Practical 1:

Aim: Write a program to implement Simple Linear Regression.

**Practical: Simple Linear Regression**

**Importing the libraries**

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

**import** pandas **as** pd

**Importing the dataset**

dataset **=** pd.read\_csv('Salary\_Data.csv')

X **=** dataset.iloc[:, :**-**1].values

y **=** dataset.iloc[:, **-**1].values

**Splitting the dataset into the Training set and Test set**

**from** sklearn.model\_selection **import** train\_test\_split

X\_train, X\_test, y\_train, y\_test **=** train\_test\_split(X, y, test\_size **=** 1**/**3, random\_state **=** 0)

**Training the Simple Linear Regression model on the Training set**

**from** sklearn.linear\_model **import** LinearRegression

regressor **=** LinearRegression()

regressor.fit(X\_train, y\_train)

Out[4]:

LinearRegression(copy\_X=True, fit\_intercept=True, n\_jobs=None, normalize=False)

**Predicting the Test set results**

y\_pred **=** regressor.predict(X\_test)

**Visualising the Training set results**

plt.scatter(X\_train, y\_train, color **=** 'red')

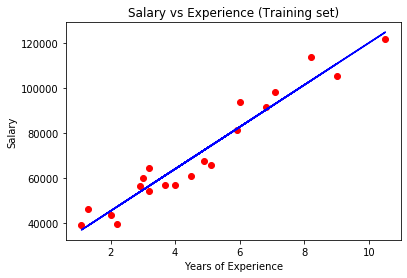
plt.plot(X\_train, regressor.predict(X\_train), color **=** 'blue')

plt.title('Salary vs Experience (Training set)')

plt.xlabel('Years of Experience')

plt.ylabel('Salary')

plt.show()



**Visualising the Test set results**

plt.scatter(X\_test, y\_test, color **=** 'red')

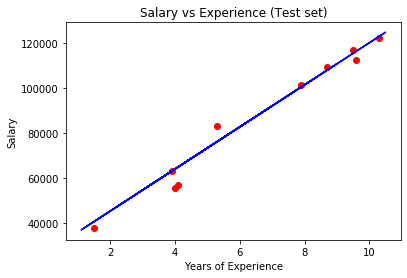
plt.plot(X\_train, regressor.predict(X\_train), color **=** 'blue')

plt.title('Salary vs Experience (Test set)')

plt.xlabel('Years of Experience')

plt.ylabel('Salary')

plt.show()



Practical 2:

Aim: Write a program to implement Multiple Linear Regression.

**Practical: Multiple Linear Regression**

**Importing the libraries**

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

**import** pandas **as** pd

**Importing the dataset**

dataset **=** pd.read\_csv('50\_Startups.csv')

X **=** dataset.iloc[:, :**-**1].values

y **=** dataset.iloc[:, **-**1].values

print(X)

[[165349.2 136897.8 471784.1 'New York']

[162597.7 151377.59 443898.53 'California']

[153441.51 101145.55 407934.54 'Florida']

.

.

.

[0.0 135426.92 0.0 'California']

[542.05 51743.15 0.0 'New York']

[0.0 116983.8 45173.06 'California']]

**Encoding categorical data**

**from** sklearn.compose **import** ColumnTransformer

**from** sklearn.preprocessing **import** OneHotEncoder

ct **=** ColumnTransformer(transformers**=**[('encoder', OneHotEncoder(), [3])], remainder**=**'passthrough')

X **=** np.array(ct.fit\_transform(X))

print(X)

[[0.0 0.0 1.0 165349.2 136897.8 471784.1]

[1.0 0.0 0.0 162597.7 151377.59 443898.53]

[0.0 1.0 0.0 153441.51 101145.55 407934.54]

.

.

.

[1.0 0.0 0.0 0.0 135426.92 0.0]

[0.0 0.0 1.0 542.05 51743.15 0.0]

[1.0 0.0 0.0 0.0 116983.8 45173.06]]

**Splitting the dataset into the Training set and Test set**

**from** sklearn.model\_selection **import** train\_test\_split

X\_train, X\_test, y\_train, y\_test **=** train\_test\_split(X, y, test\_size **=** 0.2, random\_state **=** 0)

**Training the Multiple Linear Regression model on the Training set**

**from** sklearn.linear\_model **import** LinearRegression

regressor **=** LinearRegression()

regressor.fit(X\_train, y\_train)

Out[7]:

LinearRegression(copy\_X=True, fit\_intercept=True, n\_jobs=None, normalize=False)

**Predicting the Test set results**

y\_pred **=** regressor.predict(X\_test)

np.set\_printoptions(precision**=**2)

print(np.concatenate((y\_pred.reshape(len(y\_pred),1), y\_test.reshape(len(y\_test),1)),1))

[[103015.2 103282.38]

[132582.28 144259.4 ]

[132447.74 146121.95]

[ 71976.1 77798.83]

[178537.48 191050.39]

[116161.24 105008.31]

[ 67851.69 81229.06]

[ 98791.73 97483.56]

[113969.44 110352.25]

[167921.07 166187.94]]

Practical 3:

Aim: Write a program to implement K-nearest Neighbors(KNN) or Support vector Machine (SVM).

**Practical: Support Vector Machine (SVM)**

**Importing the libraries**

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

**import** pandas **as** pd

**Importing the dataset**

dataset **=** pd.read\_csv('Social\_Network\_Ads.csv')

X **=** dataset.iloc[:, :**-**1].values

y **=** dataset.iloc[:, **-**1].values

**Splitting the dataset into the Training set and Test set**

**from** sklearn.model\_selection **import** train\_test\_split

X\_train, X\_test, y\_train, y\_test **=** train\_test\_split(X, y, test\_size **=** 0.25, random\_state **=** 0)

print(X\_train)

[[ 44 39000]

[ 32 120000]

[ 38 50000]

[ 32 135000]

[ 52 21000]

.

.

.

[ 48 30000]

[ 29 43000]

[ 36 52000]

[ 27 54000]

[ 26 118000]]

print(y\_train)

[0 1 0 1 1 1 0 0 0 0 0 0 1 1 1 0 1 0 0 1 0 1 0 1 0 0 1 1 1 1 0 1 0 1 0 0 1

0 0 1 0 0 0 0 0 1 1 1 1 0 0 0 1 0 1 0 1 0 0 1 0 0 0 1 0 0 0 1 1 0 0 1 0 1

1 1 0 0 1 1 0 0 1 1 0 1 0 0 1 1 0 1 1 1 0 0 0 0 0 1 0 0 1 1 1 1 1 0 1 1 0

1 0 0 0 0 0 0 0 1 1 0 0 1 0 0 1 0 0 0 1 0 1 1 0 1 0 0 0 0 1 0 0 0 1 1 0 0

0 0 1 0 1 0 0 0 1 0 0 0 0 1 1 1 0 0 0 0 0 0 1 1 1 1 1 0 1 0 0 0 0 0 1 0 0

0 0 0 0 1 1 0 1 0 1 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 1 1 0 0 0 0 0

0 1 1 0 0 0 0 1 0 0 0 0 1 0 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 0 0 0 1 1 0 0 0

0 0 1 0 1 1 0 0 0 0 0 1 0 1 0 0 1 0 0 1 0 1 0 0 0 0 0 0 1 1 1 1 0 0 0 0 1

0 0 0 0]

print(X\_test)

[[ 30 87000]

[ 38 50000]

[ 35 75000]

[ 30 79000]

[ 35 50000]

.

.

.

[ 27 96000]

[ 23 63000]

[ 48 33000]

[ 48 90000]

[ 42 104000]]

print(y\_test)

[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 0 1 0 0 0 0 0 1 1 0 0 0 0

0 0 1 0 0 0 0 1 0 0 1 0 1 1 0 0 0 1 1 0 0 1 0 0 1 0 1 0 1 0 0 0 0 1 0 0 1

0 0 0 0 1 1 1 0 0 0 1 1 0 1 1 0 0 1 0 0 0 1 0 1 1 1]

**Feature Scaling**

**from** sklearn.preprocessing **import** StandardScaler

sc **=** StandardScaler()

X\_train **=** sc.fit\_transform(X\_train)

X\_test **=** sc.transform(X\_test)

print(X\_train)

[[ 0.58164944 -0.88670699]

[-0.60673761 1.46173768]

[-0.01254409 -0.5677824 ]

[-0.60673761 1.89663484]

[ 1.37390747 -1.40858358]

.

.

.

[ 0.97777845 -1.14764529]

[-0.90383437 -0.77073441]

[-0.21060859 -0.50979612]

[-1.10189888 -0.45180983]

[-1.20093113 1.40375139]]

print(X\_test)

[[-0.80480212 0.50496393]

[-0.01254409 -0.5677824 ]

[-0.30964085 0.1570462 ]

[-0.80480212 0.27301877]

[-0.30964085 -0.5677824 ]

.

.

.

[-1.10189888 0.76590222]

[-1.49802789 -0.19087153]

[ 0.97777845 -1.06066585]

[ 0.97777845 0.59194336]

[ 0.38358493 0.99784738]]

**Training the SVM model on the Training set**

**from** sklearn.svm **import** SVC

classifier **=** SVC(kernel **=** 'linear', random\_state **=** 0)

classifier.fit(X\_train, y\_train)

Out[11]:

SVC(C=1.0, break\_ties=False, cache\_size=200, class\_weight=None, coef0=0.0,

decision\_function\_shape='ovr', degree=3, gamma='scale', kernel='linear',

max\_iter=-1, probability=False, random\_state=0, shrinking=True, tol=0.001,

verbose=False)

**Predicting a new result**

print(classifier.predict(sc.transform([[30,87000]])))

[0]

**Predicting the Test set results**

y\_pred **=** classifier.predict(X\_test)

print(np.concatenate((y\_pred.reshape(len(y\_pred),1), y\_test.reshape(len(y\_test),1)),1))

[[0 0]

[0 0]

[0 0]

[0 0]

[0 0]

.

.

.

[0 1]

[0 0]

[0 1]

[1 1]

[1 1]]

**Making the Confusion Matrix**

**from** sklearn.metrics **import** confusion\_matrix, accuracy\_score

cm **=** confusion\_matrix(y\_test, y\_pred)

print(cm)

accuracy\_score(y\_test, y\_pred)

[[66 2]

[ 8 24]]

Out[14]:

0.9

**Visualising the Training set results**

**from** matplotlib.colors **import** ListedColormap

X\_set, y\_set **=** sc.inverse\_transform(X\_train), y\_train

X1, X2 **=** np.meshgrid(np.arange(start **=** X\_set[:, 0].min() **-** 10, stop **=** X\_set[:, 0].max() **+** 10, step **=** 0.25),

np.arange(start **=** X\_set[:, 1].min() **-** 1000, stop **=** X\_set[:, 1].max() **+** 1000, step **=** 0.25))

plt.contourf(X1, X2, classifier.predict(sc.transform(np.array([X1.ravel(), X2.ravel()]).T)).reshape(X1.shape),

alpha **=** 0.75, cmap **=** ListedColormap(('red', 'green')))

plt.xlim(X1.min(), X1.max())

plt.ylim(X2.min(), X2.max())

**for** i, j **in** enumerate(np.unique(y\_set)):

plt.scatter(X\_set[y\_set **==** j, 0], X\_set[y\_set **==** j, 1], c **=** ListedColormap(('red', 'green'))(i), label **=** j)

plt.title('SVM (Training set)')

plt.xlabel('Age')

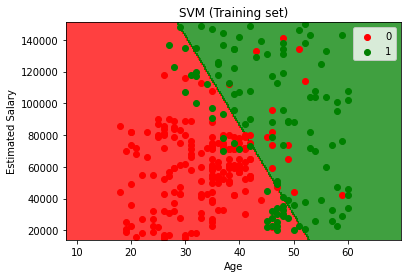
plt.ylabel('Estimated Salary')

plt.legend()

plt.show()

'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with 'x' & 'y'. Please use a 2-D array with a single row if you really want to specify the same RGB or RGBA value for all points.

'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with 'x' & 'y'. Please use a 2-D array with a single row if you really want to specify the same RGB or RGBA value for all points.



**Visualising the Test set results**

**from** matplotlib.colors **import** ListedColormap

X\_set, y\_set **=** sc.inverse\_transform(X\_test), y\_test

X1, X2 **=** np.meshgrid(np.arange(start **=** X\_set[:, 0].min() **-** 10, stop **=** X\_set[:, 0].max() **+** 10, step **=** 0.25),

np.arange(start **=** X\_set[:, 1].min() **-** 1000, stop **=** X\_set[:, 1].max() **+** 1000, step **=** 0.25))

plt.contourf(X1, X2, classifier.predict(sc.transform(np.array([X1.ravel(), X2.ravel()]).T)).reshape(X1.shape),

alpha **=** 0.75, cmap **=** ListedColormap(('red', 'green')))

plt.xlim(X1.min(), X1.max())

plt.ylim(X2.min(), X2.max())

**for** i, j **in** enumerate(np.unique(y\_set)):

plt.scatter(X\_set[y\_set **==** j, 0], X\_set[y\_set **==** j, 1], c **=** ListedColormap(('red', 'green'))(i), label **=** j)

plt.title('SVM (Test set)')

plt.xlabel('Age')

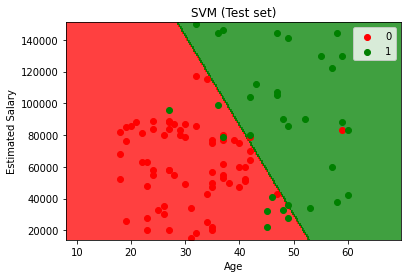
plt.ylabel('Estimated Salary')

plt.legend()

plt.show()

'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with 'x' & 'y'. Please use a 2-D array with a single row if you really want to specify the same RGB or RGBA value for all points.

'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with 'x' & 'y'. Please use a 2-D array with a single row if you really want to specify the same RGB or RGBA value for all points.



Practical 4:

Aim: Write a program to implement Naïve Bayse / DT.

**Practical: Naive Bayes**

**Importing the libraries**

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

**import** pandas **as** pd

**Importing the dataset**

dataset **=** pd.read\_csv('Social\_Network\_Ads.csv')

X **=** dataset.iloc[:, :**-**1].values

y **=** dataset.iloc[:, **-**1].values

**Splitting the dataset into the Training set and Test set**

**from** sklearn.model\_selection **import** train\_test\_split

X\_train, X\_test, y\_train, y\_test **=** train\_test\_split(X, y, test\_size **=** 0.25, random\_state **=** 0)

print(X\_train)

[[ 44 39000]

[ 32 120000]

[ 38 50000]

[ 32 135000]

[ 52 21000]

.

.

.

[ 48 30000]

[ 29 43000]

[ 36 52000]

[ 27 54000]

[ 26 118000]]

print(y\_train)

[0 1 0 1 1 1 0 0 0 0 0 0 1 1 1 0 1 0 0 1 0 1 0 1 0 0 1 1 1 1 0 1 0 1 0 0 1

0 0 1 0 0 0 0 0 1 1 1 1 0 0 0 1 0 1 0 1 0 0 1 0 0 0 1 0 0 0 1 1 0 0 1 0 1

1 1 0 0 1 1 0 0 1 1 0 1 0 0 1 1 0 1 1 1 0 0 0 0 0 1 0 0 1 1 1 1 1 0 1 1 0

1 0 0 0 0 0 0 0 1 1 0 0 1 0 0 1 0 0 0 1 0 1 1 0 1 0 0 0 0 1 0 0 0 1 1 0 0

0 0 1 0 1 0 0 0 1 0 0 0 0 1 1 1 0 0 0 0 0 0 1 1 1 1 1 0 1 0 0 0 0 0 1 0 0

0 0 0 0 1 1 0 1 0 1 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 1 1 0 0 0 0 0

0 1 1 0 0 0 0 1 0 0 0 0 1 0 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 0 0 0 1 1 0 0 0

0 0 1 0 1 1 0 0 0 0 0 1 0 1 0 0 1 0 0 1 0 1 0 0 0 0 0 0 1 1 1 1 0 0 0 0 1

0 0 0 0]

print(X\_test)

[[ 30 87000]

[ 38 50000]

[ 35 75000]

[ 30 79000]

[ 35 50000]

.

.

.

[ 27 96000]

[ 23 63000]

[ 48 33000]

[ 48 90000]

[ 42 104000]]

print(y\_test)

[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 0 1 0 0 0 0 0 1 1 0 0 0 0

0 0 1 0 0 0 0 1 0 0 1 0 1 1 0 0 0 1 1 0 0 1 0 0 1 0 1 0 1 0 0 0 0 1 0 0 1

0 0 0 0 1 1 1 0 0 0 1 1 0 1 1 0 0 1 0 0 0 1 0 1 1 1]

**Feature Scaling**

**from** sklearn.preprocessing **import** StandardScaler

sc **=** StandardScaler()

X\_train **=** sc.fit\_transform(X\_train)

X\_test **=** sc.transform(X\_test)

print(X\_train)

[[ 0.58164944 -0.88670699]

[-0.60673761 1.46173768]

[-0.01254409 -0.5677824 ]

[-0.60673761 1.89663484]

[ 1.37390747 -1.40858358]

[ 1.47293972 0.99784738]

.

.

.

[ 0.97777845 -1.14764529]

[-0.90383437 -0.77073441]

[-0.21060859 -0.50979612]

[-1.10189888 -0.45180983]

[-1.20093113 1.40375139]]

print(X\_test)

[[-0.80480212 0.50496393]

[-0.01254409 -0.5677824 ]

[-0.30964085 0.1570462 ]

[-0.80480212 0.27301877]

[-0.30964085 -0.5677824 ]

.

.

.

[-1.10189888 0.76590222]

[-1.49802789 -0.19087153]

[ 0.97777845 -1.06066585]

[ 0.97777845 0.59194336]

[ 0.38358493 0.99784738]]

**Training the Naive Bayes model on the Training set**

**from** sklearn.naive\_bayes **import** GaussianNB

classifier **=** GaussianNB()

classifier.fit(X\_train, y\_train)

Out[11]:

GaussianNB(priors=None, var\_smoothing=1e-09)

**Predicting a new result**

print(classifier.predict(sc.transform([[30,87000]])))

[0]

**Predicting the Test set results**

y\_pred **=** classifier.predict(X\_test)

print(np.concatenate((y\_pred.reshape(len(y\_pred),1), y\_test.reshape(len(y\_test),1)),1))

[[0 0]

[0 0]

[0 0]

[0 0]

[0 0]

.

.

.

[0 1]

[0 0]

[1 1]

[1 1]

[1 1]]

**Making the Confusion Matrix**

**from** sklearn.metrics **import** confusion\_matrix, accuracy\_score

cm **=** confusion\_matrix(y\_test, y\_pred)

print(cm)

accuracy\_score(y\_test, y\_pred)

[[65 3]

[ 7 25]]

Out[14]:

0.9

**Visualising the Training set results**

**from** matplotlib.colors **import** ListedColormap

X\_set, y\_set **=** sc.inverse\_transform(X\_train), y\_train

X1, X2 **=** np.meshgrid(np.arange(start **=** X\_set[:, 0].min() **-** 10, stop **=** X\_set[:, 0].max() **+** 10, step **=** 0.25),

np.arange(start **=** X\_set[:, 1].min() **-** 1000, stop **=** X\_set[:, 1].max() **+** 1000, step **=** 0.25))

plt.contourf(X1, X2, classifier.predict(sc.transform(np.array([X1.ravel(), X2.ravel()]).T)).reshape(X1.shape),

alpha **=** 0.75, cmap **=** ListedColormap(('red', 'green')))

plt.xlim(X1.min(), X1.max())

plt.ylim(X2.min(), X2.max())

**for** i, j **in** enumerate(np.unique(y\_set)):

plt.scatter(X\_set[y\_set **==** j, 0], X\_set[y\_set **==** j, 1], c **=** ListedColormap(('red', 'green'))(i), label **=** j)

plt.title('Naive Bayes (Training set)')

plt.xlabel('Age')

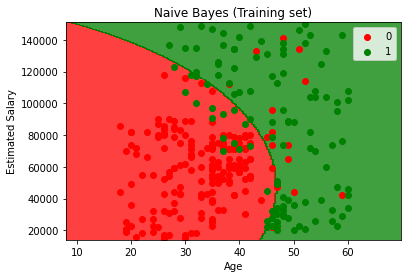
plt.ylabel('Estimated Salary')

plt.legend()

plt.show()

'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with 'x' & 'y'. Please use a 2-D array with a single row if you really want to specify the same RGB or RGBA value for all points.

'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with 'x' & 'y'. Please use a 2-D array with a single row if you really want to specify the same RGB or RGBA value for all points.



**Visualising the Test set results**

**from** matplotlib.colors **import** ListedColormap

X\_set, y\_set **=** sc.inverse\_transform(X\_test), y\_test

X1, X2 **=** np.meshgrid(np.arange(start **=** X\_set[:, 0].min() **-** 10, stop **=** X\_set[:, 0].max() **+** 10, step **=** 0.25),

np.arange(start **=** X\_set[:, 1].min() **-** 1000, stop **=** X\_set[:, 1].max() **+** 1000, step **=** 0.25))

plt.contourf(X1, X2, classifier.predict(sc.transform(np.array([X1.ravel(), X2.ravel()]).T)).reshape(X1.shape),

alpha **=** 0.75, cmap **=** ListedColormap(('red', 'green')))

plt.xlim(X1.min(), X1.max())

plt.ylim(X2.min(), X2.max())

**for** i, j **in** enumerate(np.unique(y\_set)):

plt.scatter(X\_set[y\_set **==** j, 0], X\_set[y\_set **==** j, 1], c **=** ListedColormap(('red', 'green'))(i), label **=** j)

plt.title('Naive Bayes (Test set)')

plt.xlabel('Age')

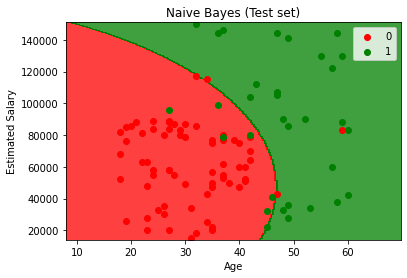
plt.ylabel('Estimated Salary')

plt.legend()

plt.show()

'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with 'x' & 'y'. Please use a 2-D array with a single row if you really want to specify the same RGB or RGBA value for all points.

'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with 'x' & 'y'. Please use a 2-D array with a single row if you really want to specify the same RGB or RGBA value for all points.



Practical 5:

Aim: Write a program to implement K-means clustering.

**Practical: K-Means Clustering**

**Importing the libraries**

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

**import** pandas **as** pd

**Importing the dataset**

dataset **=** pd.read\_csv('Mall\_Customers.csv')

X **=** dataset.iloc[:, [3, 4]].values

**Using the elbow method to find the optimal number of clusters**

**from** sklearn.cluster **import** KMeans

wcss **=** []

**for** i **in** range(1, 11):

kmeans **=** KMeans(n\_clusters **=** i, init **=** 'k-means++', random\_state **=** 42)

kmeans.fit(X)

wcss.append(kmeans.inertia\_)

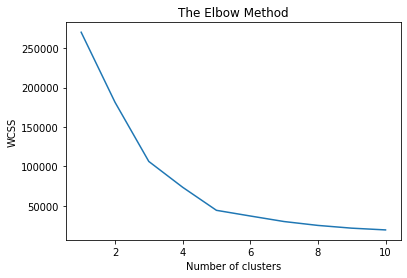
plt.plot(range(1, 11), wcss)

plt.title('The Elbow Method')

plt.xlabel('Number of clusters')

plt.ylabel('WCSS')

plt.show()



**Training the K-Means model on the dataset**

kmeans **=** KMeans(n\_clusters **=** 5, init **=** 'k-means++', random\_state **=** 42)

y\_kmeans **=** kmeans.fit\_predict(X)

**Visualising the clusters**

plt.scatter(X[y\_kmeans **==** 0, 0], X[y\_kmeans **==** 0, 1], s **=** 100, c **=** 'red', label **=** 'Cluster 1')

plt.scatter(X[y\_kmeans **==** 1, 0], X[y\_kmeans **==** 1, 1], s **=** 100, c **=** 'blue', label **=** 'Cluster 2')

plt.scatter(X[y\_kmeans **==** 2, 0], X[y\_kmeans **==** 2, 1], s **=** 100, c **=** 'green', label **=** 'Cluster 3')

plt.scatter(X[y\_kmeans **==** 3, 0], X[y\_kmeans **==** 3, 1], s **=** 100, c **=** 'cyan', label **=** 'Cluster 4')

plt.scatter(X[y\_kmeans **==** 4, 0], X[y\_kmeans **==** 4, 1], s **=** 100, c **=** 'magenta', label **=** 'Cluster 5')

plt.scatter(kmeans.cluster\_centers\_[:, 0], kmeans.cluster\_centers\_[:, 1], s **=** 300, c **=** 'yellow', label **=** 'Centroids')

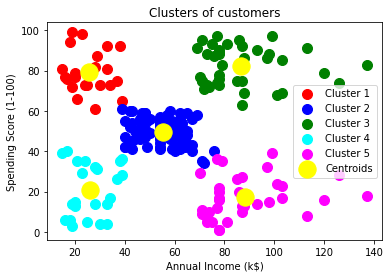
plt.title('Clusters of customers')

plt.xlabel('Annual Income (k$)')

plt.ylabel('Spending Score (1-100)')

plt.legend()

plt.show()



Practical 6:

Aim: Write a program to implement Hierarchical clustering.

**Practical: Hierarchical Clustering**

**Importing the libraries**

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

**import** pandas **as** pd

**Importing the dataset**

dataset **=** pd.read\_csv('Mall\_Customers.csv')

X **=** dataset.iloc[:, [3, 4]].values

**Using the dendrogram to find the optimal number of clusters**

**import** scipy.cluster.hierarchy **as** sch

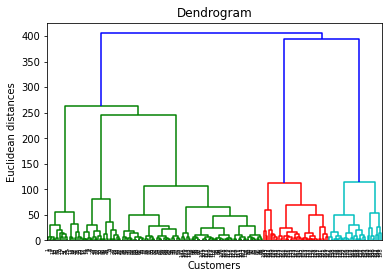
dendrogram **=** sch.dendrogram(sch.linkage(X, method **=** 'ward'))

plt.title('Dendrogram')

plt.xlabel('Customers')

plt.ylabel('Euclidean distances')

plt.show()



**Training the Hierarchical Clustering model on the dataset**

**from** sklearn.cluster **import** AgglomerativeClustering

hc **=** AgglomerativeClustering(n\_clusters **=** 5, affinity **=** 'euclidean', linkage **=** 'ward')

y\_hc **=** hc.fit\_predict(X)

**Visualising the clusters**

plt.scatter(X[y\_hc **==** 0, 0], X[y\_hc **==** 0, 1], s **=** 100, c **=** 'red', label **=** 'Cluster 1')

plt.scatter(X[y\_hc **==** 1, 0], X[y\_hc **==** 1, 1], s **=** 100, c **=** 'blue', label **=** 'Cluster 2')

plt.scatter(X[y\_hc **==** 2, 0], X[y\_hc **==** 2, 1], s **=** 100, c **=** 'green', label **=** 'Cluster 3')

plt.scatter(X[y\_hc **==** 3, 0], X[y\_hc **==** 3, 1], s **=** 100, c **=** 'cyan', label **=** 'Cluster 4')

plt.scatter(X[y\_hc **==** 4, 0], X[y\_hc **==** 4, 1], s **=** 100, c **=** 'magenta', label **=** 'Cluster 5')

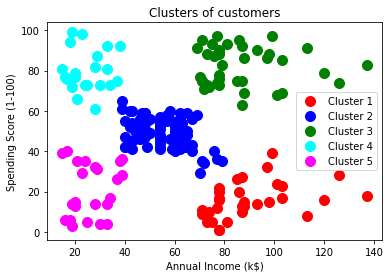
plt.title('Clusters of customers')

plt.xlabel('Annual Income (k$)')

plt.ylabel('Spending Score (1-100)')

plt.legend()

plt.show()



Practical 7:

Aim: Write a program to build ANN (Artificial Neural Network)

Code:

# ****Practical: Artificial Neural Network****

### **Importing the libraries**

[2]

3s

import numpy as np  
import pandas as pd  
import tensorflow as tf

[3]

0s

tf.\_\_version\_\_

## ****Part 1 - Data Preprocessing****

### **Importing the dataset**

[4]

0s

dataset = pd.read\_csv('Churn\_Modelling.csv')  
X = dataset.iloc[:, 3:-1].values  
y = dataset.iloc[:, -1].values

[5]

0s

print(X)

[[619 'France' 'Female' ... 1 1 101348.88]

[608 'Spain' 'Female' ... 0 1 112542.58]

[502 'France' 'Female' ... 1 0 113931.57]

...

[709 'France' 'Female' ... 0 1 42085.58]

[772 'Germany' 'Male' ... 1 0 92888.52]

[792 'France' 'Female' ... 1 0 38190.78]]

[6]

0s

print(y)

[1 0 1 ... 1 1 0]

### **Encoding categorical data**

Label Encoding the "Gender" column

[7]

0s

from sklearn.preprocessing import LabelEncoder  
le = LabelEncoder()  
X[:, 2] = le.fit\_transform(X[:, 2])  
print(X)

[[619 'France' 0 ... 1 1 101348.88]

[608 'Spain' 0 ... 0 1 112542.58]

[502 'France' 0 ... 1 0 113931.57]

...

[709 'France' 0 ... 0 1 42085.58]

[772 'Germany' 1 ... 1 0 92888.52]

[792 'France' 0 ... 1 0 38190.78]]

One Hot Encoding the "Geography" column

[8]

0s

from sklearn.compose import ColumnTransformer  
from sklearn.preprocessing import OneHotEncoder  
ct = ColumnTransformer(transformers=[('encoder', OneHotEncoder(), [1])], remainder='passthrough')  
X = np.array(ct.fit\_transform(X))  
print(X)

[[1.0 0.0 0.0 ... 1 1 101348.88]

[0.0 0.0 1.0 ... 0 1 112542.58]

[1.0 0.0 0.0 ... 1 0 113931.57]

...

[1.0 0.0 0.0 ... 0 1 42085.58]

[0.0 1.0 0.0 ... 1 0 92888.52]

[1.0 0.0 0.0 ... 1 0 38190.78]]

### **Splitting the dataset into the Training set and Test set**

[10]

0s

from sklearn.model\_selection import train\_test\_split  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.2, random\_state = 0)

### **Fearture scaling**

[11]

0s

from sklearn.preprocessing import StandardScaler  
sc = StandardScaler()  
X\_train = sc.fit\_transform(X\_train)  
X\_test = sc.transform(X\_test)

## ****Part 2 - Building the ANN****

### **Initializing the ANN**

[12]

0s

ann = tf.keras.models.Sequential()

### **Adding the input layer and the first hidden layer**

[13]

0s

ann.add(tf.keras.layers.Dense(units=6, activation='relu'))

### **Adding the second hidden layer**

[14]

0s

ann.add(tf.keras.layers.Dense(units=6, activation='relu'))

### **Adding the output layer**

[15]

0s

ann.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))

## ****Part 3 - Training the ANN****

### **Compiling the ANN**

[16]

0s

ann.compile(optimizer = 'adam', loss = 'binary\_crossentropy', metrics = ['accuracy'])

### **Training the ANN on the Training set**

[17]

40s

ann.fit(X\_train, y\_train, batch\_size = 32, epochs = 100)

Epoch 1/100

250/250 [==============================] - 1s 2ms/step - loss: 0.5278 - accuracy: 0.7920

Epoch 2/100

250/250 [==============================] - 0s 2ms/step - loss: 0.4637 - accuracy: 0.7983

Epoch 3/100

250/250 [==============================] - 0s 2ms/step - loss: 0.4439 - accuracy: 0.8050

Epoch 4/100

250/250 [==============================] - 0s 2ms/step - loss: 0.4350 - accuracy: 0.8098

Epoch 5/100

250/250 [==============================] - 0s 2ms/step - loss: 0.4291 - accuracy: 0.8124

Epoch 6/100

250/250 [==============================] - 0s 2ms/step - loss: 0.4249 - accuracy: 0.8156

Epoch 7/100

250/250 [==============================] - 0s 2ms/step - loss: 0.4210 - accuracy: 0.8191

Epoch 8/100

250/250 [==============================] - 0s 2ms/step - loss: 0.4172 - accuracy: 0.8229

Epoch 9/100

250/250 [==============================] - 0s 2ms/step - loss: 0.4139 - accuracy: 0.8264

Epoch 10/100

250/250 [==============================] - 0s 2ms/step - loss: 0.4114 - accuracy: 0.8292

Epoch 11/100

250/250 [==============================] - 0s 2ms/step - loss: 0.4093 - accuracy: 0.8305

Epoch 12/100

250/250 [==============================] - 0s 2ms/step - loss: 0.4071 - accuracy: 0.8298

Epoch 13/100

250/250 [==============================] - 0s 2ms/step - loss: 0.4056 - accuracy: 0.8326

Epoch 14/100

250/250 [==============================] - 0s 2ms/step - loss: 0.4037 - accuracy: 0.8317

Epoch 15/100

250/250 [==============================] - 0s 2ms/step - loss: 0.4023 - accuracy: 0.8339

Epoch 16/100

250/250 [==============================] - 0s 2ms/step - loss: 0.4005 - accuracy: 0.8338

Epoch 17/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3987 - accuracy: 0.8344

Epoch 18/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3966 - accuracy: 0.8355

Epoch 19/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3944 - accuracy: 0.8345

Epoch 20/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3920 - accuracy: 0.8353

Epoch 21/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3900 - accuracy: 0.8355

Epoch 22/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3873 - accuracy: 0.8367

Epoch 23/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3848 - accuracy: 0.8369

Epoch 24/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3819 - accuracy: 0.8364

Epoch 25/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3792 - accuracy: 0.8379

Epoch 26/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3764 - accuracy: 0.8407

Epoch 27/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3738 - accuracy: 0.8432

Epoch 28/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3712 - accuracy: 0.8454

Epoch 29/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3690 - accuracy: 0.8464

Epoch 30/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3667 - accuracy: 0.8475

Epoch 31/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3640 - accuracy: 0.8478

Epoch 32/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3627 - accuracy: 0.8493

Epoch 33/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3610 - accuracy: 0.8511

Epoch 34/100

250/250 [==============================] - 0s 1ms/step - loss: 0.3594 - accuracy: 0.8504

Epoch 35/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3580 - accuracy: 0.8518

Epoch 36/100

250/250 [==============================] - 0s 1ms/step - loss: 0.3570 - accuracy: 0.8511

Epoch 37/100

250/250 [==============================] - 0s 1ms/step - loss: 0.3557 - accuracy: 0.8536

Epoch 38/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3544 - accuracy: 0.8534

Epoch 39/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3534 - accuracy: 0.8550

Epoch 40/100

250/250 [==============================] - 0s 1ms/step - loss: 0.3522 - accuracy: 0.8570

Epoch 41/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3514 - accuracy: 0.8572

Epoch 42/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3504 - accuracy: 0.8560

Epoch 43/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3494 - accuracy: 0.8576

Epoch 44/100

250/250 [==============================] - 0s 1ms/step - loss: 0.3483 - accuracy: 0.8594

Epoch 45/100

250/250 [==============================] - 0s 1ms/step - loss: 0.3475 - accuracy: 0.8576

Epoch 46/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3461 - accuracy: 0.8596

Epoch 47/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3458 - accuracy: 0.8601

Epoch 48/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3446 - accuracy: 0.8593

Epoch 49/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3441 - accuracy: 0.8600

Epoch 50/100

250/250 [==============================] - 0s 1ms/step - loss: 0.3439 - accuracy: 0.8599

Epoch 51/100

250/250 [==============================] - 0s 1ms/step - loss: 0.3428 - accuracy: 0.8609

Epoch 52/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3418 - accuracy: 0.8610

Epoch 53/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3411 - accuracy: 0.8627

Epoch 54/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3409 - accuracy: 0.8602

Epoch 55/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3406 - accuracy: 0.8610

Epoch 56/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3403 - accuracy: 0.8648

Epoch 57/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3386 - accuracy: 0.8605

Epoch 58/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3393 - accuracy: 0.8624

Epoch 59/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3389 - accuracy: 0.8630

Epoch 60/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3385 - accuracy: 0.8622

Epoch 61/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3385 - accuracy: 0.8631

Epoch 62/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3376 - accuracy: 0.8629

Epoch 63/100

250/250 [==============================] - 0s 1ms/step - loss: 0.3379 - accuracy: 0.8636

Epoch 64/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3374 - accuracy: 0.8610

Epoch 65/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3373 - accuracy: 0.8630

Epoch 66/100

250/250 [==============================] - 0s 1ms/step - loss: 0.3367 - accuracy: 0.8630

Epoch 67/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3370 - accuracy: 0.8615

Epoch 68/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3367 - accuracy: 0.8640

Epoch 69/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3366 - accuracy: 0.8639

Epoch 70/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3362 - accuracy: 0.8614

Epoch 71/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3362 - accuracy: 0.8643

Epoch 72/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3362 - accuracy: 0.8630

Epoch 73/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3356 - accuracy: 0.8622

Epoch 74/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3355 - accuracy: 0.8644

Epoch 75/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3358 - accuracy: 0.8611

Epoch 76/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3355 - accuracy: 0.8637

Epoch 77/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3352 - accuracy: 0.8624

Epoch 78/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3359 - accuracy: 0.8618

Epoch 79/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3352 - accuracy: 0.8621

Epoch 80/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3360 - accuracy: 0.8624

Epoch 81/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3354 - accuracy: 0.8633

Epoch 82/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3350 - accuracy: 0.8640

Epoch 83/100

250/250 [==============================] - 0s 1ms/step - loss: 0.3358 - accuracy: 0.8616

Epoch 84/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3349 - accuracy: 0.8650

Epoch 85/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3345 - accuracy: 0.8645

Epoch 86/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3349 - accuracy: 0.8615

Epoch 87/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3347 - accuracy: 0.8635

Epoch 88/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3345 - accuracy: 0.8624

Epoch 89/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3341 - accuracy: 0.8649

Epoch 90/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3340 - accuracy: 0.8656

Epoch 91/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3345 - accuracy: 0.8631

Epoch 92/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3348 - accuracy: 0.8656

Epoch 93/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3338 - accuracy: 0.8648

Epoch 94/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3340 - accuracy: 0.8634

Epoch 95/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3337 - accuracy: 0.8633

Epoch 96/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3339 - accuracy: 0.8643

Epoch 97/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3339 - accuracy: 0.8640

Epoch 98/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3343 - accuracy: 0.8635

Epoch 99/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3336 - accuracy: 0.8630

Epoch 100/100

250/250 [==============================] - 0s 2ms/step - loss: 0.3339 - accuracy: 0.8636

<keras.callbacks.History at 0x7fac3f090890>

## ****Part 4 - Making the predictions and evaluating the model****

### **Predicting the result of a single observation**

**Homework**

Use our ANN model to predict if the customer with the following informations will leave the bank:

Geography: France

Credit Score: 600

Gender: Male

Age: 40 years old

Tenure: 3 years

Balance: $ 60000

Number of Products: 2

Does this customer have a credit card ? Yes

Is this customer an Active Member: Yes

Estimated Salary: $ 50000

So, should we say goodbye to that customer ?

**Solution**

[18]

0s

print(ann.predict(sc.transform([[1, 0, 0, 600, 1, 40, 3, 60000, 2, 1, 1, 50000]])) > 0.5)

[[False]]

Therefore, our ANN model predicts that this customer stays in the bank!

Important note 1: Notice that the values of the features were all input in a double pair of square brackets. That's because the "predict" method always expects a 2D array as the format of its inputs. And putting our values into a double pair of square brackets makes the input exactly a 2D array.

Important note 2: Notice also that the "France" country was not input as a string in the last column but as "1, 0, 0" in the first three columns. That's because of course the predict method expects the one-hot-encoded values of the state, and as we see in the first row of the matrix of features X, "France" was encoded as "1, 0, 0". And be careful to include these values in the first three columns, because the dummy variables are always created in the first columns.

### **Predicting the Test set results**

[19]

0s

y\_pred = ann.predict(X\_test)  
y\_pred = (y\_pred > 0.5)  
print(np.concatenate((y\_pred.reshape(len(y\_pred),1), y\_test.reshape(len(y\_test),1)),1))

[[0 0]

[0 1]

[0 0]

...

[0 0]

[0 0]

[0 0]]

### **Making the Confusion Matrix**

[20]

0s

from sklearn.metrics import confusion\_matrix, accuracy\_score

cm = confusion\_matrix(y\_test, y\_pred)

print(cm)

accuracy\_score(y\_test, y\_pred)





[[1499 96]

[ 189 216]]

0.8575