|  |  |
| --- | --- |
| **Ex. No: 01** | **LINEAR REGRESSION** |
| **27/12/2023** |

**Aim:**

1. To simulate 1000 different lines of fit for a randomly generated 2-D NumPy array, find the line with the least error among them, and then predict the target variable for the remaining 500 values using this line of best fit.
2. To build a simple linear regression model from scratch using data from data1.csv without using sklearn libraries.

**Algorithm:**

1.

1. Generate Data:
2. Create a random 2-D NumPy array with 1500 values.
3. Simulation of Lines of Fit:

Randomly select 1000 values from the array to simulate lines of fit.

1. For each selection, compute the parameters of the line of fit (e.g., slope and intercept).
2. Calculate the errors for each line of fit using the selected 1000 values.
3. Find Line of Best Fit:

Identify the line with the least error among all simulated lines.

1. Prediction:

Using the line of best fit, predict the target variable for the remaining 500 values.

1. Output:

Display the line of best fit along with the predicted values

2.

1. Data Loading:
2. Load the data from data1.csv.
3. Data Preprocessing:

If necessary, handle missing values and ensure the data is in a suitable format for regression.

1. Split Data:

Split the data into features (X) and target variable (y).

1. Equation Method:

Calculate the coefficients using the equations for simple linear regression.

1. Prediction:

Use the obtained coefficients to make predictions on the test data.

1. Evaluation:

Calculate RMSE and MAE using the predictions and actual target values.

1. Comparison:

Compare the intercept and coefficient values obtained from the equation method and gradient descent method.

1. Print the RMSE, MAE, and coefficient values for comparison.

**Program :**

1.

import numpy as np

import matplotlib.pyplot as plt

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error

random\_array = np.random.rand(1500,1)

# Split the 2D array into independent variable(X) and dependent variable (y)

X = random\_array[:1000]

y = 3 \* X.squeeze() + 2 + 0.1 \* np.random.randn(1000)  # Simulate a line with some noise,is a 1d array

#variables to store the best fit line and its error

best\_fit\_line = None

min\_error = float('inf')

#different lines of fit using 1000 values

for i in range(1000):

    # Randomly select 1000 indices for fitting the line

    indices = np.random.choice(1000, size=1000, replace=True)

    # Fit a line for selected indices

    X\_train = X[indices]

    y\_train = y[indices]

    model = LinearRegression()

    model.fit(X\_train, y\_train)#to train the model

    # Predict on the entire dataset,predicts y based on x

    y\_pred = model.predict(X)

    # Calculate mean squared error

    error = mean\_squared\_error(y, y\_pred)

    # Update the best fit line if the current error is smaller

    if error < min\_error:

        min\_error = error

        best\_fit\_line = model

# Using the best fit line, predict the target variable for the other 500 values

X\_test = random\_array[1000:]

y\_pred\_test = best\_fit\_line.predict(X\_test)

# Print the coefficients of the best fit line

print("Intercept:", best\_fit\_line.intercept\_)

print("Mean Squared Error of Best Fit Line:", min\_error)

# Plot the data points and the best fit line

plt.scatter(X, y, label='Data Points',c='yellow')#plot data points

plt.plot(X\_test,y\_pred\_test, color='magenta', label='Best Fit Line')#plot best line of fit

plt.xlabel('X')

plt.ylabel('y')

plt.title('Scatter Plot with Best Fit Line')

plt.legend()

plt.show()

**2.**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

data = pd.read\_csv("data1.csv")

X = data.iloc[:,:-1].values

y = data.iloc[:,-1].values

def mean\_absolute\_error(y\_true, y\_pred):

    return np.mean(np.abs(y\_true - y\_pred))

def root\_mean\_squared\_error(y\_true, y\_pred):

    return np.sqrt(np.mean((y\_true - y\_pred) \*\* 2))

def simple\_linear\_regression(X, y):

    n = len(X)

    mean\_X = np.mean(X)  #x'

    mean\_y = np.mean(y)   #y'

    numerator = np.sum((X - mean\_X) \* (y - mean\_y))

    denominator = np.sum((X - mean\_X) \*\* 2)

     #c=y'-mx'

    m = numerator / denominator

    b = mean\_y - m \* mean\_X

    return m, b #m-slope,b-intercept

#simple linear regression

slope, intercept = simple\_linear\_regression(X, y)

# Print the coefficients (slope and intercept)

print("Coefficient (slope):", slope)

print("Intercept:", intercept)

# Predict the values using the model

y\_pred = slope \* X + intercept

rmse = root\_mean\_squared\_error(y, y\_pred)

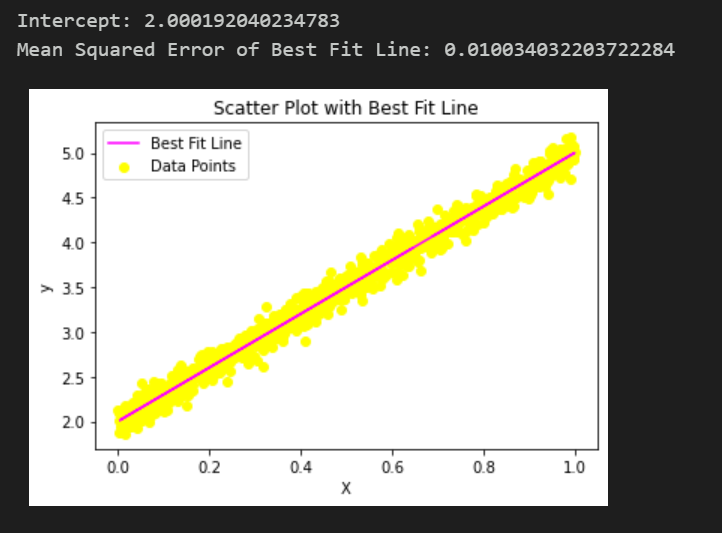
mae = mean\_absolute\_error(y, y\_pred)

print("Root Mean Squared Error (RMSE):", rmse)

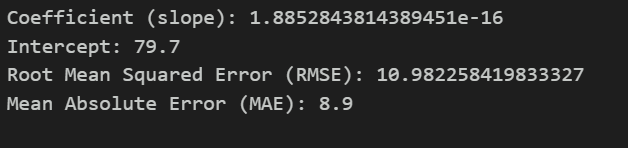
print("Mean Absolute Error (MAE):", mae)

**Output:**

**1.**

****

**2.**



**Result:** The above program to execute linear regression from scratch and using in built library has been successfully executed.

|  |  |
| --- | --- |
| **Ex. No: 02** | **MULTIPLE LINEAR REGRESSION** |
| **03/01/2024** |

**Aim:** To build a multiple linear regression model using the provided dataset to predict house prices.

**Algorithm:**

1. Preprocess the data, including handling missing values, encoding categorical variables, and scaling numerical features.
2. Check for outliers and handle them appropriately.
3. Check for multicollinearity among the independent variables.
4. Train the multiple linear regression model.
5. Apply the same preprocessing steps to the test dataset (test.csv) for consistency.
6. Use the trained model to predict house prices on the test dataset.
7. Evaluate the model's performance using Root Mean Squared Error (RMSE).

**Program :**

# Import necessary libraries

import pandas as pd

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

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error

# Load the dataset

df = pd.read\_csv('house\_pred.csv')

df.head()

X =  df.iloc[:,:-1].values

y = df.iloc[:,-1].values

df['intercept']=1

vif = pd.DataFrame()

vif['variable']= df.columns

from statsmodels.stats.outliers\_influence import \

variance\_inflation\_factor

vif['vif'] = [variance\_inflation\_factor(df.values,i)

 for i in range(df.shape[1])]

vif

df = df.drop(['BsmtFinSF2','GrLivArea'], axis=1)

df= df.values

df = pd.DataFrame(df)

vif = pd.DataFrame()

vif['variable']= df.columns

from statsmodels.stats.outliers\_influence import \

variance\_inflation\_factor

vif['vif'] = [variance\_inflation\_factor(df.values,i)

 for i in range(df.shape[1])]

vif

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2)

# Build the linear regression model

model = LinearRegression()

# Fit the model to the training data

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

# Make predictions on the test set

y\_pred = model.predict(X\_test)

# Evaluate the model using Root Mean Squared Error (RMSE)

rmse = np.sqrt(mean\_squared\_error(y\_test, y\_pred))

print(f'Root Mean Squared Error (RMSE): {rmse}')

from sklearn.metrics import r2\_score, mean\_squared\_error

r2\_score(y\_test,y\_pred)

# Now, load the test dataset for making predictions

test\_data = pd.read\_csv('test.csv')

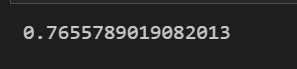
# Use the trained model to predict house prices on the test set

test\_predictions = model.predict(test\_data)

submission\_df = pd.DataFrame({'Id': test\_data['Id'], 'PredictedPrice': test\_predictions})

submission\_df.to\_csv('predictions4.csv', index=False)

**Output:**

****

**Result:** The above program to build a multiple linear regression model using the provided dataset to predict house prices has been successfully executed.

|  |  |
| --- | --- |
| **Ex. No: 03** | **LOGISTIC REGRESSION** |
| **17/01/2024** |

**Aim:** To build a logistic regression model using the provided dataset, perform necessary preprocessing, and evaluate its performance using a 70-30 train-test split.

**Algorithm:**

1. Load the dataset into a DataFrame.
2. Perform exploratory data analysis (EDA) to understand the data's structure, distributions, and relationships.
3. Preprocess the data
4. Split the dataset into features (X) and the target variable (y).
5. Split the data into training and testing sets using a 70-30 ratio.
6. Build a logistic regression model using the training data.
7. Train the logistic regression model on the training data.
8. Make predictions on the test data using the trained model.
9. Evaluate the performance of the model using various metrics:
10. Print the performance metrics to assess the model's effectiveness in classifying instances.

**Program :**

import pandas as pd

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

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score,precision\_score, recall\_score, f1\_score

df = pd.read\_csv('telecom\_customer\_churn.csv')

df.head()

#one hot encoding

import pandas as pd

col= ['Gender','Married','Phone Service','Multiple Lines','Internet Service','Online Security','Online Backup','Premium Tech Support','Streaming TV','Unlimited Data',]

data = pd.get\_dummies(df, columns=col, drop\_first=True)

data.head()

#segregate independent and dependent variable

x = data.iloc[:,:-1].values

y = data.iloc[:,-1].values

#build the model

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x, y, test\_size=0.3)

reg= LogisticRegression()

reg.fit(x\_train, y\_train)

y\_pred =reg.predict(x\_test)

# performance metrics

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

print("Accuracy" ,accuracy)

precision = precision\_score(y\_test, y\_pred)

recall = recall\_score(y\_test, y\_pred)

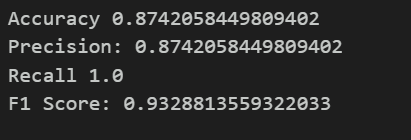
print("Precision:",precision)

print("Recall",recall)

f1 = f1\_score(y\_test, y\_pred)

print("F1 Score:",f1)

**Output:**

****

**Result:** The above program to build a logistic regression model using the provided dataset, perform necessary preprocessing, and evaluate its performance using a 70-30 train-test split has been successfully executed.

|  |  |
| --- | --- |
| **Ex. No: 04** | **DECISION TREE** |
| **24/01/2024** |

**Aim:**

1. To compute the Gini index for the 'age' and 'salary' columns in the classification.csv file.
2. To implement a decision tree algorithm from scratch without using the sklearn library.

**Algorithm:**

**1.**

1. Data Loading.
2. Data Preprocessing.
3. If necessary, handle missing values and ensure that 'age' and 'salary' columns are numerical.
4. Compute Gini Index:

For each column (age and salary):

Sort the unique values in ascending order.

For each unique value, split the dataset into two groups based on whether the value is less than or equal to the current value.

Calculate the Gini index for each split.

Weighted sum the Gini indices for all splits to compute the Gini index for the column.

1. Comparison:

Compare the computed Gini indices for the 'age' and 'salary' columns.

**2.**

1. Determine the best feature to split on based on a chosen criterion (e.g., Gini impurity, entropy).
2. Calculate impurity of each feature and select the one that reduces impurity the most.
3. Define conditions to stop further splitting.
4. Criteria may include maximum depth limit, minimum samples in a node, or no further reduction in impurity.
5. Recursively build the tree:
6. Repeat splitting and stopping criteria for each child node created from the split.
7. Use the built tree to make predictions:
8. Traverse down the tree based on feature values of the input sample until a leaf node is reached.
9. Assign the majority class in the leaf node as the predicted class.

**Program :**

1.

import numpy as np

import pandas as pd

df = pd.read\_csv("classification.csv")

df.head()

def GI(df,col,target,diff):

    tot = df.shape[0]

    lower = df[df[col] <= diff].shape[0]

    greater = df[df[col] > diff].shape[0]

    yes\_and\_greater = df[(df[target] == 1) & (df[col] > diff)].shape[0]

    no\_and\_greater = df[(df[target] == 0) & (df[col] > diff)].shape[0]

    yes\_and\_lower = df[(df[target] == 1) & (df[col] <= diff)].shape[0]

    no\_and\_lower = df[(df[target] == 0) & (df[col] <= diff)].shape[0]

    gini\_greater = 1 - (yes\_and\_greater/greater)\*\*2 - (no\_and\_greater/greater)\*\*2

    gini\_lower = 1 - (yes\_and\_lower/lower)\*\*2 - (no\_and\_lower/lower)\*\*2

    ans = gini\_greater\*(greater/tot) + gini\_lower\*(lower/tot)

    return ans

l = list(df["Age"].unique())

l.sort()

for i in range(len(l)-1):

    diff = (l[i]+l[i+1]) / 2

    print(diff," : ",GI(df,"Age","Purchased",diff))

l = list(df["EstimatedSalary"].unique())

l.sort()

for i in range(len(l)-1):

    diff = (l[i]+l[i+1]) / 2

    print(diff," : ",GI(df,"EstimatedSalary","Purchased",diff))

2.

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

train = {"Color" : ["Green","Yellow","Red","Red","Yellow"], "Diameter" : [3,3,1,1,3], "Label" : ["Apple","Apple","Grape","Grape","Lemon"]}

train = pd.DataFrame(train)

def unique\_value(df,col):

    return list(df[col].unique())

def class\_counts(df):

    ans = {}

    for i in df.iloc[:,-1]:

        if i not in ans:

            ans[i] = 0

        ans[i] += 1

    return ans

def is\_numeric(val):

    if(type(val) == int or  type(val) == float):

        return True

    else:

        return False

def is\_numeric(val):

    if(type(val) == int or  type(val) == float):

        return True

    else:

        return False

class Question:

    def \_\_init\_\_(self,col,value):

        self.column = col

        self.value = value

    def match(self,example):

        val = example[self.column]

        if is\_numeric(val):

            return val >= self.value

        else:

            return val == self.value

    def \_\_repr\_\_(self):

        condition = "=="

        if is\_numeric(self.value):

            condition = ">="

        return "Is %s %s %s?" % (self.column,condition,str(self.value))

q = Question("Color","Green")

q.match(train)

def partition(df,question):

    true = pd.DataFrame()

    false = pd.DataFrame()

    true = df[question.match(df) == True]

    false = df[question.match(df) == False]

    return true,false

true\_rows,false\_rows = partition(train,Question("Color","Red"))

def gini(l):

    counts = class\_counts(l)

    impurity = 1

    for i in counts:

        prob = counts[i]/float(len(l))

        impurity -= prob\*\*2

    return impurity

l = pd.DataFrame({"Res" : ["Apple","Apple"]})

print(gini(l))

l = pd.DataFrame({"Res" : ["Orange","Apple"]})

print(gini(l))

l = pd.DataFrame({"Res" : ["Apple","Apple","Grape","Fruit"]})

print(gini(l))

def info\_gain(left,right,current\_uncertainty):

    left\_len = left.shape[0]

    right\_len = right.shape[0]

    p = float(left\_len) / (left\_len + right\_len)

    return current\_uncertainty - p\*gini(left) - (1-p)\*gini(right)

current\_uncertainty = gini(train)

current\_uncertainty

# how much info do we gain by partioning on "Green"

true\_rows, false\_rows = partition(train, Question("Color", 'Green'))

info\_gain(true\_rows, false\_rows, current\_uncertainty)

# how much info do we gain by partioning on "Red"

true\_rows, false\_rows = partition(train, Question("Color",'Red'))

info\_gain(true\_rows, false\_rows, current\_uncertainty)

def find\_best\_split(df):

    best\_gain = 0

    best\_question = None

    current\_uncertainty = gini(df)

    col = df.columns[:-1]

    for i in col:

        l = unique\_value(df,i)

        for j in l:

            question = Question(i,j)

            true\_rows,false\_rows = partition(df,question)

            if(len(true\_rows) == 0 or len(false\_rows) == 0):

                continue

            gain = info\_gain(true\_rows,false\_rows,current\_uncertainty)

            if(gain >= best\_gain):

                best\_gain, best\_question = gain, question

    return best\_gain, best\_question

best\_gain, best\_question = find\_best\_split(train)

class Leaf:

    def \_\_init\_\_(self,df):

        self.predictions = class\_counts(df)

class Decision\_Node:

    def \_\_init\_\_(self,question,true\_branch,false\_branch):

        self.question = question

        self.true\_branch = true\_branch

        self.false\_branch = false\_branch

def built\_tree(df):

    gain,question = find\_best\_split(df)

    if(gain == 0):

        return Leaf(df)

    true\_rows, false\_rows = partition(df,question)

    true\_branch = built\_tree(true\_rows)

    false\_branch = built\_tree(false\_rows)

    return Decision\_Node(question,true\_branch,false\_branch)

def print\_tree(node,spacing=""):

    if isinstance(node,Leaf):

        print(spacing + "Predcit",node.predictions)

        return

    print(spacing + str(node.question))

    print(spacing + "--> True:")

    print\_tree(node.true\_branch,spacing + " ")

    print(spacing + "--> False:")

    print\_tree(node.false\_branch,spacing + " ")

my\_tree = built\_tree(train)

print\_tree(my\_tree)

def Classify(df,node):

    if isinstance(node,Leaf):

        return node.predictions

    if node.question.match(df):

        return Classify(df,node.true\_branch)

    else:

        return Classify(df,node.false\_branch)

test = pd.DataFrame({"Color" : ["Green","Yellow","Red","Red","Yellow"],

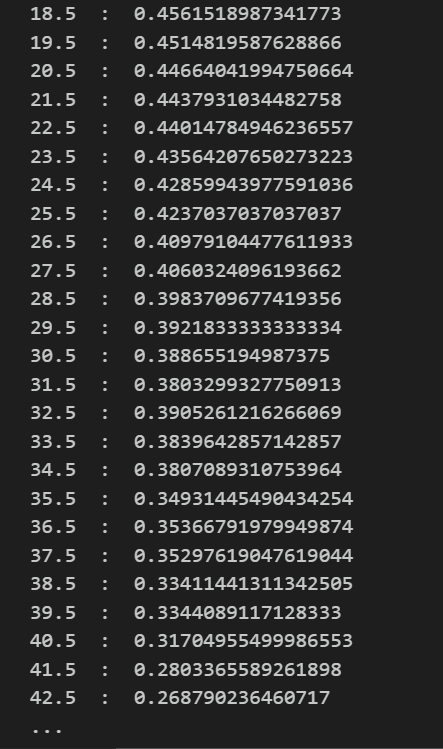
                     "Diameter" : [3,4,2,1,3]})

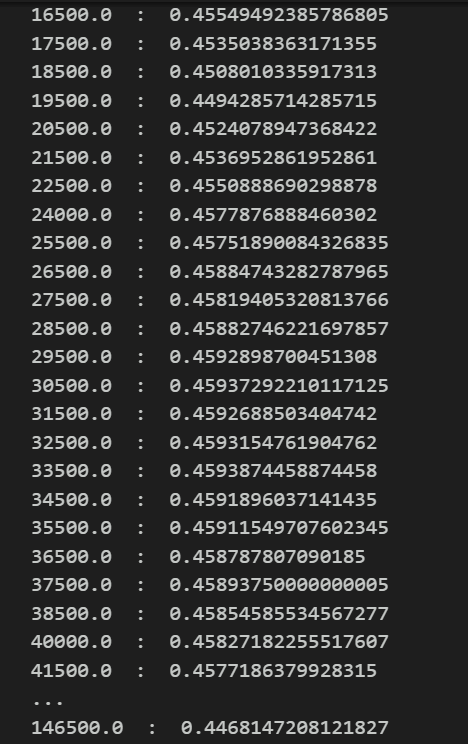
test

for i in range(test.shape[0]):

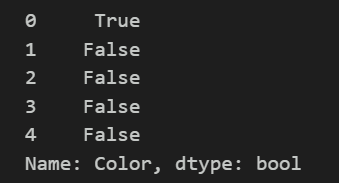
    print(Classify(test.iloc[i],my\_tree)

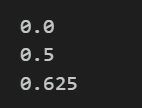
**Output:**



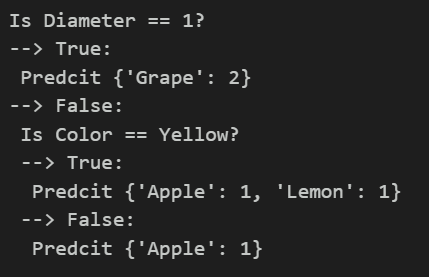
****

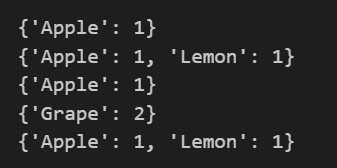
**2.**

****

****

****

****

****

**Result:** The above program to compute gini index and to build decision tree from scratch has been successfully executed.

|  |  |
| --- | --- |
| **Ex. No: 05** | **CLASSIFICATION ALGORITHMS** |
| **31/01/2024** |

**Aim:** To compare the performance of different classification algorithms (KNN, Logistic Regression, Naive Bayes, Decision Trees, SVM) on the teleco-customer-churn dataset.

**Algorithm:**

1. Load the teleco-customer-churn dataset into a Data Frame.
2. Perform exploratory data analysis (EDA) to understand the data's structure, distributions, and relationships.
3. Preprocess the data: a. Handle missing values, if any, using imputation techniques like mean, median, or mode. b. Encode categorical variables using one-hot encoding or label encoding. c. Scale numerical features using standardization or normalization.
4. Split the dataset into features (X) and the target variable (y).
5. Split the data into training and testing sets.
6. Apply each classification algorithm: a. Train a KNN (K-Nearest Neighbours) classifier. b. Train a Logistic Regression classifier. c. Train a Naive Bayes classifier. d. Train a Decision Trees classifier. e. Train a Support Vector Machine (SVM) classifier.
7. Evaluate the performance of each classifier using accuracy.
8. Print the accuracies of all the classifiers.
9. Determine which algorithm achieved the best accuracy.
10. Provide a justification for why that algorithm performed the best

**Program :**

import pandas as pd

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

from sklearn.tree import DecisionTreeClassifier, export\_graphviz

from sklearn.metrics import accuracy\_score,precision\_score

import pydotplus

from sklearn.linear\_model import LogisticRegression

from sklearn.naive\_bayes import GaussianNB

from IPython.display import Image

from sklearn.neighbors import KNeighborsClassifier

from sklearn.tree import DecisionTreeClassifier, export\_graphviz

df = pd.read\_csv('telco-customer-churn.csv')

df.head()

#one got encoding

col= ['gender','Partner','Dependents','PhoneService','MultipleLines','InternetService','DeviceProtection','OnlineSecurity','OnlineBackup','TechSupport','StreamingTV','StreamingMovies','PaperlessBilling','Churn']

data = pd.get\_dummies(df, columns=col, drop\_first=True)

data.head()

# Assume 'X' contains features and 'y' contains the target variable (churn)

X = data.iloc[:,:-1].values

y = data.iloc[:,-1].values

# Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3)

# Create and train the Decision Tree model

decision\_tree\_model = DecisionTreeClassifier()

decision\_tree\_model.fit(X\_train, y\_train)

# Make predictions on the test set

y\_pred = decision\_tree\_model.predict(X\_test)

# Assume 'X' contains features and 'y' contains the target variable (churn)

X = data.iloc[:,:-1].values

y = data.iloc[:,-1].values

# Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3)

# Create and train the Decision Tree model

decision\_tree\_model = DecisionTreeClassifier()

decision\_tree\_model.fit(X\_train, y\_train)

# Make predictions on the test set

y\_pred = decision\_tree\_model.predict(X\_test)

# performance metrics

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

print("Accuracy" ,accuracy)

precision = precision\_score(y\_test, y\_pred)

print("Precision:",precision)

knn\_model = KNeighborsClassifier(n\_neighbors=2)

knn\_model.fit(X\_train\_scaled, y\_train)

y\_pred = knn\_model.predict(X\_test\_scaled)

# performance metrics

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

print("Accuracy" ,accuracy)

precision = precision\_score(y\_test, y\_pred)

print("Precision:",precision)

reg= LogisticRegression()

reg.fit(x\_train, y\_train)

y\_pred =reg.predict(x\_test)

# performance metrics

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

print("Accuracy" ,accuracy)

precision = precision\_score(y\_test, y\_pred)

print("Precision:",precision)

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

svm\_model = SVC(kernel='linear', random\_state=42)

svm\_model.fit(X\_train\_scaled, y\_train)

y\_pred = svm\_model.predict(X\_test\_scaled)

#performance

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

print("Accuracy" ,accuracy)

precision = precision\_score(y\_test, y\_pred)

print("Precision:",precision)

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

naive\_bayes\_model = GaussianNB()

naive\_bayes\_model.fit(X\_train\_scaled, y\_train)

y\_pred = naive\_bayes\_model.predict(X\_test\_scaled)

# performance metrics

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

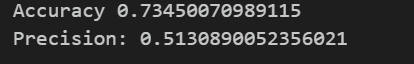
print("Accuracy" ,accuracy)

precision = precision\_score(y\_test, y\_pred)

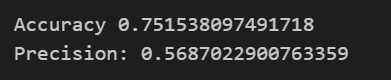
print("Precision:",precision)

**Output:**

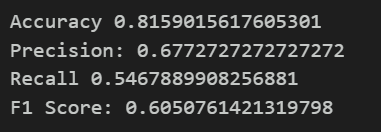
**Decision Tree:**

****

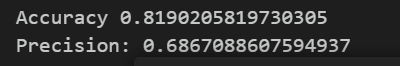
**KNN:**

****

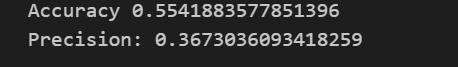
**Logistic Regression:**

****

**SVM:**

****

**Naïve bayes:**

****

**Result:** The above program to compare the performance of different classification algorithms (KNN, Logistic Regression, Naive Bayes, Decision Trees, SVM) has been successfully executed.

|  |  |
| --- | --- |
| **Ex. No: 06** | **GRIDSEARCHCV AND RANDOMIZEDSEARCHCV** |
| **21/02/2024** |

**Aim:** To Optimize classification algorithms (SVM, Decision Tree, Random Forest, and boosting) on a dataset by tuning parameters using GridSearchCV and RandomizedSearchCV to maximize accuracy.

**Algorithm:**

1. Load the dataset provided.
2. Perform any necessary preprocessing steps such as handling missing values, encoding categorical variables, and scaling numerical features.
3. Split the dataset into features (X) and the target variable (y).
4. For each classification algorithm (SVM, Decision Tree, Random Forest, and one boosting algorithm) identify the different tuneable parameters that can be adjusted to improve performance.
5. Apply GridSearchCV and RandomizedSearchCV to find the best combination of parameters for each algorithm
6. Fit each classification algorithm with the best parameters obtained from GridSearchCV or RandomizedSearchCV.
7. Evaluate the performance of each model using appropriate evaluation metrics such as accuracy, precision, recall, F1-score, etc., on a holdout test set.
8. Compare the performance of each algorithm and identify the best-performing one.

**Program :**

**DECISION TREE:**

import pandas as pd

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

from sklearn.tree import DecisionTreeClassifier

from sklearn.model\_selection import GridSearchCV, RandomizedSearchCV

from sklearn.metrics import classification\_report, confusion\_matrix

df = pd.read\_csv('telco-customer-churn.csv')

df.head()

#one hot encoding

col= ['gender','Partner','Dependents','PhoneService','MultipleLines','InternetService','DeviceProtection','OnlineSecurity','OnlineBackup','TechSupport','StreamingTV','StreamingMovies','PaperlessBilling','Churn']

data = pd.get\_dummies(df, columns=col, drop\_first=True)

data.head()

##segregate independent and dependent variable

X = data.iloc[:,:-1].values

y = data.iloc[:,-1].values

#build the model

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3)

# Decision Tree

dt\_model = DecisionTreeClassifier()

#tunable parameters for Decision Tree

print("Decision Tree Parameters:", dt\_model.get\_params())

dt\_model.fit(X\_train, y\_train)

# print prediction results

predictions = dt\_model.predict(X\_test)

print(classification\_report(y\_test, predictions))

# parameter grid for Decision Tree

dt\_param\_grid = {'criterion': ['gini', 'entropy'], 'max\_depth': [None, 10, 20, 30], 'min\_samples\_split': [2, 5, 10], 'min\_samples\_leaf': [1, 2, 4]}

#GridSearchCV for Decision Tree

dt\_grid\_search = GridSearchCV(dt\_model, dt\_param\_grid, cv=5)

dt\_grid\_search.fit(X\_train, y\_train)

print("Best Parameters for Decision Tree (GridSearchCV):", dt\_grid\_search.best\_params\_)

grid\_predictions =dt\_grid\_search.predict(X\_test)

# print classification report

print(classification\_report(y\_test, grid\_predictions))

#RandomizedSearchCV for Decision Tree

dt\_random\_search = RandomizedSearchCV(dt\_model, dt\_param\_grid, n\_iter=5, cv=5)

dt\_random\_search.fit(X\_train, y\_train)

print("Best Parameters for Decision Tree (RandomizedSearchCV):", dt\_random\_search.best\_params\_)

grid\_predictions =dt\_random\_search.predict(X\_test)

# print classification report

print(classification\_report(y\_test, grid\_predictions))

**SVM:**

##segregate independent and dependent variable

X = data.iloc[:,:-1].values

y = data.iloc[:,-1].values

#build the mode

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3)

model = SVC()

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

# print prediction results

predictions = model.predict(X\_test)

print(classification\_report(y\_test, predictions))

#to get all tunable parameters:

print(model.get\_params())

from sklearn.model\_selection import GridSearchCV

# defining parameter range

param\_grid = {'C': [0.1, 1, 10, 100, 1000],

              'gamma': [1, 0.1, 0.01, 0.001, 0.0001],

              'kernel': ['rbf']}

#perform grid search

grid = GridSearchCV(SVC(), param\_grid, refit = True, verbose = 3)

# fitting the model for grid search

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

#best parameter after tuning

print(grid.best\_params\_)

grid\_predictions = grid.predict(X\_test)

# print classification report

print(classification\_report(y\_test, grid\_predictions))

**ADABOOST:**

##segregate independent and dependent variable

X = data.iloc[:,:-1].values

y = data.iloc[:,-1].values

#build the model

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3,random\_state=20)

adaboost\_model = AdaBoostClassifier()

#tunable parameters for AdaBoost

print("AdaBoost Parameters:", adaboost\_model.get\_params())

adaboost\_model.fit(X\_train, y\_train)

# print prediction results

predictions = adaboost\_model.predict(X\_test)

print(classification\_report(y\_test, predictions))

#parameter grid for AdaBoost

adaboost\_param\_grid = {'n\_estimators': [50, 100, 200], 'learning\_rate': [0.01, 0.1, 0.2], 'algorithm': ['SAMME', 'SAMME.R']}

#GridSearchCV for AdaBoost

adaboost\_grid\_search = GridSearchCV(adaboost\_model, adaboost\_param\_grid, cv=5)

adaboost\_grid\_search.fit(X\_train, y\_train)

print("Best Parameters for AdaBoost (GridSearchCV):", adaboost\_grid\_search.best\_params\_)

grid\_predictions =adaboost\_grid\_search.predict(X\_test)

# print classification report

print(classification\_report(y\_test, grid\_predictions))

#RandomizedSearchCV for AdaBoost

adaboost\_random\_search = RandomizedSearchCV(adaboost\_model, adaboost\_param\_grid, n\_iter=5, cv=5)

adaboost\_random\_search.fit(X\_train, y\_train)

print("Best Parameters for AdaBoost (RandomizedSearchCV):", adaboost\_random\_search.best\_params\_)

grid\_predictions =adaboost\_random\_search.predict(X\_test)

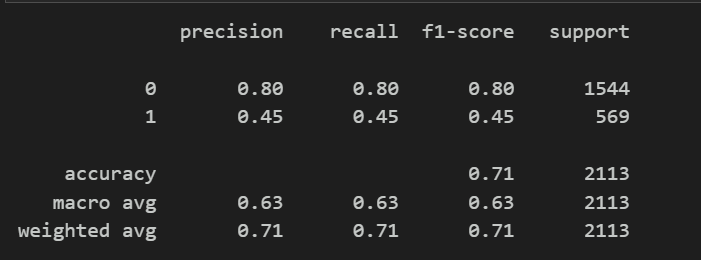
# print classification report

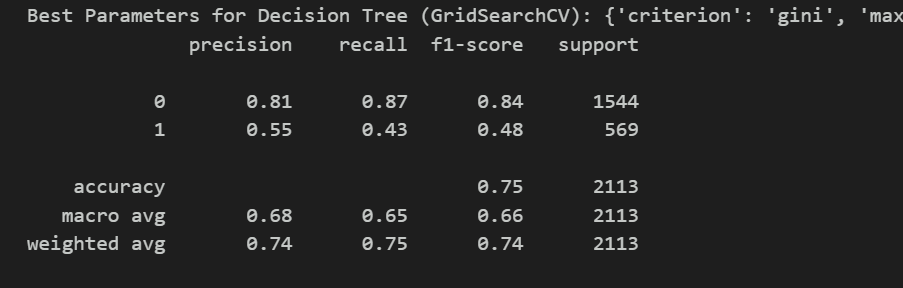
print(classification\_report(y\_test, grid\_predictions))

**Output:**

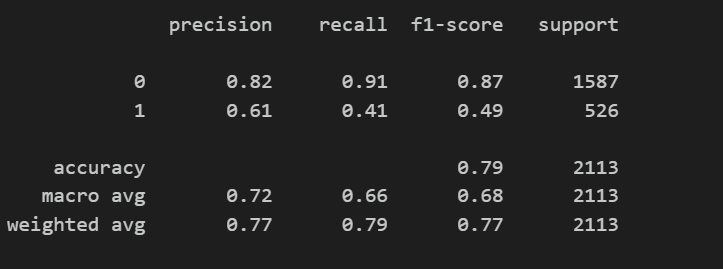
**DECISION TREE:**

****

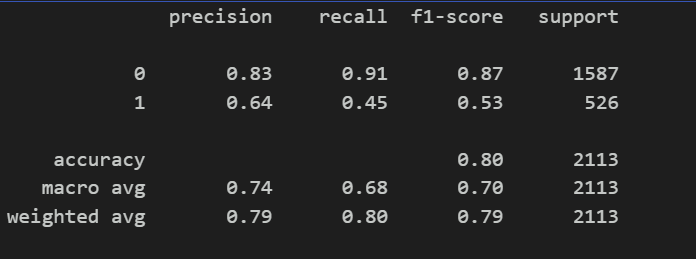
****

****

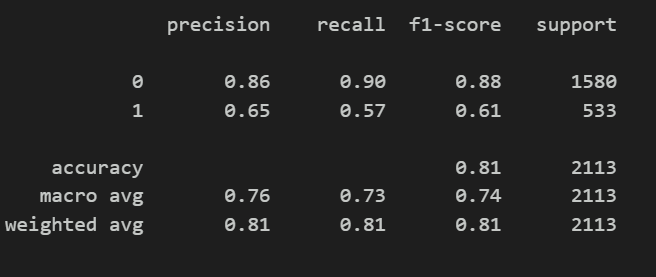
**SVM:**

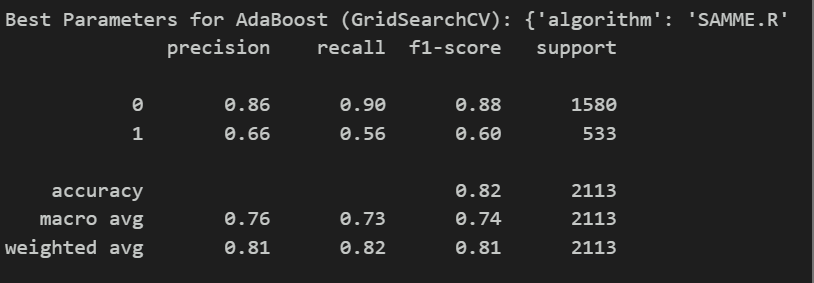
****

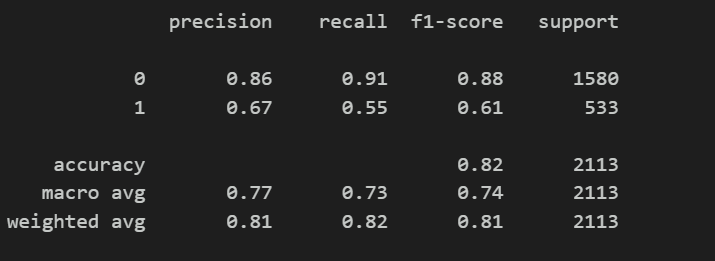
****

****

**ADABOOST:**

****

****

****

**Result:** The above program to optimize classification algorithms (SVM, Decision Tree, Random Forest, and boosting) on a dataset by tuning parameters using GridSearchCV and RandomizedSearchCV to maximize accuracy. has been successfully executed.

|  |  |
| --- | --- |
| **Ex. No: 07** | **K-MEANS** |
| **28/02/2024** |

**Aim:** to implement the K-Means clustering algorithm from scratch and compare its results with the implementation provided by scikit-learn.

**Algorithm:**

1. Load the given dataset.
2. Preprocess the data if necessary (e.g., handling missing values, scaling features).
3. Implement K-Means clustering algorithm from scratch:
   1. Initialize K cluster centroids randomly.
   2. Assign each data point to the nearest cluster centroid based on Euclidean distance.
   3. Update the cluster centroids by calculating the mean of all data points assigned to each cluster.
   4. Repeat steps b and c until convergence (either the cluster centroids stop changing significantly or a maximum number of iterations is reached).
4. Use the scikit-learn implementation of K-Means clustering
5. Compare the results of the two implementations

**Program :**

import pandas as pd

import numpy as np

from sklearn.cluster import KMeans

from sklearn.metrics import silhouette\_score

# Function to implement K-Means from scratch

def kmeans\_from\_scratch(X, n\_clusters, max\_iters=400):

    # Initialize centroids randomly

    centroids = X[np.random.choice(X.shape[0], n\_clusters, replace=False)]

    for i in range(max\_iters):

        # Assign each data point to the closest centroid

        labels = np.argmin(np.linalg.norm(X[:, np.newaxis] - centroids, axis=2), axis=1)

        # Update centroids

        new\_centroids = np.array([X[labels == i].mean(axis=0) for i in range(n\_clusters)])

        # Check for convergence

        if np.all(centroids == new\_centroids):

            break

        centroids = new\_centroids

    return labels, centroids

df1 =pd.read\_csv('data (1).csv')

df = df1.sort\_values(by='Age', ascending=True)

col= ['Gender']

data = pd.get\_dummies(df, columns=col, drop\_first=True)

data.head()

n\_clusters = 5

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

data\_scaled = scaler.fit\_transform(data)

#K-Means from scratch

labels\_scratch, centroids\_scratch = kmeans\_from\_scratch(data.values, n\_clusters)

#K-Means using sklearn

kmeans\_sklearn = KMeans(n\_clusters=n\_clusters,init='k-means++',  max\_iter=400,random\_state=90)

labels\_sklearn = kmeans\_sklearn.fit\_predict(data\_scaled)

# Print the silhouette scores

from sklearn.metrics import silhouette\_score

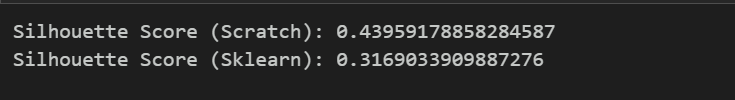
silhouette\_scratch = silhouette\_score(data, labels\_scratch)

silhouette\_sklearn = silhouette\_score(data\_scaled, labels\_sklearn)

print(f"Silhouette Score (Scratch): {silhouette\_scratch}")

print(f"Silhouette Score (Sklearn): {silhouette\_sklearn}")

**Output:**

****

**Result**: The above program to implement the K-Means clustering algorithm from scratch and compare its results with the implementation provided by scikit-learn has been successfully executed.

|  |  |
| --- | --- |
| **Ex. No: 08** | **PRINCIPAL COMPONENT ANALYSIS**  **(PCA)** |
| **13/03/2024** |

**Aim:** To apply Principal Component Analysis (PCA) from scratch to the MNIST dataset.

**Algorithm:**

1. Download the MNIST dataset.
2. Preprocess the MNIST dataset.
3. Compute the covariance matrix.
4. Perform eigen decomposition.
5. Select the top k eigenvectors.
6. Project the data onto the new feature space.
7. Apply PCA to reduce the dimensionality of the MNIST dataset
8. Analyse the results.

**Program:**

# MNIST dataset downloaded from Kaggle : https://www.kaggle.com/c/digit-recognizer/data

# importing necessary libraries

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

# load data from csv into pandas dataframe

mnist\_df = pd.read\_csv('train.csv')

# display first few rows

mnist\_df.head()

# store the labels into a variable y

y = mnist\_df['label']

# store the pixel data in X

X = mnist\_df.drop("label", axis = 1)

# shape of data

print(X.shape)

print(y.shape)

(42000, 784)

(42000,)

# display some random number from datset

for i in [2, 99, 10001, 20422]:

    index = i

    plt.figure(figsize = (3,3))

    grid\_data = X.iloc[index].values.reshape(28,28)  # reshape from 1d to 2d pixel array

    plt.imshow(grid\_data, interpolation = "none", cmap = 'gray')

    plt.show()

    # print the corresponding class label

    print("- "\*50)

    print("class label:", y[index])

# Data-preprocessing: Standardizing the data with standard scaler

from sklearn.preprocessing import StandardScaler

X\_s= StandardScaler()

X\_std=X\_s.fit\_transform(X)

print(X\_std.shape)

# find the co-variance matrix:  (A^T \* A)/n

# matrix multiplication using numpy

covar\_matrix = np.matmul(X\_std.T , X\_std)/X\_std.shape[1]

print ( "The shape of covariance matrix = ", covar\_matrix.shape)

# find top two eigen-values and corresponding eigen-vectors for projection on a 2-D space

from scipy.linalg import eigh

# the parameter 'eigvals' is defined (low value to high value)

# eigh function will return the eigen values in asending order

# this code generates only the top 2 (782 and 783) eigenvalues

values, vectors = eigh(covar\_matrix, eigvals = (782,783))

print("Shape of eigen vectors: ",vectors.shape)

# projecting the original data sample on the plane formed by two principal eigen vectors by vector-vector multiplication

new\_coordinates = np.matmul(vectors.T, X\_std.T)

print ("resultant new data points' shape ", vectors.T.shape, "X", X\_std.T.shape," = ", new\_coordinates.shape)

# appending class label to the new coordinated in 2D projected space

new\_coordinates = np.vstack((new\_coordinates, y)).T

# creating a new data frame for plotting the labeled points

new\_df = pd.DataFrame(data = new\_coordinates, columns = ("2nd\_principal", "1st\_principal", "label"))

print(new\_df.head())

# ploting the 2D data points with seaborn

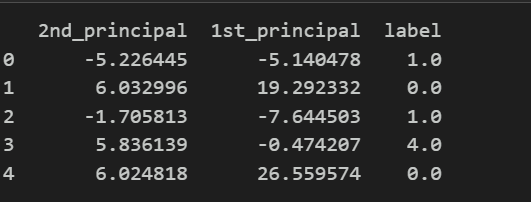
sns.lmplot(x = '1st\_principal', y = '2nd\_principal', data = new\_df,

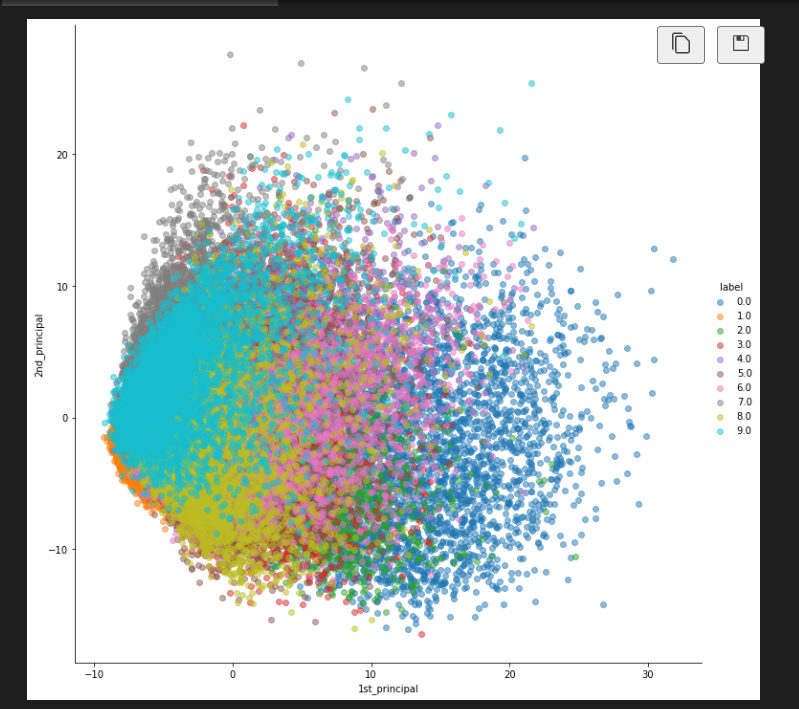
           hue = 'label', legend = True, legend\_out = True, height = 10,

           fit\_reg = False, scatter\_kws = {'alpha': 0.5})

plt.show()

**Output:**

****



**Result:** The above program to apply Principal Component Analysis (PCA) from scratch to the MNIST dataset. has been successfully executed.

|  |  |
| --- | --- |
| **Ex. No: 09** | **NEURAL NETWORK** |
| **19/03/2024** |

**Aim:** To compare the performance of a neural network for regression built from scratch with one built using the Keras library, using the MNIST dataset.

**Algorithm:**

1. Initialize parameters: Randomly initialize weights and biases.
2. Forward propagation: Compute predictions using the current weights and biases.
3. Compute cost: Calculate the mean squared error (MSE) between predictions and actual values.
4. Backpropagation: Compute gradients of the cost function with respect to weights and biases.
5. Update parameters: Update weights and biases using gradient descent.
6. Repeat steps 2-5 for a number of iterations (epochs).

**Program :**

FROM SCRATCH:

#import libraries

import numpy as np

from mnist import MNIST

from urllib.request import urlretrieve

import gzip

import os

#to download and extract MNIST dataset

def load\_mnist():

    url\_base = 'http://yann.lecun.com/exdb/mnist/'

    file\_names = ['train-images-idx3-ubyte.gz', 'train-labels-idx1-ubyte.gz',

                  't10k-images-idx3-ubyte.gz', 't10k-labels-idx1-ubyte.gz']

    save\_path = './mnist\_data/'

    if not os.path.exists(save\_path):

        os.makedirs(save\_path)

    for file\_name in file\_names:

        url = (url\_base + file\_name).format(\*\*locals())

        file\_path = save\_path + file\_name

        if not os.path.exists(file\_path):

            print('Downloading ' + file\_name + ' ... ')

            urlretrieve(url, file\_path)

        else:

            print('Already exists: ' + file\_name)

        with gzip.open(file\_path, 'rb') as f\_in:

            file\_content = f\_in.read()

            with open(file\_path.replace('.gz', ''), 'wb') as f\_out:

                f\_out.write(file\_content)

        print('Extracting ' + file\_name + ' ...')

# Load MNIST dataset

def load\_data():

    mnist = MNIST('./mnist\_data/')

    images, \_ = mnist.load\_training()

    images = np.array(images) / 255.0

    images = images.reshape((-1, 784))

    return images

# Load MNIST dataset

load\_mnist()

images = load\_data()

# Generate random weights and bias

np.random.seed(42)

input\_size = 784

hidden\_size = 128

output\_size = 784

weights1 = np.random.randn(input\_size, hidden\_size)

bias1 = np.random.randn(hidden\_size)

weights2 = np.random.randn(hidden\_size, output\_size)

bias2 = np.random.randn(output\_size)

#activation function used-sigmoid

def sigmoid(x):

    return 1 / (1 + np.exp(-x))

#derivative of activation function

def sigmoid\_derivative(x):

    return x \* (1 - x)

# Training parameters

learning\_rate = 0.001

epochs = 200   #200 iterations

# Training loop

for epoch in range(epochs):

    # Forward propagation

    hidden\_layer\_input = np.dot(images, weights1) + bias1

    hidden\_layer\_output = sigmoid(hidden\_layer\_input)

    output\_layer\_input = np.dot(hidden\_layer\_output, weights2) + bias2

    output\_layer\_output = sigmoid(output\_layer\_input)

    # Calculate loss

    loss = np.mean(np.square(output\_layer\_output - images))

    # Backpropagation

    output\_error = output\_layer\_output - images

    output\_delta = output\_error \* sigmoid\_derivative(output\_layer\_output)

    hidden\_error = np.dot(output\_delta, weights2.T)

    hidden\_delta = hidden\_error \* sigmoid\_derivative(hidden\_layer\_output)

    # Update weights and bias

    weights2 -= learning\_rate \* np.dot(hidden\_layer\_output.T, output\_delta)

    bias2 -= learning\_rate \* np.sum(output\_delta, axis=0)

    weights1 -= learning\_rate \* np.dot(images.T, hidden\_delta)

    bias1 -= learning\_rate \* np.sum(hidden\_delta, axis=0)

    if epoch % 10 == 0:

        print(f'Epoch {epoch}, Loss: {loss}')

print(f'Final Loss: {loss}')

USING KERAS:

# Load MNIST dataset

# Function to download and extract MNIST dataset

def load\_mnist():

    url\_base = 'http://yann.lecun.com/exdb/mnist/'

    file\_names = ['train-images-idx3-ubyte.gz', 'train-labels-idx1-ubyte.gz',

                  't10k-images-idx3-ubyte.gz', 't10k-labels-idx1-ubyte.gz']

    save\_path = './mnist\_data/'

    if not os.path.exists(save\_path):

        os.makedirs(save\_path)

    for file\_name in file\_names:

        url = (url\_base + file\_name).format(\*\*locals())

        file\_path = save\_path + file\_name

        if not os.path.exists(file\_path):

            print('Downloading ' + file\_name + ' ... ')

            urlretrieve(url, file\_path)

        else:

            print('Already exists: ' + file\_name)

        with gzip.open(file\_path, 'rb') as f\_in:

            file\_content = f\_in.read()

            with open(file\_path.replace('.gz', ''), 'wb') as f\_out:

                f\_out.write(file\_content)

        print('Extracting ' + file\_name + ' ...')

# Load MNIST dataset

def load\_data():

    mnist = MNIST('./mnist\_data/')

    images, \_ = mnist.load\_training()

    images = np.array(images) / 255.0

    images = images.reshape((-1, 784))

    return images

# Load MNIST dataset

load\_mnist()

images = load\_data()

# Convert images to numpy arrays

images = np.array(images)

# Normalize pixel values

images = images / 255.0

# Flatten images

images = images.reshape((-1, 784))

#neural network parameters

model = Sequential([

    Dense(128, input\_shape=(784,), activation='sigmoid'),

    Dense(784, activation='sigmoid')

])

# Compile model

sgd = SGD(learning\_rate=0.001)

model.compile(optimizer=sgd, loss='mean\_squared\_error',metrics=['accuracy'])

# Training parameters

epochs = 200

batch\_size = 32

# Train model

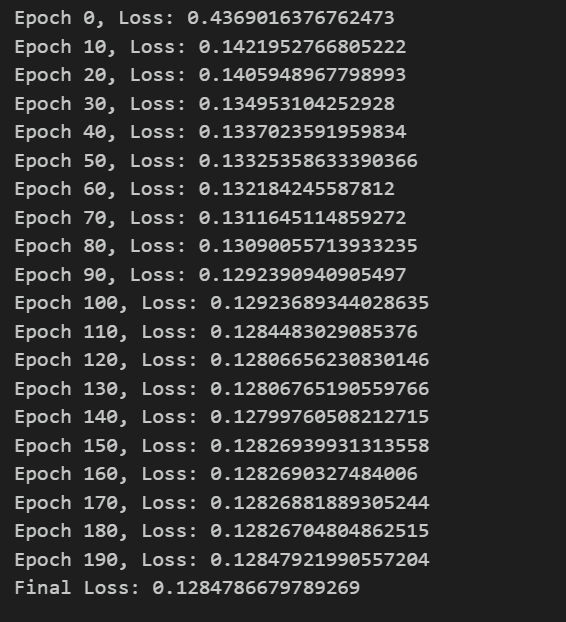
model.fit(images, images, epochs=epochs, batch\_size=batch\_size, verbose=1)

# Calculate loss

loss = model.evaluate(images, images, verbose=0)

print(f'Keras Model Final Loss: {loss}')

**Output:**





**Result:** The above program to execute neural network from scratch and using keras library has been successfully executed.

|  |  |
| --- | --- |
| **Ex. No: 10** | **ASSOCIATION RULE LEARNING** |
| **28/03/2024** |

**Aim:** To Implement Apriori from scratch on the given dataset.

**Algorithm:**

1. Initialize a list of frequent item sets (F) with frequent individual items.
2. Set the minimum support threshold (min\_sup) for frequent item sets.
3. Generate candidate item sets by combining frequent item sets of size k to create item sets of size k+1.
4. Prune candidate item sets that contain subsets which are not frequent. This step is based on the Apriori property which states that if an itemset is infrequent, all its supersets will also be infrequent.
5. Count the support for each candidate itemset by scanning the transaction dataset.
6. Discard candidate item sets whose support is below the minimum support threshold.
7. Repeat steps 3-5 until no new frequent item sets can be generated.
8. Generate association rules from the set of frequent item sets.
9. Calculate confidence for each rule and discard rules with confidence below a specified threshold.
10. Output the frequent item sets found in the dataset along with their support values.

**Program :**

import pandas as pd

from itertools import combinations

dataset = pd.read\_csv('Market\_Basket\_Optimisation.csv', header=None)

dataset = dataset.fillna(0)

transactions = []

for i in range(0, 7501):

    transactions.append([str(dataset.values[i, j]) for j in range(0, 20)])

def create\_candidates(transactions, length):

    """Create initial candidate itemsets of length 1"""

    candidates = []

    for transaction in transactions:

        for item in transaction:

            if [item] not in candidates:

                candidates.append([item])

    candidates.sort()

    return list(map(frozenset, candidates))

def scan\_database(transactions, candidates, min\_support):

    """Scan the database and count the support of each candidate itemset"""

    itemset\_counts = {}

    for transaction in transactions:

        for candidate in candidates:

            if candidate.issubset(transaction):

                itemset\_counts[candidate] = itemset\_counts.get(candidate, 0) + 1

    num\_transactions = float(len(transactions))

    frequent\_itemsets = []

    support\_data = {}

    for itemset in itemset\_counts:

        support = itemset\_counts[itemset] / num\_transactions

        if support >= min\_support:

            frequent\_itemsets.append(itemset)

        support\_data[itemset] = support

    return frequent\_itemsets, support\_data

def generate\_candidates(frequent\_itemsets, length):

    """Generate candidate itemsets of length k from frequent itemsets of length k-1"""

    candidates = []

    num\_frequent\_itemsets = len(frequent\_itemsets)

    for i in range(num\_frequent\_itemsets):

        for j in range(i + 1, num\_frequent\_itemsets):

            itemset1 = list(frequent\_itemsets[i])[:length - 2]

            itemset2 = list(frequent\_itemsets[j])[:length - 2]

            itemset1.sort()

            itemset2.sort()

            if itemset1 == itemset2:

                candidates.append(frequent\_itemsets[i] | frequent\_itemsets[j])

    return candidates

def apriori(transactions, min\_support=0.5):

    """Apriori algorithm"""

    candidates = create\_candidates(transactions, 1)

    frequent\_itemsets1, support\_data = scan\_database(transactions, candidates, min\_support)

    frequent\_itemsets = [frequent\_itemsets1]

    k = 2

    while len(frequent\_itemsets[k - 2]) > 0:

        candidates = generate\_candidates(frequent\_itemsets[k - 2], k)

        frequent\_itemsets\_k, support\_data\_k = scan\_database(transactions, candidates, min\_support)

        support\_data.update(support\_data\_k)

        frequent\_itemsets.append(frequent\_itemsets\_k)

        k += 1

    return frequent\_itemsets, support\_data

if \_\_name\_\_ == "\_\_main\_\_":

    frequent\_itemsets, support\_data = apriori(transactions, min\_support=0.03)

    """Print results"""

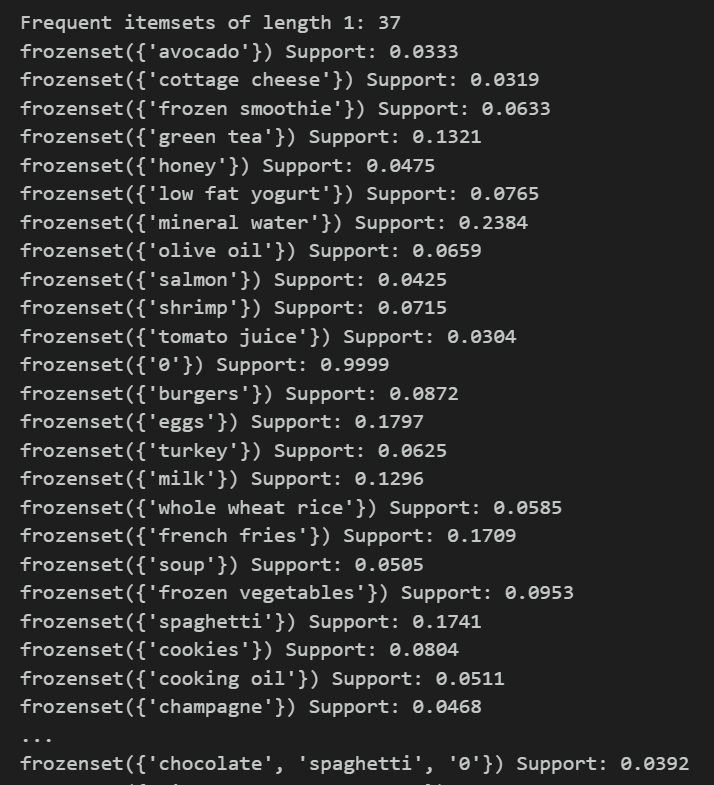
    for i, itemset\_list in enumerate(frequent\_itemsets):

        print("Frequent itemsets of length {}: {}".format(i+1, len(itemset\_list)))

        for itemset in itemset\_list:

            print(itemset, "Support:", round(support\_data[itemset], 4))

**Output:**



**Result:** The above program to implement apriori algorithm from scratch has been successfully executed.