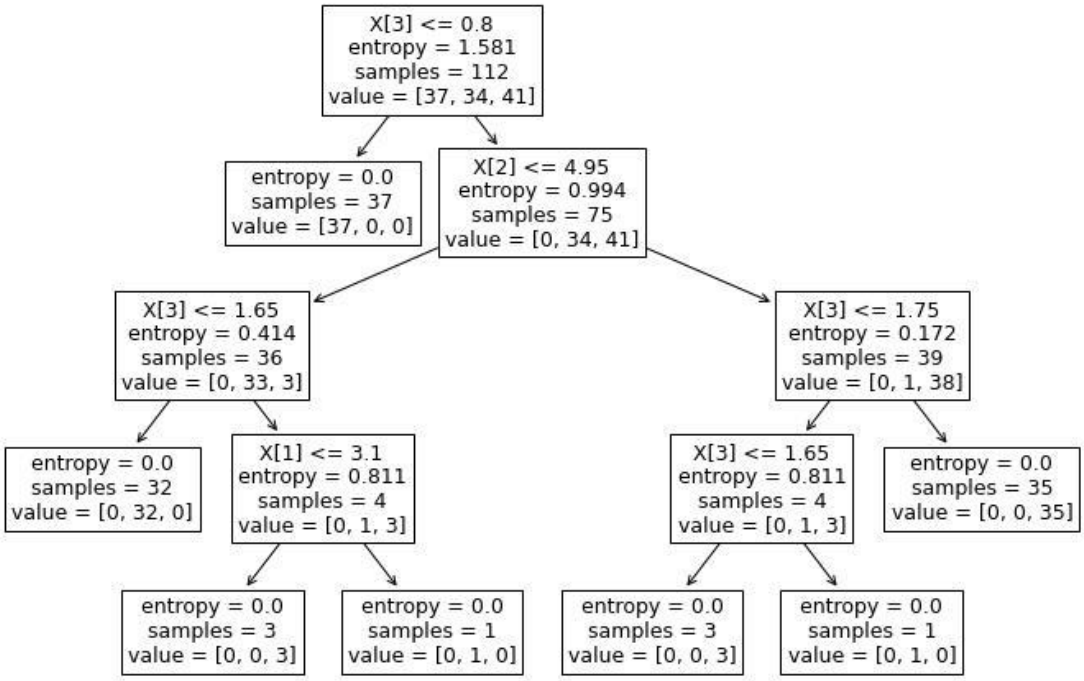
Polynomial Kernel



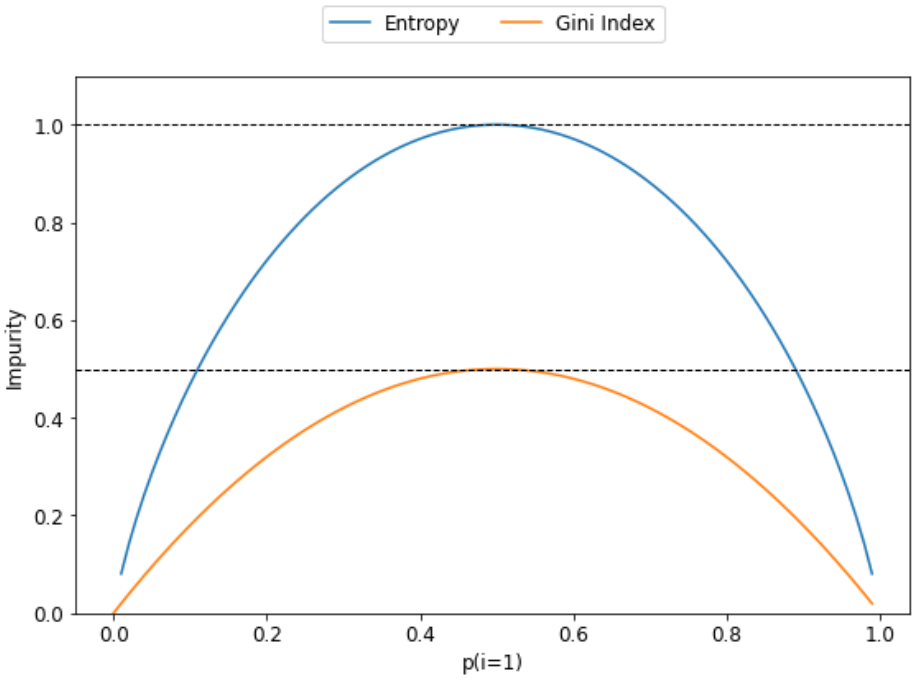
l

The Decision tree view for the iris dataset is shown below:

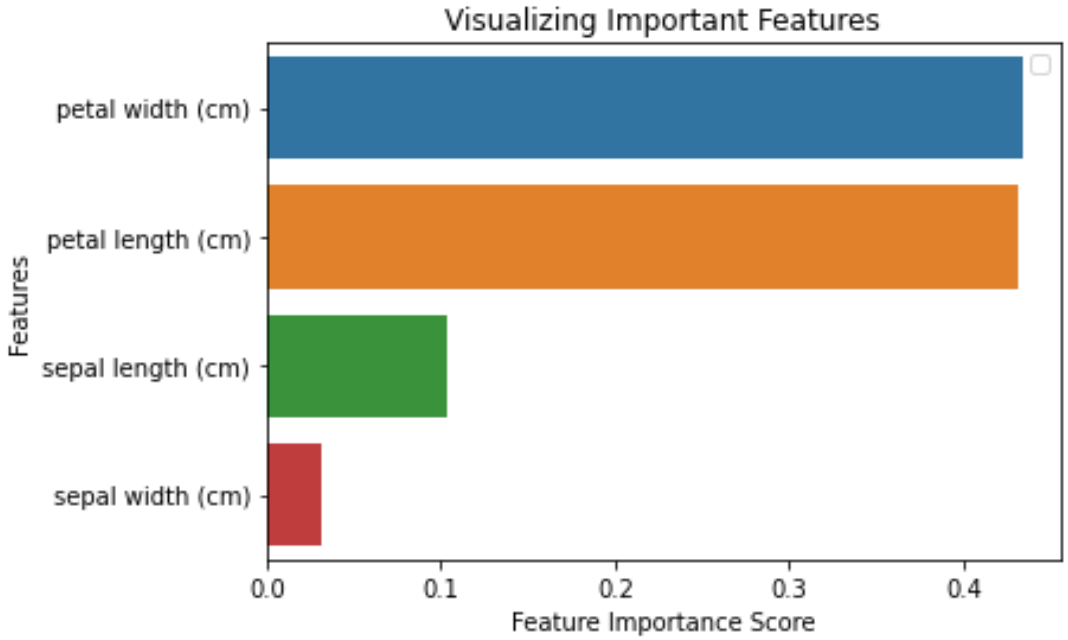




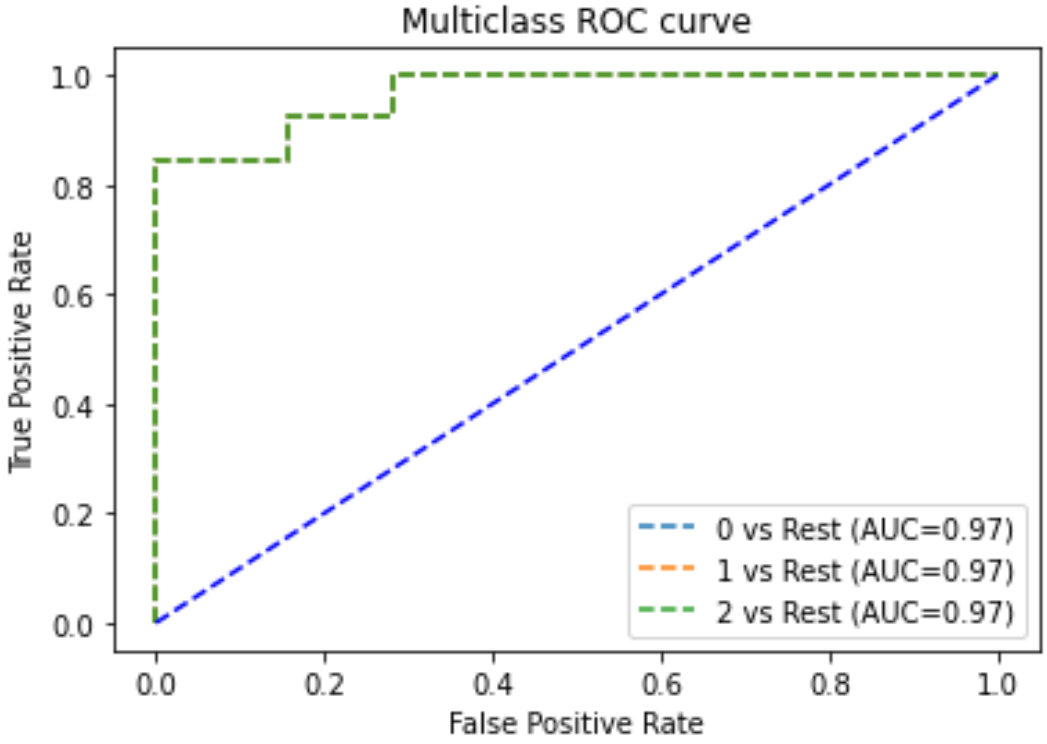
The range of attribute selection measures entropy and gini index is visualised as below:



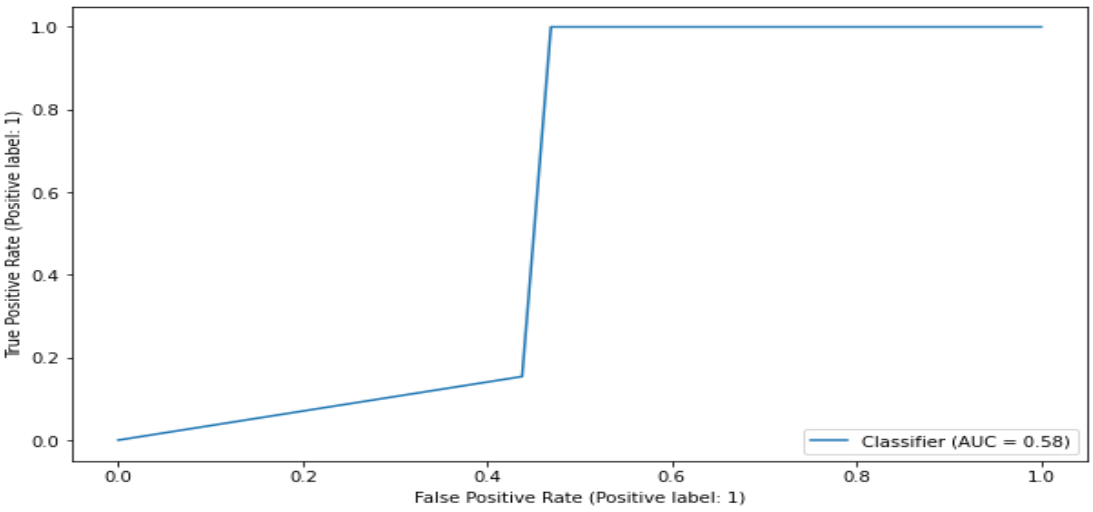
The feature importance is visualised as shown below:



ROC curve - Random Forest



ROC curve – Gradient Boosting



Performance metrices of various built models

Classifier(s)	Accuracy (%)	Precision (%)	Recall (%)
KNN (K=3)	96.6	94.7	94.7
KNN (K=7)	96.6	94.7	94.7
KNN (K=9)	99.9	99.9	99.9
SVM Linear	76.3	76.3	76.3
SVM RBF	76.3	76.3	76.3
SVM Sigmoid	28.94	28.94	28.94
SVM polynomial	76.3	76.3	76.3
Decision Tree (Entropy)	100	100	100
Decision Tree (Gini Index)	97	97	96
Random Forest	93.33	93.33	93.33
AdaBoost	82.2	82.2	82.2
Gradient Boosting	93	100	100

Conclusion

The iris dataset is considered in this work and classifiers KNN, DT, SVM, RF, AB and GB are used to build machine learning models and they are evaluated.