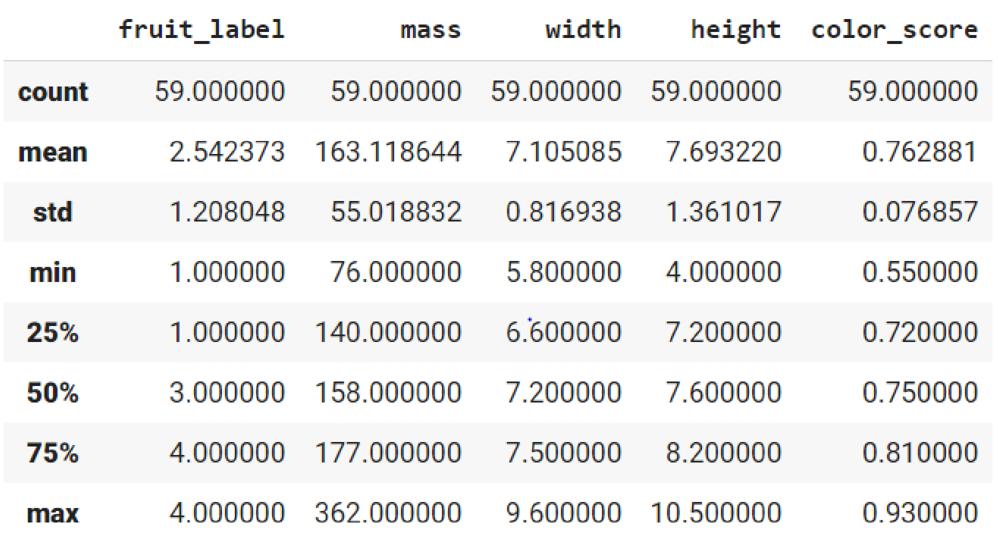
Code: SD03Q03

Question-1: Fruit classification using KNN- algorithm.

Dataset:

The fruits.xlsx dataset consist of six columns (fruit_label,fruit_name,mass,width,height and color_score) and 59 observations.

Summary of dataset



	fruit_label	mass	width	height	color_score
count	59.000000	59.000000	59.000000	59.000000	59.000000
mean	2.542373	163.118644	7.105085	7.693220	0.762881
std	1.208048	55.018832	0.816938	1.361017	0.076857
min	1.000000	76.000000	5.800000	4.000000	0.550000
25%	1.000000	140.000000	6.600000	7.200000	0.720000
50%	3.000000	158.000000	7.200000	7.600000	0.750000
75%	4.000000	177.000000	7.500000	8.200000	0.810000
max	4.000000	362.000000	9.600000	10.500000	0.930000

Exploratory analysis:

There are four different class labels for fruits, apple, mandarin, orange and lemon.

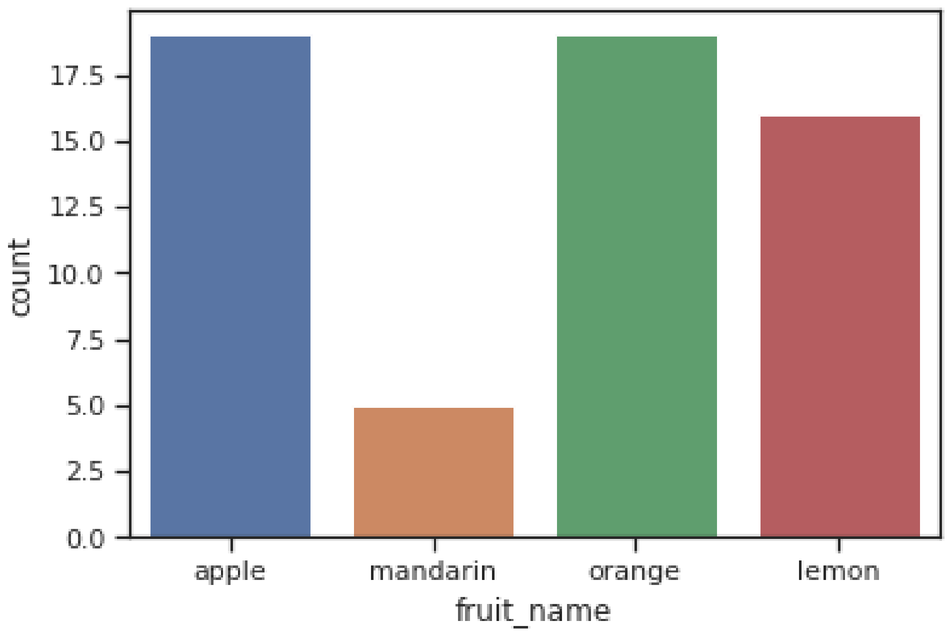


Fig 1.1 Count of each fruits.

Generating scatter plots for various combination of parameters:

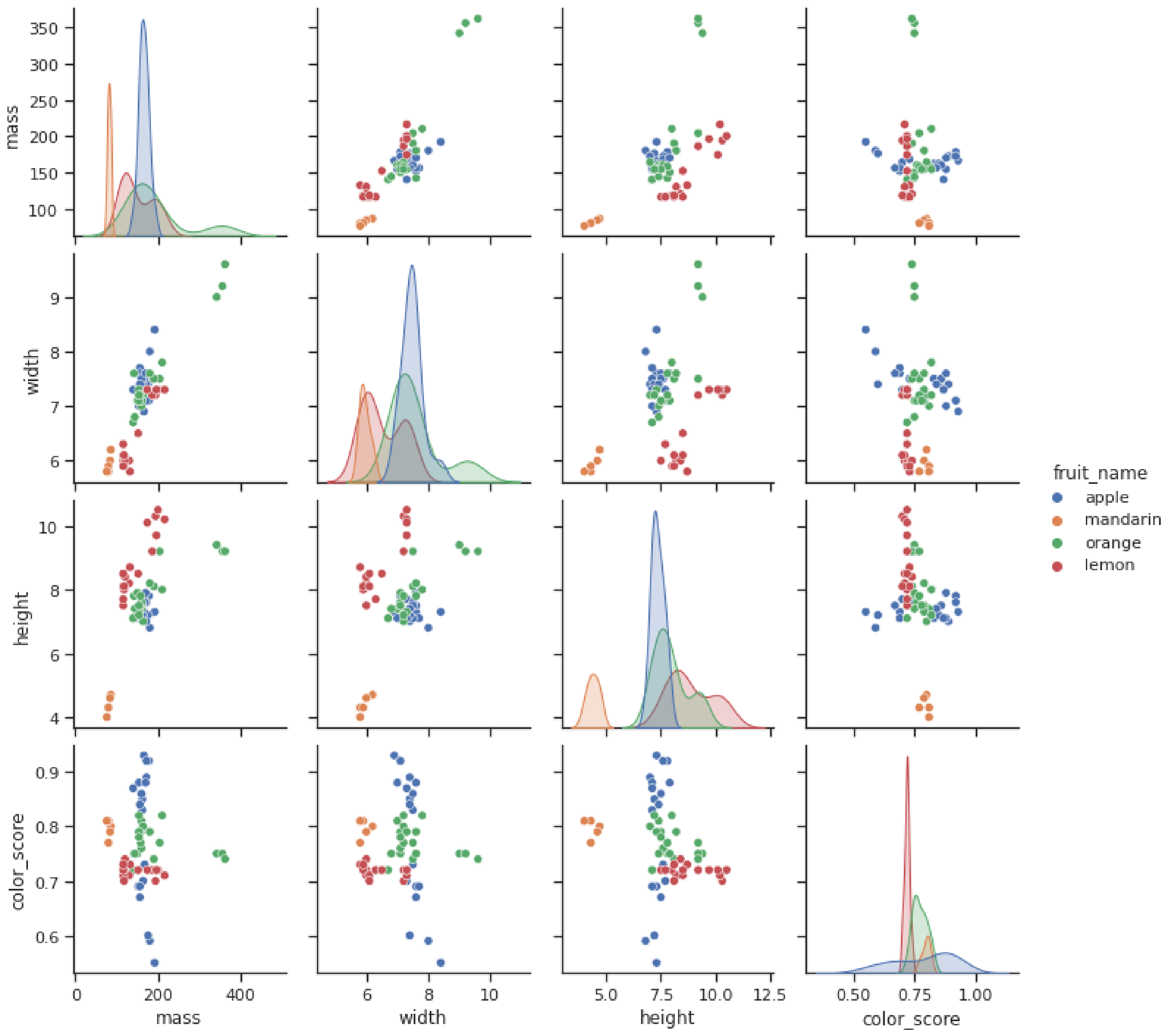


Fig 1.2 Scatter plots for various combination

The above plot shows that there is high correlation between variables. The variable mass and width are highly correlated while compared with others.

Train test split:

The 70% of the data here is used for training and the rest 30% for testing.

KNN model from scratch:

The k-nearest neighbors (KNN) algorithm is a supervised machine learning algorithm that can be used to solve both classification and regression problems.

Calculate Euclidean distance:

As the first step, KNN model calculates the distance of the new data point from every single data point within the 'fitted' training data. For calculating the distances between the data points, we will be using the Euclidean distance formula.

Sqrt(pow((ins1[x]-ins2[x]),2))

Here, ins1 and ins2 aare the first and second rows of the data.

Euclidean distance, the smaller the value, the more similar the records will be. A value of 0 means that there is no difference between the records.

Code for euclideandistance:

```
import math
def euclideanDistance(ins1,ins2,length):
  distance=0
  for x in range(length):
    distance+=pow((ins1[x]-ins2[x]),2)
  return math.sqrt(distance)
```

Getting nearest neighbors:

Neighbors for a new piece of data in the dataset are the *k* closest instances, as defined by our distance measure.
To locate the neighbors for a new piece of data within a dataset we must first calculate the distance between each record in the dataset to the new piece of data. We can do this using our distance function given above.

Code to get nearest neighbors:

```
def getNeighbors(trainingset,testinst,k):
  distances = []
  length = len(testinst)-1
  for x in range(len(trainingset)):
    dist = euclideanDistance(testinst,trainingset[x],length)
    distances.append((trainingset[x],dist))
  distances.sort(key=operator.itemgetter(1))
  neighbors = []
  for x in range(k):
    neighbors.append(distances[x][0])
```

```
 return neighbors
```

Sort the records:

Once distances are calculated, we must sort all of the records in the training dataset by their distance to the new data. We can then select the top *k* to return as the most similar neighbors.

Code to sort the records:

```
def getResponse(neighbors):
 classvotes = {}
 for x in range(len(neighbors)):
  response = neighbors[x][-1]
  if response in classvotes:
   classvotes[response]+=1
  else:
   classvotes[response]=1
 sortedvotes = sorted(classvotes.items(),key=operator.itemgetter(1),reverse=True)
 return sortedvotes[0][0]
```

Making predictions:

The most similar neighbors collected from the training dataset can be used to make predictions.
In the case of classification, we can return the most represented class among the neighbors.

Code for predictions:

```
for x in range(len(test)):
  neighbors = getNeighbors(train,test[x],k)
  result = getResponse(neighbors)
  prediction.append(result)
  predicted=result
  actual=test[x][-1]
  print('predicted= ' + repr(result) + 'actual= ' + repr(test[x][-1]))
```

The actual and the predicted outputs are printed.

Test with new instances:

Three new test instances are created and it is being predicted.

```
new_data=[[190,8,7,0.8],[170,6,70.55],[76,5,6,0.7]]
```

```
Test with new data
predicted= 4.0
predicted= 1.0
predicted= 2.0
```

Finding accuracy:

The accuracy is calculated using the predicted and actual output values.

Code to find accuracy

```
 correct=0
  for x in range(len(test)):
     if test[x][-1] == prediction[x]:
       correct+=1
  return(correct/float(len(test)))*100
```

Accuracy: 83.33

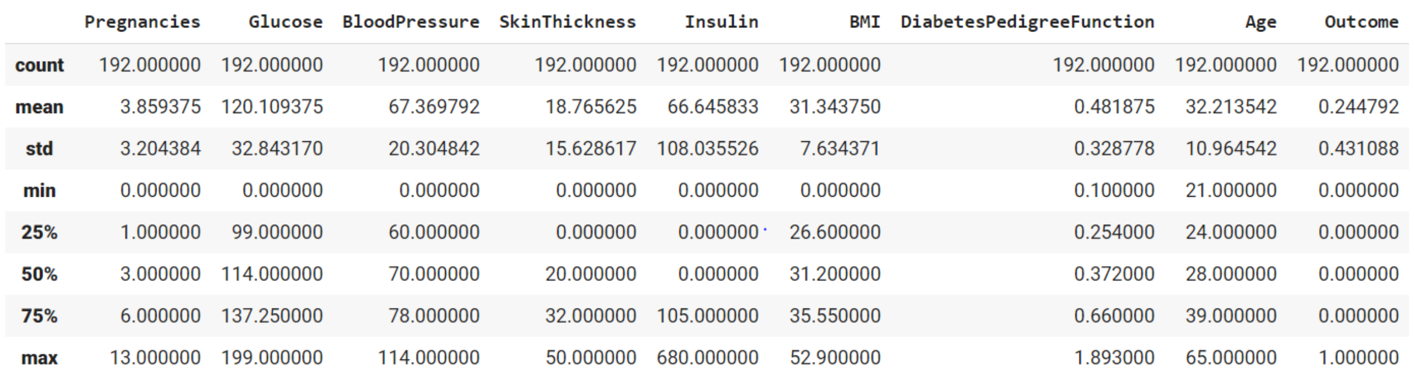
Question 2:

Diabetes prediction

Dataset:

The dataset consist of 9 variables (Pregnancies, Glucose, BloodPressure, SkinThickness, Insulin, BMI, DiabetesPedigreeFunction, Age) and 192 observations.

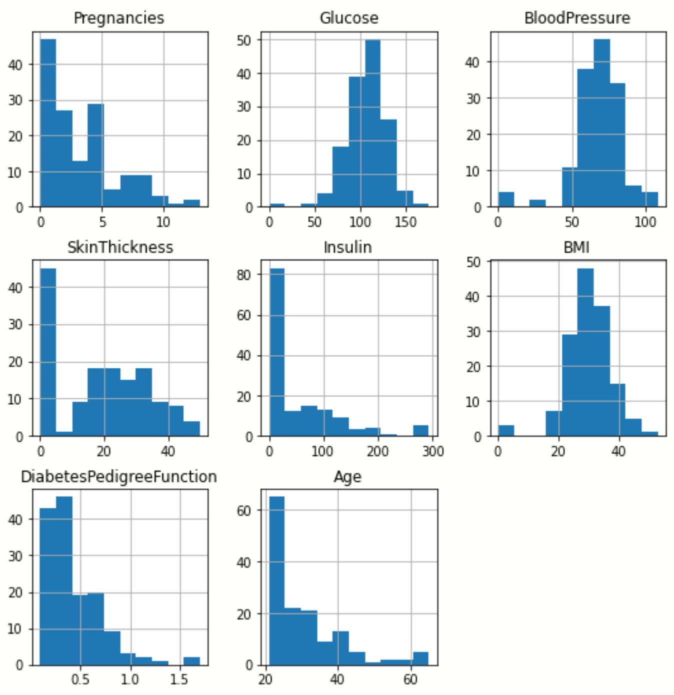
Data summary:



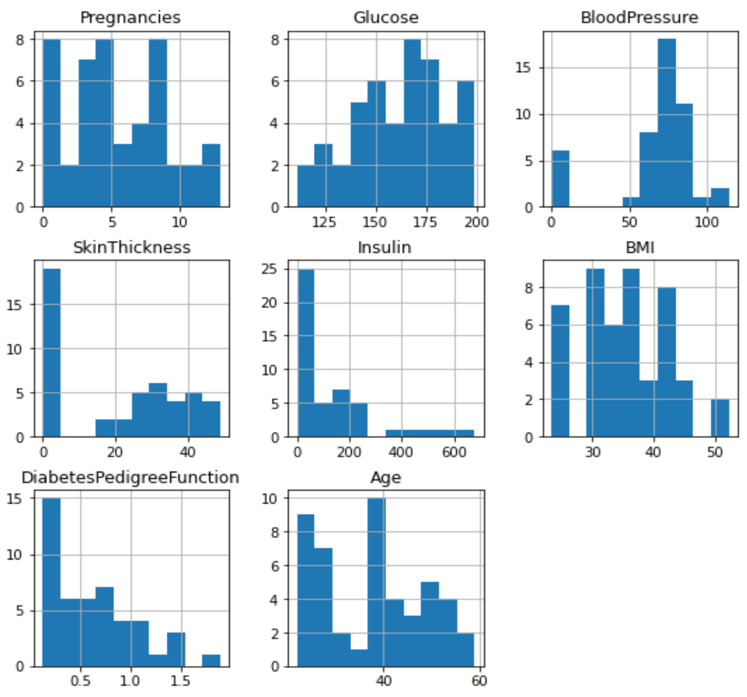
	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	192.000000	192.000000	192.000000	192.000000	192.000000	192.000000	192.000000	192.000000	192.000000
mean	3.859375	120.109375	67.369792	18.765625	66.645833	31.343750	0.481875	32.213542	0.244792
std	3.204384	32.843170	20.304842	15.628617	108.035526	7.634371	0.328778	10.964542	0.431088
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.100000	21.000000	0.000000
25%	1.000000	99.000000	60.000000	0.000000	0.000000	26.600000	0.254000	24.000000	0.000000
50%	3.000000	114.000000	70.000000	20.000000	0.000000	31.200000	0.372000	28.000000	0.000000
75%	6.000000	137.250000	78.000000	32.000000	105.000000	35.550000	0.660000	39.000000	0.000000
max	13.000000	199.000000	114.000000	50.000000	680.000000	52.900000	1.893000	65.000000	1.000000

Exploratory analysis:

Outcome:0(No diabetes)



outcome:1(Diabetes)



From the above figure, the patients with diabetes glucose level ranges between125 to 190 where as the patients without diabetes have a glucose level of 50 to 150.

Similarly the age for the patients with diabetes ranges between 30 to 60.

Model Building:

Logistic regression:

Logistic Regression is used when the dependent variable(target) is categorical.

The first step is the sigmoid function, A sigmoid function is an activation function. The output of the sigmoid function is always between a range of 0 to1.

Code for sigmoid function

```
def sigmoid(input):
    output = 1 / (1 + np.exp(-input))
    return output
```

After initialization of the parameters, optimization function is defined which optimizes the parameter values.

Then, the model is trained with the learning rate of 0.6 and the number of iterations as 500.

Code for model train:

```
def train(X, y, learning_rate,iterations):
    parameters_out = optimize(X, y, learning_rate, iterations ,init_parameters)
    return parameters_out
parameters_out = train(X, y, learning_rate = 0.6, iterations = 500)
parameters_out
```

making predictions:

The output values are predicted with the sigmoid values as 0.5.

```
output_values = np.dot(X, parameters_out["weight"]) + parameters_out["bias"]
predictions = sigmoid(output_values) >= 1/2
expected=y
print("RMS: %r " % np.sqrt(np.mean((predictions - expected) ** 2)))
```

The root mean squared value is calculated by taking the squareroot of the mean of predicted-expected.
The RMS value obtained is `0.728`