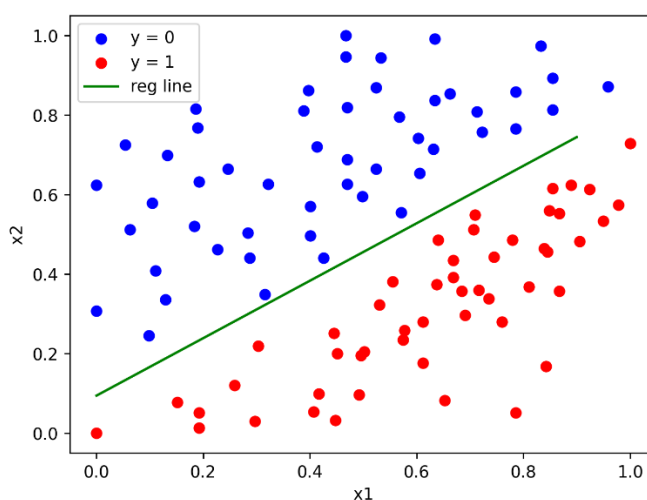


Assignment #9 – CPSC 552

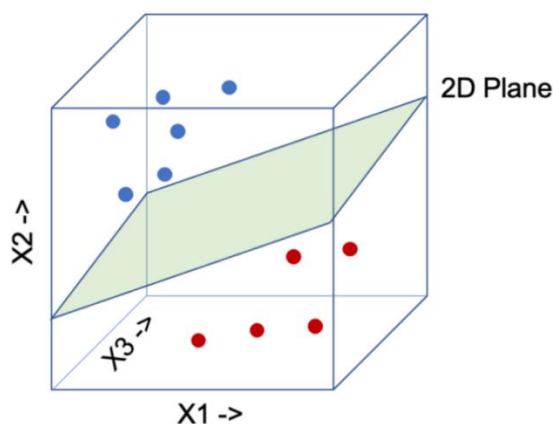
Linear and Non-Linear Classification, Logistic Regression, SVM, Decision Trees, Random Forests

We often need to analyze data and make decisions on the new data as to which category or class it belongs to. For example, given Gene Expression data, we may want to predict the type of cancer a patient may have. Depending upon the complexity of the dataset, the number of possible categories or classes that each data item belongs to may be two or quite a bit more. If there are two classes, then the classification is binary i.e., true, false, or yes and no e.g., does the patient have cancer? The answer is one of two possible values i.e., yes or no.

For classification, the next question is, are the different classes separable linearly. For example, for a 2-D two class problem, can we separate the 2-classes by a straight line? If so, the problem can be solved by implementing a linear classifier such as Logistic regression, or a Support Vector Machine (SVM).



In higher dimensions, the linear decision boundary between the classes is modeled as a hyperplane. For example, the picture below shows a 2D plane separating the two classes for a 3-D data.



For linear classification, there are two popular algorithms. One is Logistic Regression and the other is SVM. We first review the theory behind the two and then will program and visualize the decision boundaries on datasets.

Linear Decision Boundary - Logistic Regression:

Logistic regression is generally used for binary classification problems.

Suppose $x_{i1}, x_{i2}, \dots, x_{in}$ are n features corresponding to some data (i.e., data is in n -dimensional space). Then in the linear model, the predicted response would be the linear combination of those features based on the set of parameters β_i as,

$$h(x_i) = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \dots + \beta_n x_{in} \quad (1)$$

If we define, the data and the hyperplane coefficients in vector form as:

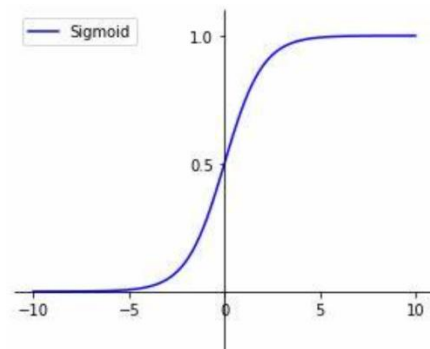
$$x_i = \begin{bmatrix} 1 \\ x_{i1} \\ x_{i2} \\ \vdots \\ x_{in} \end{bmatrix} \quad \text{and} \quad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{bmatrix}$$

Then equation (1) can be written in matrix form as:

$$h(x_i) = \beta^T x_i = [\beta_0 \ \beta_1 \ \dots \ \beta_n] \begin{bmatrix} 1 \\ x_{i1} \\ x_{i2} \\ \vdots \\ x_{in} \end{bmatrix} = \beta_0 + \beta_1 x_{i1} + \dots + \beta_n x_{in}$$

To convert our hypothesis into probability so that it varies between 0 and 1, we can pass $h(x_i)$ to a sigmoid function. Sigmoid function is also called the Logistic function, and that is why the decision making algorithm is called Logistic Regression.

Note that $\sigma_g(x) = \frac{1}{1+e^{-x}}$



Lets redefine our hypothesis as

$$h(x_i) = \sigma_g(\beta^T x_i) \quad (2)$$

Suppose y_i is the output for an input of x_i , then we can define the conditional probabilities on the two labels as:

$$P(y_i = 1|x_i, \beta) = h(x_i) \quad (3)$$

$$P(y_i = 0|x_i, \beta) = 1-h(x_i) \quad (4)$$

We can combine equation 3 and 4 into one equation as:

$$P(y_i |x_i, \beta) = (h(x_i))^{y_i} (1-h(x_i))^{1-y_i} \quad (5)$$

If we have “ m ” training samples, then our goal is to determine β such that the probability of correctly classifying data in equation (5) is maximized. We formally describe this as the maximum likelihood.

$$Ll(\beta) = \prod_{i=1}^m P(y_i |x_i, \beta)$$

$$\text{Or } Ll(\beta) = \prod_{i=1}^m (h(x_i))^{y_i} (1-h(x_i))^{1-y_i} \quad (6)$$

We want to maximize likelihood in Equation (6) and determine the β that will maximize it. Since Equation (6) is a multiplication of the probabilities, it can approach 0 if the number of training samples is large. To overcome this, we try to maximize the log (likelihood) as:

$$L(\beta) = \log(Ll(\beta)) = \sum_{i=1}^m y_i \log h(x_i) + (1 - y_i) \log(1-h(x_i))$$

The maximization of log of likelihood can be converted to a minimization problem by putting a minus sign before the maximization equation.

$$L(\beta) = - \sum_{i=1}^m y_i \log h(x_i) + (1 - y_i) \log(1-h(x_i)) \quad (7)$$

The optimal value of β that minimizes the above loss function will define the separating hyperplane between the two classes.

$$\beta = \underset{\beta}{\operatorname{argmin}} (L(\beta)) \quad (8)$$

Since equation (7) is a transcendental equation, it does not have closed form solution. We can find a near optimal solution by using the **Gradient Descent** technique.

In Gradient Descent, the β values are randomly initialized and iteratively improved until the solution converges. The updated equation appears as

$$\beta = \beta - \alpha \nabla(L(\beta)) \quad (9)$$

$$\nabla_{\beta}(L(\beta)) = \frac{\partial L}{\partial \beta}(L(\beta))$$

$$L(\beta) = - \sum_{i=1}^m y_i \log(h(x_i)) + (1 - y_i) \log(1 - h(x_i))$$

$$h(x_i) = \frac{1}{1 + e^{-\beta^T x_i}}$$

In the lecture, we derived the $\frac{\partial L(\beta)}{\partial \beta}$ as:

$$\frac{\partial L(\beta)}{\partial \beta} = - \sum_{i=1}^m (y_i - a_i) x_i$$

where $a_i = h(x_i)$ = actual output after passing the data through equation 2, y_i = expected output. Equation 9 will be repeated executed for a fixed number of iterations. The final value of β will define the separating hyperplane. The decision as to which class a given data belongs to will be determined by equation 2. If the value after passing through the sigmoid is > 0.5 , the answer is yes, otherwise no.

Programming Logistic Regression:

Create a Python application called LogisticReg. Add a class to the project called Utils.py with the following code in it. The Utils class has functions for reading two types of 2-D data. One is from a csv file called DataSet.csv. This will be provided to you on the CPSC 552 kiwi web site. It has simple two class data with two features in each row as shown below:

The screenshot shows an Excel spreadsheet with a dataset. The first three columns are labeled A, B, and C. Column A contains values ranging from 4.0 to 7.0. Column B contains values ranging from 3.8128 to 7.1865. Column C contains binary values (0 or 1). The data is as follows:

	A	B	C
40	7.8593	3.8128	1
41	6.9999	3.2406	1
42	5.5061	2.9052	1
43	4.9241	2.6882	1
44	6.6447	3.8325	1
45	7.6822	4.5428	1
46	8.0364	5.7857	1
47	8.9221	6.5552	1
48	7.8593	5.253	1
49	6.5941	5.2333	1
50	6.0374	4.7598	1
51	2.7227	4.5822	0
52	1.9383	3.6549	0
53	1.6852	2.9841	0
54	4.3168	4.4244	0
55	3.4312	3.7536	0
56	5.4808	5.2728	0
57	4.1144	4.8387	0
58	3.2034	4.4244	0
59	4.1144	5.3911	0
60	5.1012	6.0817	0
61	4.8988	5.5687	0
62	5.9615	6.4565	0
63	5.7591	6.0028	0
64	6.6953	6.7722	0
65	5.7338	6.6538	0
66	6.6194	7.1471	0
67	7.2014	7.5219	0
68	7.2014	6.8314	0
69	8.5931	7.6206	0
70	7.7581	7.1865	0

The Utils class has another function to create 2-D data by combining data from a few normal distributions.

```
import csv
import numpy as np
import matplotlib.pyplot as plt

class Utils(object):
    def readData(self,filename): # return as numpy array
        with open(filename,"r") as csvfile:
            lines = csv.reader(csvfile)
            data = list(lines)
            for i in range(len(data)): # convert data to float
                data[i] = [float(x) for x in data[i]]
            return np.array(data)

    def readDataRandom(self):
        np.random.seed(12)
        num_observations = 50
        x1 = np.random.multivariate_normal([1.5, 4], [[1, .75],[.75, 1]],
num_observations)
        print(x1.shape)
        x2 = np.random.multivariate_normal([1, 2], [[1, .75],[.75, 1]],
num_observations)
        data = np.vstack((x1, x2)).astype(np.float32)
        print(data.shape)
        labels = np.hstack((np.zeros(num_observations),
np.ones(num_observations)))
        print(labels.shape)
        dataWithLabels = np.hstack((data,labels.reshape(labels.shape[0],1)))
        #print(dataWithLabels.shape)
        #print(labels.shape)
        plt.figure(figsize=(12,8))
        plt.scatter(data[:, 0], data[:, 1], c = labels, alpha = .4)
        plt.show()
        return dataWithLabels

    def normalizeData(self,X):
        min = np.min(X, axis = 0)
        max = np.max(X, axis = 0)
        normX = 1 - ((max - X)/(max-min))
        return normX

    def plot_result(self,X, y, beta):
        x_0 = X[np.where(y == 0.0)]
        x_1 = X[np.where(y == 1.0)]

        # plot the data points
        plt.scatter([x_0[:, 1]], [x_0[:, 2]], c='b', label='y = 0')
        plt.scatter([x_1[:, 1]], [x_1[:, 2]], c='r', label='y = 1')

        # plot the decision boundary
        x1 = np.arange(0, 1, 0.1)
        x2 = -(beta[0,0] + beta[0,1]*x1)/beta[0,2]
        plt.plot(x1, x2, c='g', label='reg line')
```

```
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend()
plt.show()
```

In the LogisticReg.py, type the following code.

```
import sys
from Utils import Utils
import numpy as np

def sigmoid(beta, X): # logistic function
    return 1.0/(1 + np.exp(-np.dot(X, beta.T)))

def gradientBeta(beta, X, y): # dL/dbeta
    a = sigmoid(beta, X)
    part1 = a - y.reshape(X.shape[0], 1)
    grad = np.dot(part1.T, X)
    return grad

def logLoss(beta, X, y):
    a = sigmoid(beta, X) # actual output
    loss = -(y * np.log(a) + (1 - y) * np.log(1 - a))
    return np.sum(loss)

def trainUsingGradientDescent(X, y, beta, num_iter, alpha = .01):
    loss = logLoss(beta, X, y)
    for i in range (num_iter):
        beta = beta - (alpha * gradientBeta(beta, X, y))
        loss = logLoss(beta, X, y)
        if (i%10 == 0):
            print('iter = ' + str(i) + ' loss=' + str(loss))
    return beta

def classify_data(beta, X): # 0 or 1
    a = sigmoid(beta, X) # actual output
    decision = np.where(a >= .5, 1, 0)
    return decision

def main():
    utils = Utils()

    #data = utils.readData('d:/DonAlpha/pythonam2/data/DataSet.csv') # load data
    data = utils.readDataRandom() # load data

    X = utils.normalize_data(data[:,0:2]) # or[:, :-1] normalize data - scale
    # between 0-1
    X = np.hstack((np.ones((1,X.shape[0])).T, X)) # add 1's column to data

    Y = data[:, -1] # expected output, -1 means last column
    beta = np.zeros((1,X.shape[1])) # (1,3) in this example
    beta = trainUsingGradientDescent(X, Y, beta, 1000) # optimize using gradient
    # descent
    print("Logistic Regression Model coefficients:", beta)
```

```

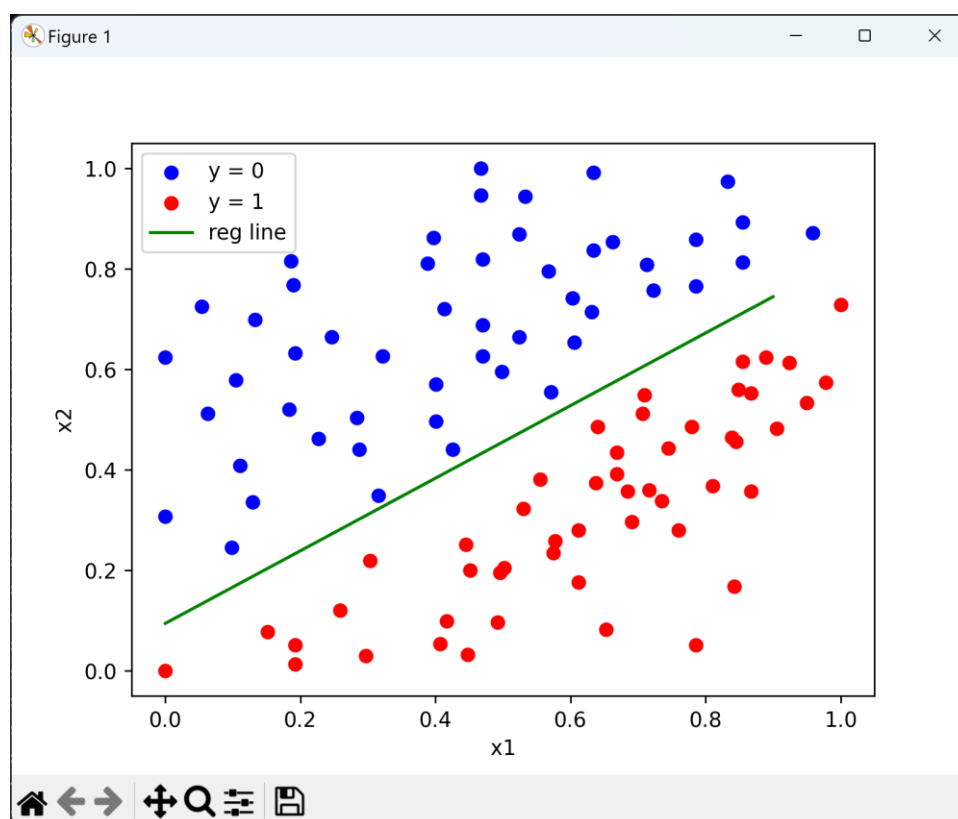
y_predicted = classify_data(beta, X) # predictions by the trained model
print("Number of correct predictions = ", str(np.sum(Y ==
y_predicted.reshape(Y.shape[0]))/len(X)*100) + '%')

utils.plot_result(X, Y, beta) # plot results

if __name__ == "__main__":
    sys.exit(int(main() or 0))

```

If you run the program, the data will be read from the csv file, then the main program uses Gradient Descent to train the algorithm i.e., determine the β that minimize the loss function. The decision boundary generated by the separating hyperplane (line in this case, since data is 2-D), is shown in green color below.



What if the given data is not linearly separable. To test how Logistic Regression performs in such a case, use the data that is generated by the readRandomData function. In the LogisticReg.py, comment and uncomment the following lines and run the program again.

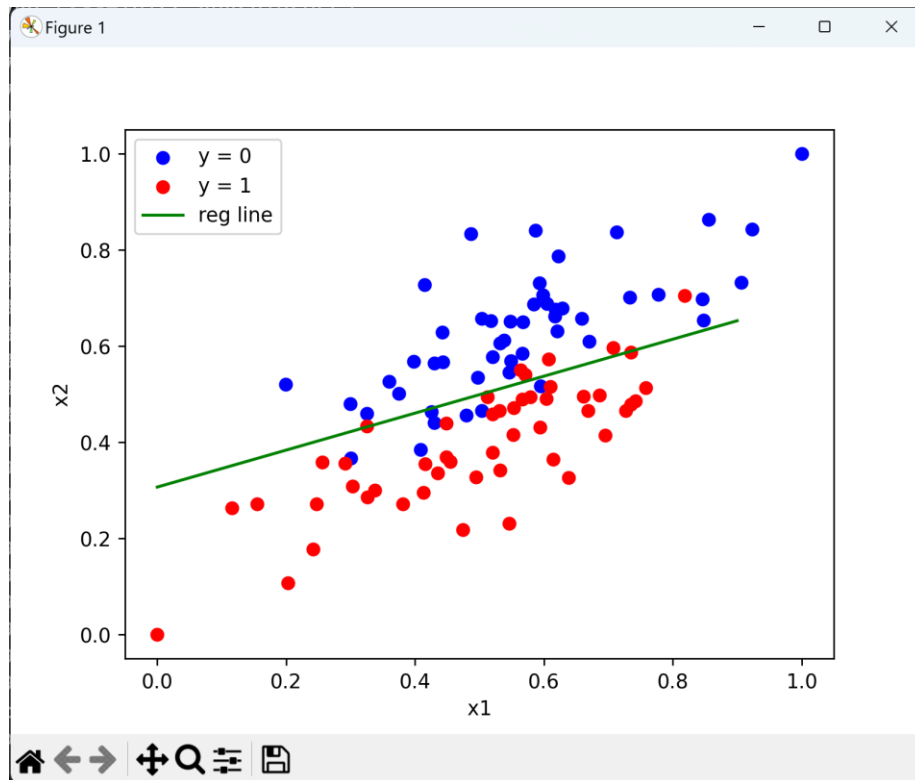
```

#data = utils.readData('d:/DonAlpha/pythonam2/data/DataSet.csv') # load data
data = utils.readDataRandom() # load data

```

Now, the result appears as:

Note that some data is misclassified as LogisticRegression can only do Linear decision boundary.



```

C:\WINDOWS\system32\cmd. x + v
iter = 890 loss=10617.077817612753
iter = 900 loss=10645.54081224325
iter = 910 loss=10673.82212422596
iter = 920 loss=10701.924164562026
iter = 930 loss=10729.849286232404
iter = 940 loss=10757.599786328023
iter = 950 loss=10785.177908080232
iter = 960 loss=10812.585842796834
iter = 970 loss=10839.825731708703
iter = 980 loss=10866.89966773169
iter = 990 loss=10893.80969714828
Logistic Regression Model coefficients: [[ 4.6006752  5.7674587 -15.0063307]]
Number of correct predictions = 87.0%

```

Logistic Regression is built into the Sklearn library. To use the built-in capability, add a file to the project called LogisticSkLearn.py with the following code in it.

```

import sys
import sys
from Utils import Utils
import numpy as np
from sklearn.linear_model import LogisticRegression

def main():
    utils = Utils()
    data = utils.readDataRandom() # load data

```



```

X = data[:,0:2] # use only two features, for visualization purposes
Y = data[:, -1] # expected output, -1 means last column

model = LogisticRegression(solver='lbfgs')
model.fit(X, Y)

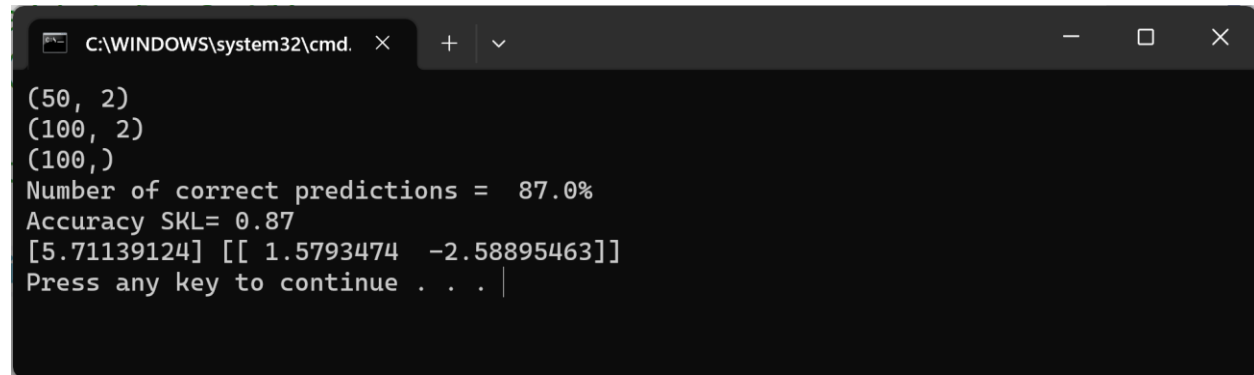
preds = model.predict(X) # predictions by the trained model
accuracy = (preds == Y).mean()
print("Number of correct predictions = ", str(np.sum(Y ==
preds.reshape(Y.shape[0]))/len(X)*100) + '%')

print("Accuracy SKL= " + str(accuracy))
print(model.intercept_, model.coef_)

if __name__ == "__main__":
    sys.exit(int(main() or 0))

```

Set the LogisticSkLearn.py as the startup file and run the program. You will see the following result.



```

C:\WINDOWS\system32\cmd. X + v - □ X
(50, 2)
(100, 2)
(100,)
Number of correct predictions = 87.0%
Accuracy SKL= 0.87
[5.71139124] [[ 1.5793474 -2.58895463]]
Press any key to continue . . . |

```

Support Vector Machine (SVM)

SVM is fundamentally a linear classifier for binary classification. It can be adapted to non-linear classification via the kernel trick, and also for multi-class classification via one-vs-rest approach. SVM is a maximum margin classifier that tries to come up with an equation of the hyperplane separating the two classes such that the margin between the two classes is maximized.

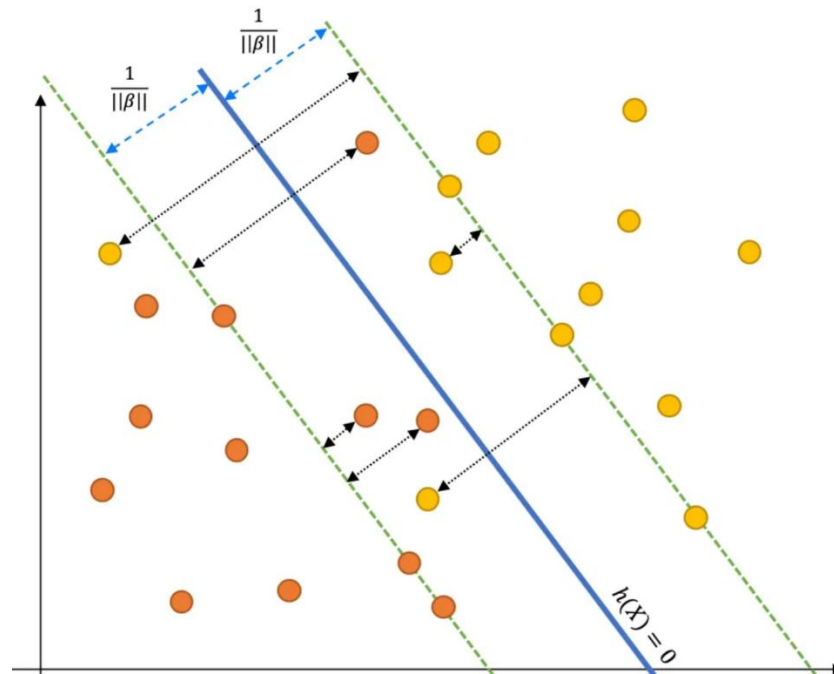
Similar to Logistic Regression, the equation of the hyperplane separating the two classes is:

$$h(x^i) = \boldsymbol{\beta}^T \mathbf{x}_i + \beta_0 = [\beta_1 \dots \beta_n] \begin{bmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{in} \end{bmatrix} = \beta_1 x_{i1} + \dots + \beta_n x_{in} + \beta_0$$

In the literature, the β s are often called the weights, so the hyperplane equation for the SVM is often described as:

$$h(x^i) = \mathbf{w}^T \mathbf{x}_i + b$$

SVM is a maximum margin classifier such that it tries to maximize the margin between the two classes. The data points that are on the margin are called support vectors as shown below.



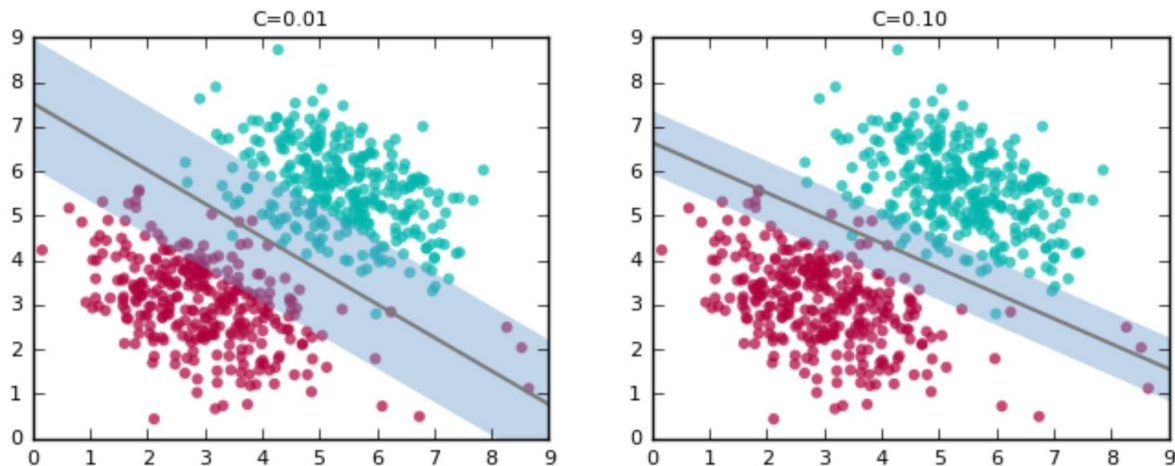
SVM is effective in cases where the number of dimensions is high but the training data is relatively less. To determine the optimum hyperplane, SVM solves the following optimization equation using a technique such as Gradient Descent.

$$\min_{\mathbf{w}} \|\mathbf{w}\|^2 \quad \text{subject to } y_i (\mathbf{w}^T \mathbf{x}_i + b) \geq 1 \text{ for } i = 1 \dots N$$

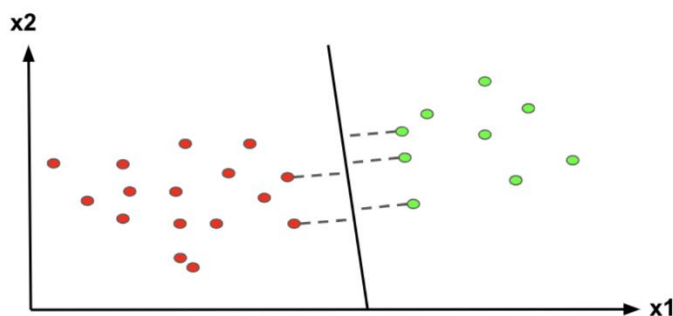
SVM classifier is built into the sklearn library. We have to understand its hyperparameters to be able to use it effectively.

SVM Hyperparameters:

C : This parameter controls the trade-off between decision boundary and misclassification term. For large values of C, the optimization produces a smaller-margin hyperplane. For small C, the optimizer produces a larger-margin separating hyperplane.

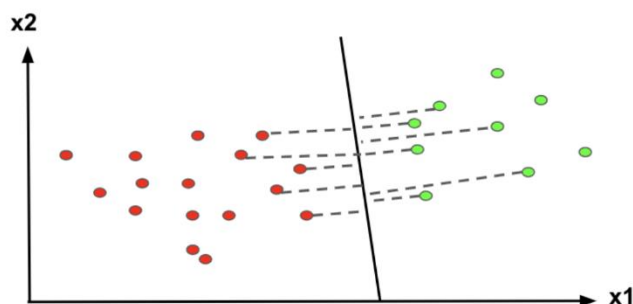


Gamma: Gamma controls the number of points to use to construct the hyperplane. Higher value of gamma causes the radius of influence to be limited to only support vectors. This may cause the model to overfit where the model produces high accuracy on the training data but does not do as well on the test data. The training accuracy usually reduces with the increasing value of gamma. The lower value of gamma results in high radius of influence. We have to experiment with the value of gamma that will result in high test accuracy.



High Gamma

- only near points are considered.



Low Gamma

- far away points are also considered

Kernel: Kernel gives the capability to SVM to create non linear decision boundaries.

SVM Analysis on the Iris DataSet:

Create a Python application called SupportVectorMachine. Add a class to the project called IrisSVM with the following code in the IrisSVM.py.

```
from sklearn import svm, datasets
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
from sklearn.decomposition import PCA
import itertools
import pandas as pd
import matplotlib

class IrisSVM(object):
    def __init__(self) -> None:
        df = pd.read_csv("d:/pythonam3/data/iris.csv")
        df1 = df.iloc[:,0:4].astype(float)
        self.X = df1.to_numpy()
        #---separate out the last column
        df2 = df.iloc[:,4]
        self.y = df2.to_numpy()

    def make_meshgrid(self, x, y, h=.02):
        x_min, x_max = x.min() - 1, x.max() + 1
        y_min, y_max = y.min() - 1, y.max() + 1
        xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                             np.arange(y_min, y_max, h))

        return xx, yy

    def plot_contours(self, ax, model, xx, yy, **params):
        """Plot the decision boundaries for a classifier.
        Parameters
        -----
        ax: matplotlib axes object
        model: a classifier
        xx: meshgrid ndarray
        yy: meshgrid ndarray
        params: dictionary of params to pass to contourf, optional
        """
        Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
        Z = Z.reshape(xx.shape)
        out = ax.contourf(xx, yy, Z, **params)
        return out

    def plotIrisDecisionBoundaries(self):
        iris = datasets.load_iris()
        # first two features for visualization
        X = iris.data[:, :2]
        y = iris.target

        C = 1.0 # SVM regularization parameter
        models = (svm.SVC(kernel='linear', C=C),
                  svm.LinearSVC(C=C),
                  svm.SVC(kernel='rbf', gamma='auto', C=C),
                  svm.SVC(kernel='poly', gamma='auto', degree=3, C=C))
```

```

models = (clf.fit(X, y) for clf in models)

# title for the plots
titles = ('SVC with linear kernel',
          'LinearSVC (linear kernel)',
          'SVC with RBF kernel',
          'SVC with polynomial (degree 3) kernel')

# Set-up 2x2 grid for plotting.
fig, sub = plt.subplots(2, 2)
plt.subplots_adjust(wspace=0.4, hspace=0.4)

X0, X1 = X[:, 0], X[:, 1]
xx, yy = self.make_meshgrid(X0, X1)

for clf, title, ax in zip(models, titles, sub.flatten()):
    self.plot_contours(ax, clf, xx, yy,
                       cmap=plt.cm.coolwarm, alpha=0.8)
    ax.scatter(X0, X1, c=y, cmap=plt.cm.coolwarm, s=20, edgecolors='k')
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xlabel('Sepal length')
    ax.set_ylabel('Sepal width')
    ax.set_xticks(())
    ax.set_yticks(())
    ax.set_title(title)

plt.show()

def svmAccuracy(self):
    iris = datasets.load_iris()
    X = iris.data[:, :2] # first two features, gives around 82% accuracy
    X = iris.data[:, :4] # all 4 features, gives around 96-98% accuracy
    y = iris.target
    C = 1.0 # SVM regularization parameter
    models = (svm.SVC(kernel='linear', max_iter=4000, C=C),
              svm.LinearSVC(C=C, max_iter=4000),
              svm.SVC(kernel='rbf', gamma='auto', C=C),
              svm.SVC(kernel='poly', gamma='auto', degree=3, C=C))
    models = (model.fit(X, y) for model in models)
    for model in models:
        Z = model.predict(X)
        countCorrect = np.sum(Z==y)
        print("accuracy of " + str(model) + " = " + str(countCorrect/len(Z)))

def visuvalize_sepal_data(self):
    iris = datasets.load_iris()
    X = iris.data[:, :2] # we only take the first two features.
    y = iris.target
    #plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.coolwarm)
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.winter)
    plt.xlabel('Sepal length')
    plt.ylabel('Sepal width')
    plt.title('Sepal Width & Length')
    plt.show()

def visuvalize_petal_data(self):
    iris = datasets.load_iris()

```

```

X = iris.data[:, 2:] # we only take the last two features.
y = iris.target
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.winter)
plt.xlabel('Petal length')
plt.ylabel('Petal width')
plt.title('Petal Width & Length')
plt.show()

def visualizeWithPCA(self):
    yy = np.zeros(self.X.shape[0])
    for i in range(len(yy)):
        if self.y[i] == 'setosa':
            yy[i] = 0
        if self.y[i] == 'versicolor':
            yy[i] = 1
        if self.y[i] == 'virginica':
            yy[i] = 2

    data = self.X # load data
    X = data[:,0:4]
    # plot the first three PCA dimensions
    fig = plt.figure(1, figsize=(8, 6))
    X_reduced = PCA(n_components=3).fit_transform(X)
    ax = plt.axes(projection='3d')
    # defining axes
    x = X_reduced[:, 0]
    y = X_reduced[:, 1]
    z = X_reduced[:, 2]
    mycmap = matplotlib.colors.ListedColormap(["blue", "red", "green"])
    ax.scatter(x, y, z, c = yy, cmap=mycmap)
    ax.set_title('PCA - IRIS (3-components)')
    plt.show()

def svmWithPCAReducedFeatures(self):
    iris = datasets.load_iris()
    X = iris.data
    y = iris.target
    pca = PCA(n_components=3)
    pca.fit(X)
    X_reduced = pca.transform(X)
    C = 1.0 # SVM regularization parameter
    models = [svm.SVC(kernel='linear', max_iter=4000, C=C),
               svm.LinearSVC(C=C, max_iter=4000),
               svm.SVC(kernel='rbf', gamma='auto', C=C),
               svm.SVC(kernel='poly', gamma='auto', degree=3, C=C)]

    i = 0
    for model in models:
        models[i] = model.fit(X_reduced, y)
        i = i + 1
    print(models[0])
    i = 0
    for model in models:
        Z = model.predict(X_reduced)
        countCorrect = np.sum(Z==y)
        print("accuracy of model " + str(i) + " = " + str
(countCorrect/len(Z)))
        i = i + 1
    X_r = pca.transform(X[125].reshape(1,4)) # predict class for data 125

```

```
Z125 = model.predict(X_r)
print("class for data 125 = " + str(Z125[0]))
```

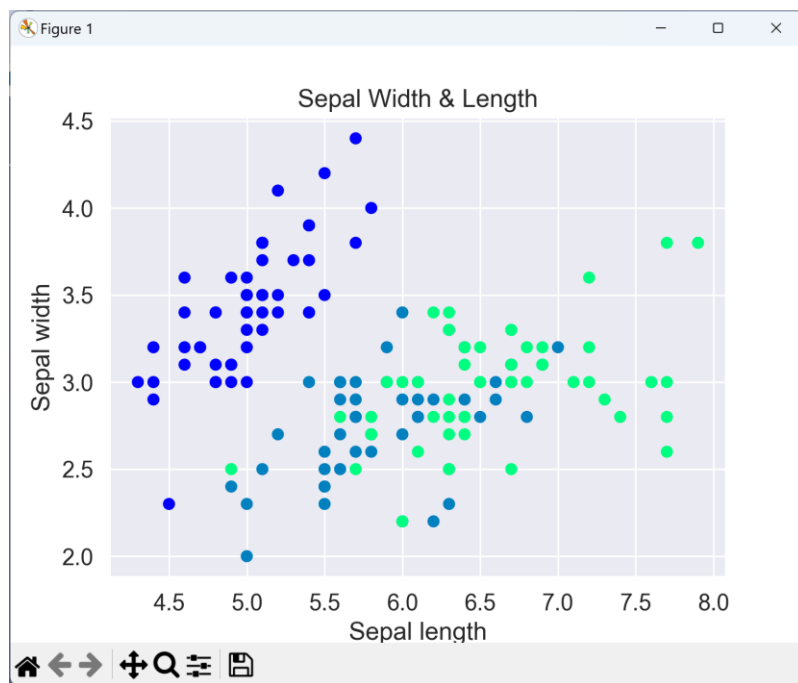
Type the following code in SupportVectorMachine.py

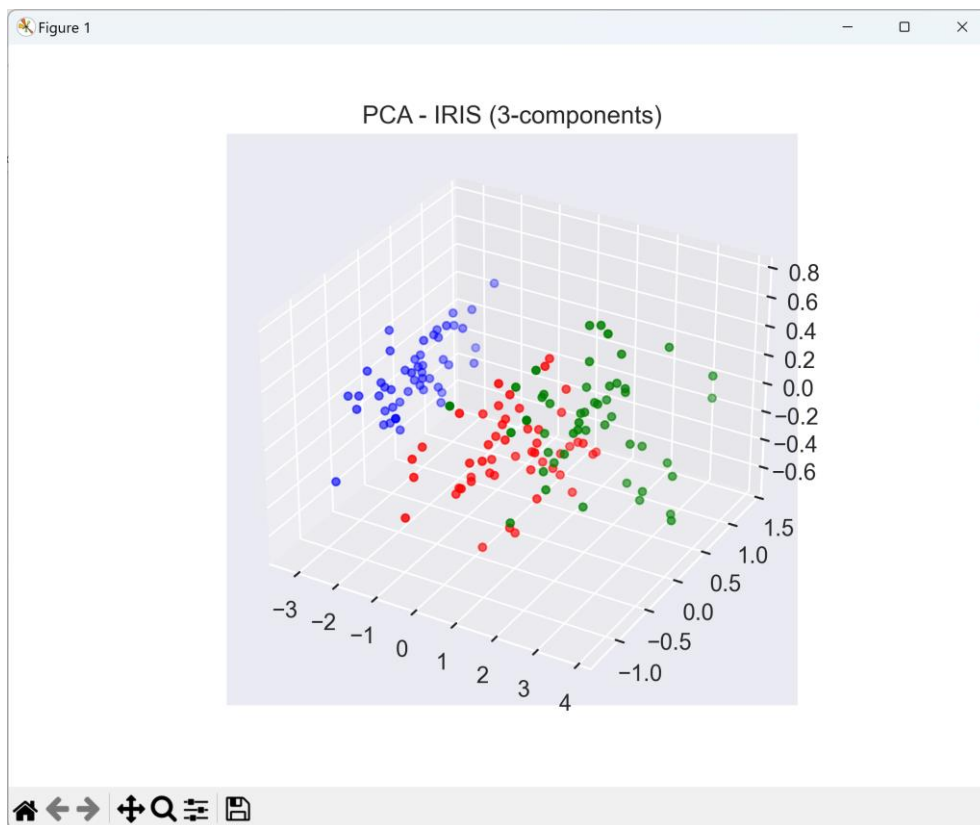
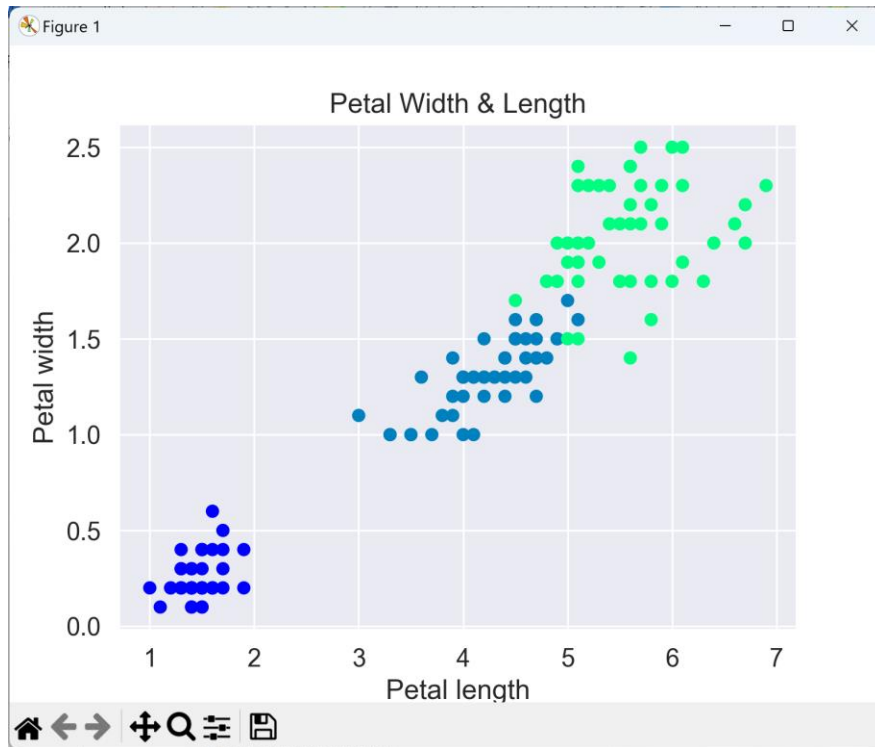
```
import sys

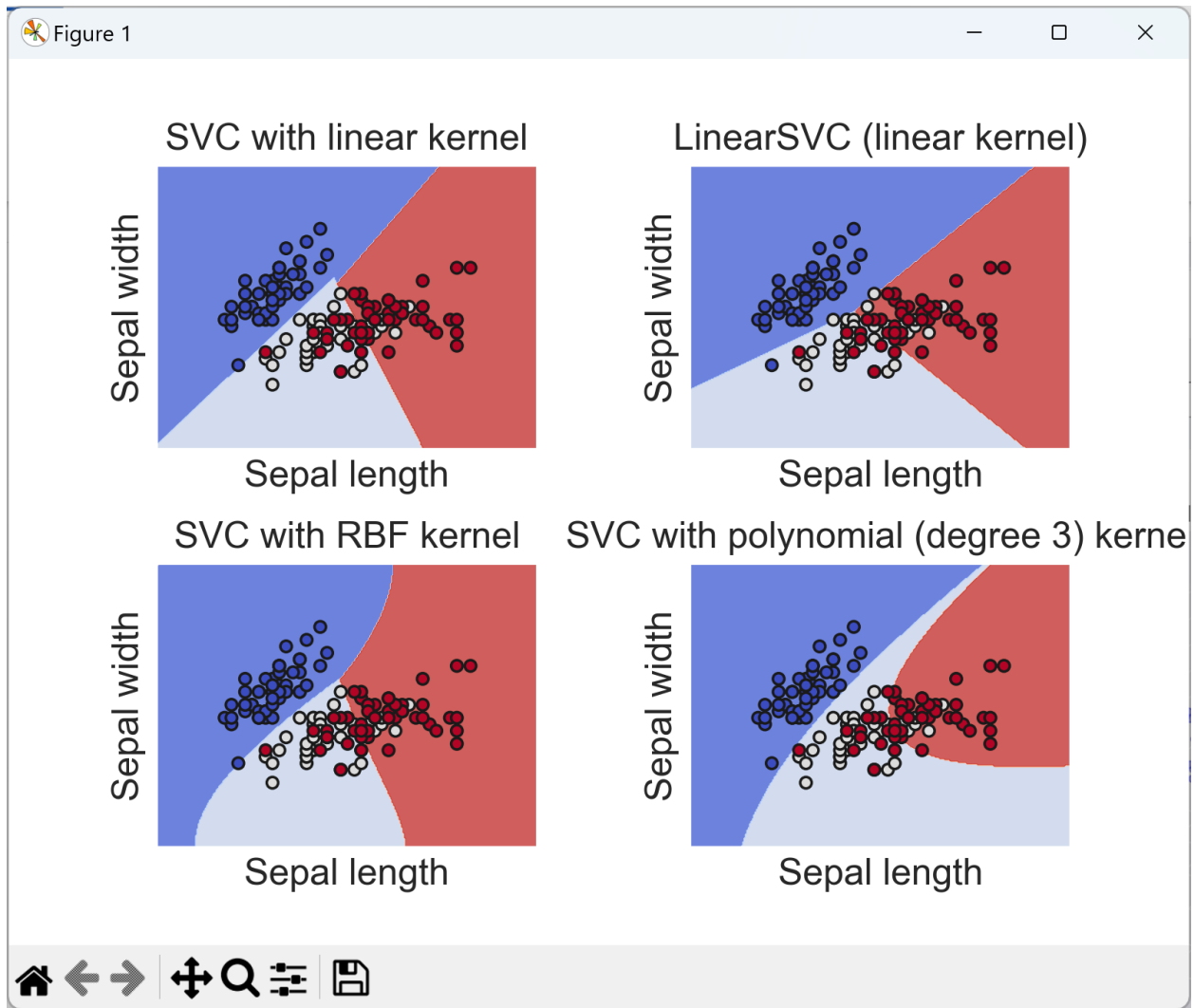
def main():
    irissvm = IrisSVM()
    irissvm.visualize_sepal_data()
    irissvm.visualize_petal_data()
    irissvm.visualizeWithPCA()
    irissvm.svmAccuracy()
    irissvm.plotIrisDecisionBoundaries()
    irissvm.svmWithPCAReducedFeatures()

if __name__ == "__main__":
    sys.exit(int(main() or 0))
```

If you run the program, you will see the following outputs.







```
C:\WINDOWS\system32\cmd. x + v
accuracy of SVC(kernel='linear', max_iter=4000) = 0.9933333333333333
accuracy of LinearSVC(max_iter=4000) = 0.9666666666666667
accuracy of SVC(gamma='auto') = 0.9866666666666667
accuracy of SVC(gamma='auto', kernel='poly') = 0.98
C:\ProgramData\Anaconda3\envs\pytorch1xA\lib\site-packages\sklearn\svm\_base.py:1244: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.
  warnings.warn(
SVC(kernel='linear', max_iter=4000)
accuracy of model 0 = 0.9866666666666667
class for data 125 = 2
accuracy of model 1 = 0.9533333333333334
class for data 125 = 2
accuracy of model 2 = 0.98
class for data 125 = 2
accuracy of model 3 = 0.9533333333333334
class for data 125 = 2
Press any key to continue . . . |
```

Decision Trees and Random Forests

Decision trees are simple supervised learning models that can be used for classification or regression. Simple decisions are made on the value of features to predict the target class for a classification problem. A tree can be seen as a piecewise linear approximation creating an overall non-linear decision boundary.

Criteria for selecting a feature and the value to split is either based on Gini Index, Information gain or Entropy. The Gini Index is defined as:

$$Gini = 1 - \sum_{i=1}^C (p_i)^2$$

Where C is the number of classes. Gini measures the probability of misclassification, so a lower value is better. For example, if we have to decide which feature should be selected to split the root node of the decision tree, the feature that results in lowest Gini index will be selected. Sklearn library can drae the decision tree with Gini values.

An extension of the decision tree is a model known as a random forest, which is a collection of decision trees. To create multiple decision trees, we take bootstrapped samples from the original dataset. (repeated sampling). Bootstrapping is loosely based on the law of large numbers, which states that if you sample over and over again, your data should approximate the true population data. For each bootstrapped sample, we build a decision tree. using a random subset of the predictor variables. We average the predictions of each tree to come up with a final decision. Random Forest generally works better than a Decision Tree.

Create a new Python application called DecisionTreesRandomForests. Add a file to the project called Utils with the following code in it.

```
import numpy as np
import matplotlib.pyplot as plt

def visualize_classifier(model, X, y, ax=None, cmap='rainbow'):
    ax = ax or plt.gca()

    # Plot the training points
    ax.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=cmap,
               clim=(y.min(), y.max()), zorder=3)
    ax.axis('tight')
    ax.axis('off')
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()

    xx, yy = np.meshgrid(np.linspace(*xlim, num=200),
                          np.linspace(*ylim, num=200))
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

    # Create a color plot with the results
    n_classes = len(np.unique(y))
    contours = ax.contourf(xx, yy, Z, alpha=0.3,
                           levels=np.arange(n_classes + 1) - 0.5,
                           cmap=cmap, clim=(y.min(), y.max()),
                           zorder=1)
```

```
ax.set(xlim=xlim, ylim=ylim)
plt.show()
```

Type the following code in DecisionTreesRandomForests.py.

```
import sys
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
import numpy as np
import Utils
from sklearn.decomposition import PCA
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import LabelEncoder
from sklearn.ensemble import RandomForestClassifier
from sklearn import tree
import matplotlib.pyplot as plt

def main():
    df = pd.read_csv("d:/pythonam3/data/iris.csv")
    #---randomize data
    dfrandom = df #df.sample(frac=1, random_state=1119).reset_index(drop=True)
    # data read from a file is read as a string, so convert the first 4 cols to
float
    df1 = dfrandom.iloc[:,0:4].astype(float)
    X = df1.values
    #---separate out the last column
    df2 = dfrandom.iloc[:,4]
    y = df2.values
    #-----
    label_encoder = LabelEncoder()
    y_enc = label_encoder.fit_transform(df2)#
'setosa'=0,'versicolor'=1,'virginica'=2

    # Creating Train and Test datasets
    X_train, X_test, y_train, y_test = train_test_split(X,y_enc, random_state = 50,
test_size = 0.3)

    clf = DecisionTreeClassifier()
    clf.fit(X_train,y_train)
    tree.plot_tree(clf)
    plt.show()

    # Predict Accuracy Score
    y_pred = clf.predict(X_test)
    print("Train data accuracy:",accuracy_score(y_true = y_train,
y_pred=clf.predict(X_train)))
    print("Test data accuracy:",accuracy_score(y_true = y_test, y_pred=y_pred))

    # To be able to visualize data, lets reduce the dimensionality of feature space
    # from 4 to 2
```

```

x_train_scaled = StandardScaler().fit_transform(X_train)
x_test_scaled = StandardScaler().fit_transform(X_test)
pca = PCA(n_components=2)
principalComponents_train = pca.fit_transform(x_train_scaled)
principalComponents_test = pca.fit_transform(x_test_scaled)
pca_train_df = pd.DataFrame(data = principalComponents_train
                             , columns = ['principal component 1', 'principal component 2'])
pca_test_df = pd.DataFrame(data = principalComponents_test
                             , columns = ['principal component 1', 'principal component 2'])
clf_pca = DecisionTreeClassifier()
clf_pca.fit(pca_train_df.values,y_train) # train decision tree on train data
y_pred_pca = clf_pca.predict(pca_test_df.values)
print("Test data accuracy after PCA:",accuracy_score(y_true = y_test,
y_pred=y_pred_pca))

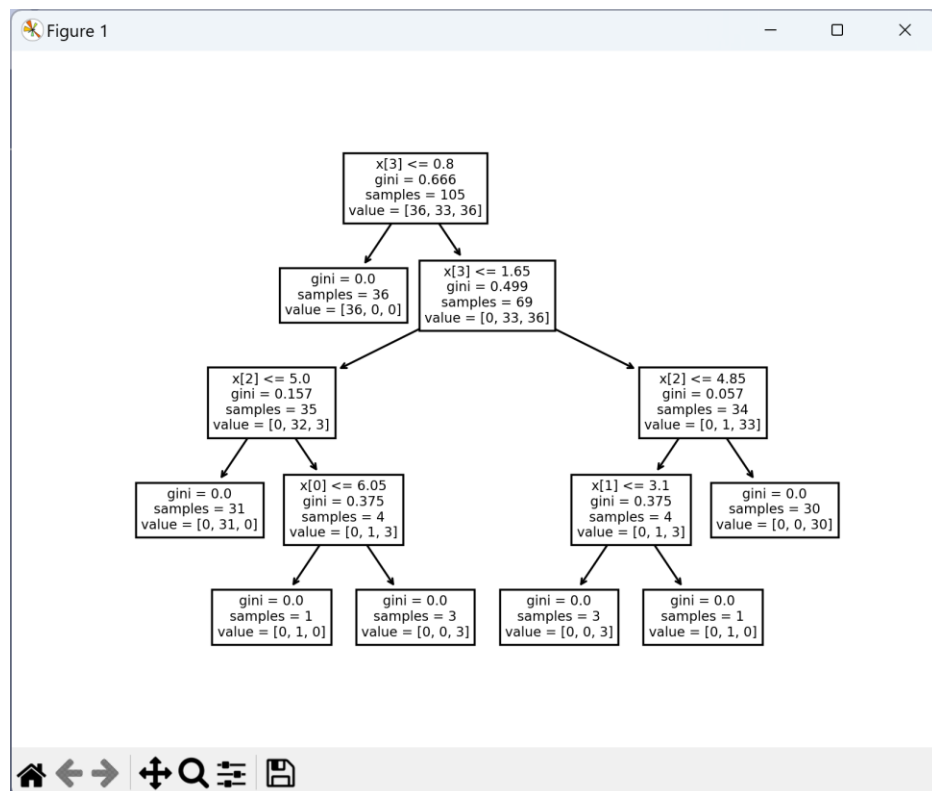
Utils.visualize_classifier(clf_pca,pca_test_df.values, y_test, ax=None,
cmap='rainbow')

#-----try random forest classifier after pca-----
modelrf = RandomForestClassifier(n_estimators=100, random_state=0)
modelrf.fit(pca_train_df.values,y_train)
Utils.visualize_classifier(modelrf, pca_test_df.values, y_test)
y_pred_rf = modelrf.predict(pca_test_df.values)
print("Test data accuracy after Random Forest:",accuracy_score(y_true = y_test,
y_pred=y_pred_rf))

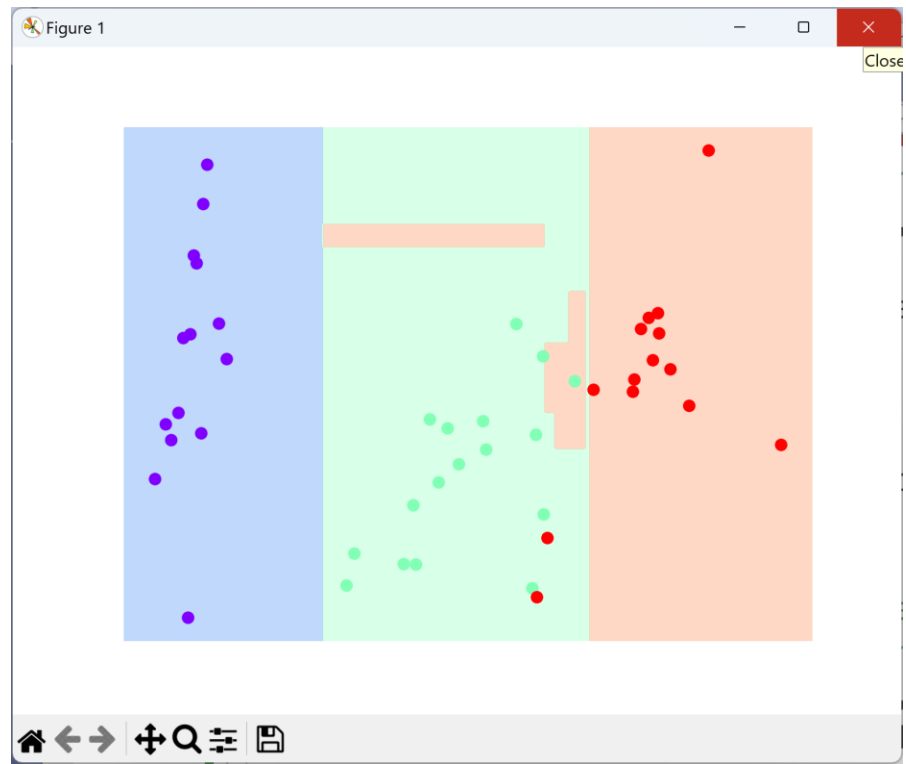
if __name__ == "__main__":
    sys.exit(int(main() or 0))

```

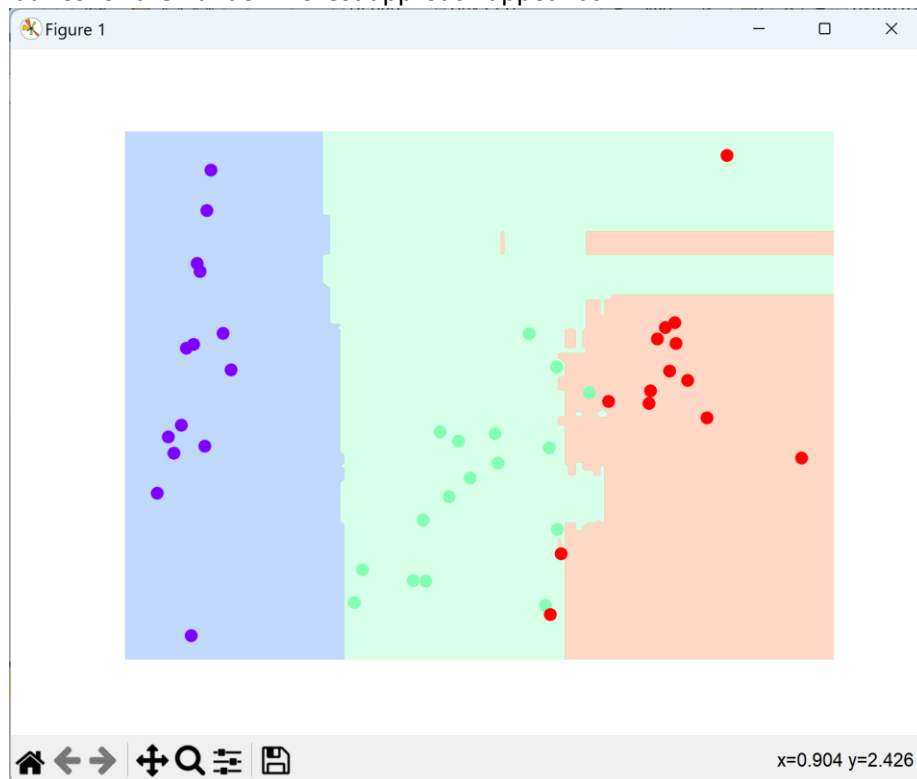
Decision tree and the Gini Index values for the Iris dataset appear as:



Decision tree decision boundaries for the Iris dataset appear as:



Decision boundaries for the Random Forest approach appear as:



```
C:\WINDOWS\system32\cmd. x + v
Train data accuracy: 1.0
Test data accuracy: 0.9555555555555556
Test data accuracy after PCA: 0.9333333333333333
D:\PythonAM3\DecisionTrees\DecisionTrees\Utils.py:21: UserWarning: The following kwargs were not used by contour: 'clim' conto
urs = ax.contourf(xx, yy, Z, alpha=0.3,
D:\PythonAM3\DecisionTrees\DecisionTrees\Utils.py:21: UserWarning: The following kwargs were not used by contour: 'clim' conto
urs = ax.contourf(xx, yy, Z, alpha=0.3,
Test data accuracy after Random Forest: 0.9333333333333333
Press any key to continue . . . |
```