EX No	Name Of Experiment	Date
1	Linear Regression	03-01-24
2	Multiple Regression	17-01-24
3	Logistic Regression	24-01-24
4	Decision Trees	31-01-24
5	Customer Churn Prediction	03-02-24
6	Hyperparameter Tuning	28-02-24
7	KMeans From Scratch	06-03-24
8	PCA MNIST	19-03-24
9	Neural Networks	19-03-24
10	Association Rule Mining	03-04-24

Ex. No: 1	Linear Regression
03.01.2024	

Aim:

To simulate and Create a Simple Linear Regression Model From Scratch using pearson correlation .

1. Create a random 2-D numpy array with 1500 values. Simulate different lines of fit using 1000 values from the array and find the errors for each of these lines. Find the line with the least error among these lines and store it as the line of best fit. Using this line of best fit, predict the target variable for the other 500 values.

2. Use the data1.csv to build a simple linear regression from scratch without using sklearn libraries and print the RMSE and mean absolute error values. Use both the equations available in the slides (in theory page) to build the model and compare the intercept and coefficient values.

Algorithm:

Question 1:

1. Create a random 2-d array using uniform / random distribution
2. Initialize the 2-D Matrix (750*2)->1500 Values
3. Convert to Data Frame for convenience
4. Using the LinearFitSimulation Class Simulate Random fit using Random Slope and intercept
5. Compute Error and minimize error to find the line of best fit.
6. Plot the line of best fit

Question 2:

1. Simulate the LinearRegression Line of best fit using Least Squares Method
2. M=∑((x-x')*(y-y'))/ ∑(x-x')**2
3. C=y'-mx'
4. Line of Best Fit can also be found using Pearson Coefficient
5. R=∑ ((x-x')*(y-y'))/√ ∑ ((x-x')**2)* ∑ (y-y')**2))
6. B1=R(y_std/x_std)
7. B0=y'-B1*x'
8. M=b1
9. C=b0
10. X_std=√ ∑ ((x-x')**2)/len(x)
11. Y_std=√ ∑ ((y-y')**2)*/len(y)
12. Once the data is fit compute error using suitable metric such as MSE /MAE/RMSE
13. Plot the Line of best fit

Code+Output:

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

0.0.1 Question 1 :

Create a random 2-D numpy array with 1500 values. Simulate different
lines of fit using 1000 values from the array and find the errors for each of
these lines. Find the line with the least error among these lines and store
it as the line of best fit. Using this line of best fit, predict the target variable
for the other 500 values.

0.0.2 Class Native to The given Question

Essentially Create A Random 2-D Array with NOT Normal Distribution Of random Function
So I Used Triangular Distribution . The Normal Distribution of Random Numbers is not
suitable for Linear Fit As such So Triangular Dist. is the closest to a nice data set
for a fit

PreProcessPipeline:

Perform The required pre-processing by

Initializing 2-d matrix (750x2)->1500 val

Convert to a DataFrame to work with and Rename columns

[2]:
```
'''
    Pre Processing is not Generic Though I have made it Generic to THIS Problem
'''
class PreProcessPipeline():
    def __init__(self):
        dataFrame=pd.DataFrame(np.random.triangular(-50,0,50,size=(750,2)))
        dataFrame=dataFrame.rename(columns={0:"X",1:"Y"})
```

```
        self.dataframe=dataFrame
        self._shuffle_data()
    def _shuffle_data(self):
        self.dataframe= self.dataframe.sample(frac=1).reset_index(drop=True)
    def X(self):
        return self.dataframe.iloc[:,:-1]["X"]
    def Y(self):
        return self.dataframe.iloc[:,-1]
pp=PreProcessPipeline()
x=pp.X()
y=pp.Y()
pp.dataframe
```

[2]:

	X	Y
0	15.057165	-8.169659
1	43.533611	-21.400248
2	-6.072795	-42.972982
3	-21.381086	13.919916
4	8.790788	-0.668049
..	…	…
745	-16.602356	1.681381
746	-16.612151	24.436824
747	11.036708	11.858102
748	-5.278104	-0.467222
749	-20.953278	-15.158881

[750 rows x 2 columns]

0.0.3 Class To Perform Train Test Split

[3]:
```
class Train_Test_Split:
    def __init__(self,x,y,split_size=0.667):
        self.x=x
        self.y=y
        self.split_size=split_size
    def split(self):
        return (x[:round(self.x.size*self.split_size)],x[round(self.x.size*self.
 ↪split_size):],y[:round(self.y.size*self.split_size)],y[round(self.y.
 ↪size*self.split_size):])
```

[4]:
```
x_train,x_test,y_train,y_test=Train_Test_Split(x,y).split()
```

0.0.4 Simulation Model

This Class Performs Multiple Random Line Fits By Altering its
weight (m) and Bias (_intercept) and we select the best parameters
based on the Error computed.

[5]:
```
class LinearFitSimulation():
    def __init__(self,x_train,x_test,y_train,y_test,epochs=500):
        self.epochs=epochs
        self.x_train=x_train
        self.x_test=x_test
        self.y_train=y_train
        self.y_test=y_test
    def predict(self):
        best_err=np.inf
        best_line=None
        for i in range(self.epochs):
            m=np.random.randn()
            c=np.random.randn()
            y_pred=m*self.x_train+c
            err= Error_Suite(self.y_train,y_pred).mse()
        if err<best_err:
            best_err=err
            best_line=(m,c)
        return (best_line,best_err,x_test*best_line[0]+best_line[-1])
```

0.0.5 Error Suite Class

Essentially A Generic Class To compute Various Error Metrics
from (y_test,y_pred) for sake of conveinience

[6]:
```
class Error_Suite:
    def __init__(self,y_test,y_pred):
        self.y_test=y_test
        self.y_pred=y_pred
        self.size=y_test.size
    def mse(self):
        return(np.mean((self.y_test-self.y_pred)**2))
    def mae(self):
        return (abs(self.y_test-self.y_pred).sum()/self.size)
    def mape(self):
        return (abs((self.y_test-self.y_pred)/self.y_test)).sum()*100/self.size
    def rmse(self):
        return self.mse()**0.5
```

0.0.6 Plot Function

Again For convenience sake and Code Reusablity . I wrote down this fn.

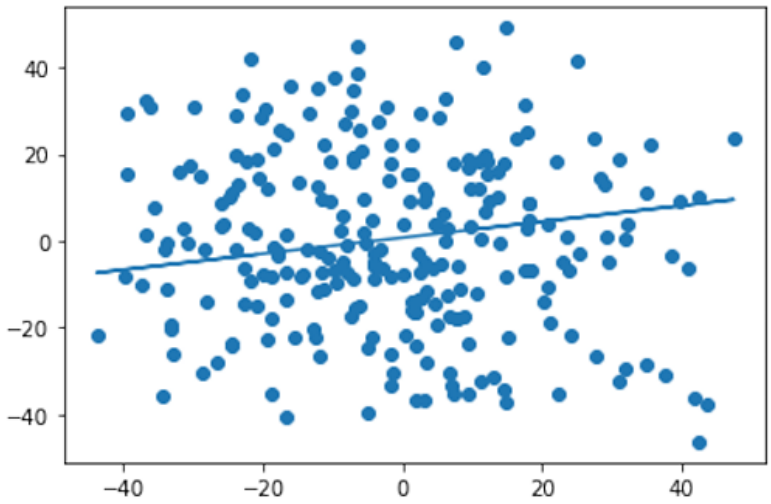
[7]:
```
class Plot():
    def __init__(self,x_test,y_test,y_pred):
        self.x_test=x_test
        self.y_test=y_test
        self.y_pred=y_pred
```

```
        plt.scatter(x_test,y_test)
        plt.plot(x_test,y_pred)
```

[10]:
```
l=LinearFitSimulation(x_train,x_test,y_train,y_test,epochs=1000).predict()
(y_test,l[-1])
```

[10]:
```
(500    16.713255
 501    -8.110374
 502     0.974025
 503    -8.181004
 504    -0.984421
          …
 745     1.681381
 746    24.436824
 747    11.858102
 748    -0.467222
 749   -15.158881
Name: Y, Length: 250, dtype: float64,
 500    2.476358
 501    3.463479
 502    6.165586
 503   -6.596648
 504   -0.706523
          …
 745   -2.322694
 746   -2.324505
 747    2.786670
 748   -0.229290
 749   -3.127006
Name: X, Length: 250, dtype: float64)
```

[11]:
```
z=Plot(x_test,y_test,l[-1])
```



0.0.7 Interpretation

The Given Data or the data that I had Used is A Triangular Distribution (-50,0,50)
which follows Somewhat Linear Ascent till Mode And Descent from Mode to the right .
So the Fit May Not be Exact since the distribution peaks around 0 and descends around [0,50)

0.0.8 Question 2:

Use the data1.csv to build a simple linear regression from scratch without
using sklearn libraries and print the RMSE and mean absolute error values.
Use both the equations available in the slides (in theory page) to build the
model and compare the intercept and coefficient values

0.0.9 Simple Linear Regression Model

This Class Essentially Performs Linear Regression using Least Squares Method.

[52]:
```
class SimpleLRModel():
    def __init__(self,x_train,x_test,y_train,y_test):
        self.x_train=x_train.flatten()
        self.x_test=x_test.flatten()    # This is necessary because without␣
↪flattening each element in x_test/train will be a sub array then being a 2-d␣
↪array messing up the computation
        self.y_train=y_train
        self.y_test=y_test
        self._slope=0
        self._intercept=0
```

```
    def fit(self):
        n=len(self.x_train)
        m_n=n*((self.x_train*self.y_train).sum())-(self.x_train.sum()*(self.
↪y_train.sum()))
        m_d=n*((self.x_train**2).sum())-((self.x_train).sum())**2
        self._slope=m_n/m_d
        self._intercept=((self.y_train.sum())-self._slope*(self.x_train.sum()))/
↪n
        return (self._slope,self._intercept)

    def predict(self):
        return self._slope*x_test+self._intercept
```

0.0.10 Simple Linear Regression Model Using Pearson Coeffiecient

[51]:
```
class SimpleLRModel_Pearson():
    def __init__(self,x_train,x_test,y_train,y_test):
        self.x_train=x_train.flatten()
        self.x_test=x_test.flatten()   # This is necessary because without␣
↪flattening each element in x_test/train will be a sub array then being a 2-d␣
↪array messing up the computation
        self.y_train=y_train
        self.y_test=y_test
        self._slope=0
        self._intercept=0


    def fit(self):
        x_mean=self.x_train.mean()
        y_mean=self.y_train.mean()
        x_std=np.sqrt(((self.x_train-x_mean)**2).sum()/len(self.x_train))
        y_std=np.sqrt(((self.y_train-y_mean)**2).sum()/len(self.y_train))
        #z_x=(self.x_train-x_mean)/x_std
        #z_y=(self.y_train-y_mean)/y_std
        #r=(z_x*z_y).sum()/len(self.x_train)-1
        r=((self.x_train-x_mean)*(self.y_train-y_mean)).sum()/np.sqrt(((self.
↪x_train-x_mean)**2).sum()*((self.y_train-y_mean)**2).sum())

        b1=r*(y_std/x_std)

        b0=y_mean-b1*x_mean
        self._slope =b1
```

```
        self._intercept=b0
        return (b1,b0)

    def predict(self):
        return self._slope*x_test+self._intercept
```

0.0.11 Data Pre processing

[48]:
```
data=pd.read_csv(r"D:\data1.csv")
x=data.iloc[:,:-1].values
y=data.iloc[:,-1].values
```

[56]:
```
x_train,x_test,y_train,y_test=Train_Test_Split(x,y).split()
```

0.0.12 Model Training (Least Squares)

[57]:
```
lrm=SimpleLRModel(x_train,x_test,y_train,y_test)
lrm.fit()
```

[57]:
```
(3.1792452830188678, 30.10377358490566)
```

0.0.13 Model prediction (Least Squares)

[58]:
```
y_pred=lrm.predict().flatten()
(y_pred,y_test)
```

[58]:
```
(array([84.1509434 , 71.43396226, 68.25471698, 77.79245283, 80.97169811,
        74.61320755, 80.97169811, 80.97169811, 87.33018868, 90.50943396,
        84.1509434 , 71.43396226, 68.25471698, 77.79245283, 80.97169811,
        74.61320755, 80.97169811, 80.97169811, 87.33018868, 90.50943396,
        84.1509434 , 71.43396226, 68.25471698, 77.79245283, 80.97169811,
        74.61320755, 80.97169811, 80.97169811, 87.33018868, 90.50943396,
        84.1509434 , 71.43396226, 68.25471698, 77.79245283, 80.97169811,
        74.61320755, 80.97169811, 80.97169811, 87.33018868, 90.50943396]),
 array([94, 73, 59, 80, 93, 85, 66, 79, 77, 91, 94, 73, 59, 80, 93, 85, 66,
        79, 77, 91, 94, 73, 59, 80, 93, 85, 66, 79, 77, 91, 94, 73, 59, 80,
        93, 85, 66, 79, 77, 91], dtype=int64))
```

0.0.14 Model Error Metrics (Least Squares)

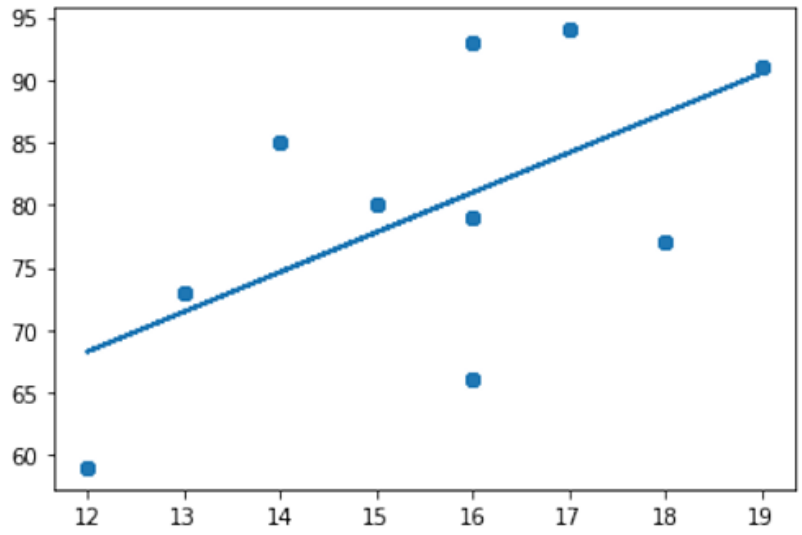
[59]:
```
mse=Error_Suite(y_test,y_pred).mse()
rmse=Error_Suite(y_test,y_pred).rmse()
mape=Error_Suite(y_test,y_pred).mape()
mae=Error_Suite(y_test,y_pred).mae()
```

```
print(f"Mean Squared Error :{mse}\nRoot Mean Squared Error:{rmse}\nMean␣
  ↪Percentage Error:{mape}\nMean AbsoluteError:{mae}\n")
```

```
Mean Squared Error :77.75377358490566
Root Mean Squared Error:8.817810022046611
Mean Percentage Error:9.535691016912438
Mean AbsoluteError:7.30566037735849
```

[60]:
```
plt.scatter(x_test,y_test)
plt.plot(x_test,y_pred)
```

[60]:
```
[<matplotlib.lines.Line2D at 0x1bee3dbf4f0>]
```



0.0.15 Model Fit Using Pearson Correlation

[61]:
```
reg=SimpleLRModel_Pearson(x_train,x_test,y_train,y_test)
reg.fit()
```

[61]:
```
(3.179245283018868, 30.10377358490566)
```

[62]:
```
y_pred_pearson=reg.predict()
(y_test,y_pred_pearson.flatten())
```

[62]:
```
(array([94, 73, 59, 80, 93, 85, 66, 79, 77, 91, 94, 73, 59, 80, 93, 85, 66,
        79, 77, 91, 94, 73, 59, 80, 93, 85, 66, 79, 77, 91, 94, 73, 59, 80,
```

```
       93, 85, 66, 79, 77, 91], dtype=int64),
array([84.1509434 , 71.43396226, 68.25471698, 77.79245283, 80.97169811,
       74.61320755, 80.97169811, 80.97169811, 87.33018868, 90.50943396,
       84.1509434 , 71.43396226, 68.25471698, 77.79245283, 80.97169811,
       74.61320755, 80.97169811, 80.97169811, 87.33018868, 90.50943396,
       84.1509434 , 71.43396226, 68.25471698, 77.79245283, 80.97169811,
       74.61320755, 80.97169811, 80.97169811, 87.33018868, 90.50943396,
       84.1509434 , 71.43396226, 68.25471698, 77.79245283, 80.97169811,
       74.61320755, 80.97169811, 80.97169811, 87.33018868, 90.50943396]))
```

0.0.16 Model Error metrics (Pearson Correlation)

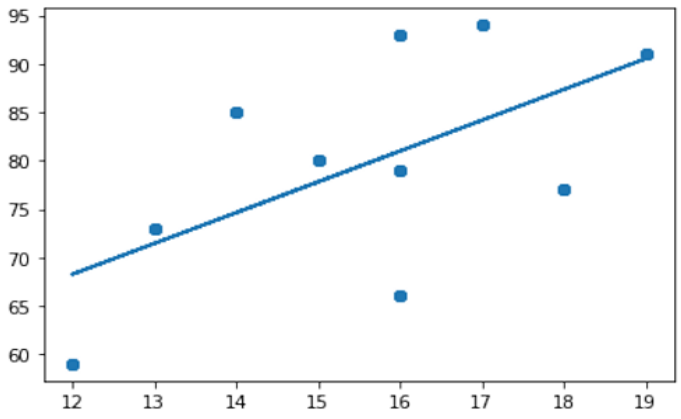
[64]:
```
mse=Error_Suite(y_test,y_pred_pearson.flatten()).mse()
rmse=Error_Suite(y_test,y_pred_pearson.flatten()).rmse()
mape=Error_Suite(y_test,y_pred_pearson.flatten()).mape()
mae=Error_Suite(y_test,y_pred_pearson.flatten()).mae()

print(f"Mean Squared Error :{mse}\nRoot Mean Squared Error:{rmse}\nMean␣
 ↪Percentage Error:{mape}\nMean AbsoluteError:{mae}\n")
```

```
Mean Squared Error :77.75377358490564
Root Mean Squared Error:8.81781002204661
Mean Percentage Error:9.535691016912434
Mean AbsoluteError:7.305660377358488
```

0.0.17 Regression Line Plot

[46]:
```
plt.scatter(x_test,y_test)
plt.plot(x_test,y_pred_pearson)
```

[46]:
```
[<matplotlib.lines.Line2D at 0x1bee41189a0>]
```



0.0.18 Inference

Thus the equations obtained are the same and the Errors obtained are comparably similar in magnitude.

Result : Thus Multiple Points were simulated and Line of Best fit Found and Line of best fit found using least squares method and pearson coefficient

Ex. No: 2	Multiple Regression
17.01.2024	

Aim:

To Establish and Explain the linear relation of multiple features using Multiple regression

Algorithm:

1. Perform Preprocessing (Clear Null,Knn_impute,label encode,drop vif)
2. After Preprocessing Build the model using LinearRegression from sklearn
3. Find the model Accuracy Metric
4. Predict the Future Prices with the established model

Code+Output:

[2]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.impute import SimpleImputer as s
from sklearn.impute import KNNImputer as knn
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import mutual_info_regression
from sklearn.preprocessing import OrdinalEncoder
from sklearn.metrics import r2_score
from sklearn.preprocessing import StandardScaler
from statsmodels.stats.outliers_influence import variance_inflation_factor
data = pd.read_csv(r'D:/house_pred.csv')
data_pred=pd.read_csv(r'D:/test.csv')
```

[349]:
```
class PreProcess():
    def __init__(self,data):
        self.data=data
        self.run()
    def run(self):
        self.ClearNull(threshold=0.5)
        l=self.get_all_Null(dtype='float64')
        self.knn_impute(2,l)
        #self.outlier_remove()
        self.data=self.data.dropna()
        self.one_hot_encoding()
        self.StdScale()
        self.outlier_remove('SalePrice')
        self.drop_correlation()

        self.drop_vif(thresh=4.5)
    def drop_correlation(self):
        k=Utils_Suite(self.data).compute_correlation(0.3)
```

```
        f=pd.DataFrame(k)
        m=list(f[(f['SalePrice']<0.1) & (f['SalePrice']>-0.1)].index)
        self.data=self.data.drop(columns=m)

    def ClearNull(self,threshold):
        x=self.data.isna().sum()>0

        for i in  list(x.index):
            thresh=self.data[i].isna().sum()/len(self.data)
            if(x[i]==True and thresh>threshold):
                print(i,self.data[i].isna().sum())
                self.data=self.data.drop(i,axis=1)


    def knn_impute(self,n_neighbors,col_list):
        imputer=knn(n_neighbors=n_neighbors)
        for i in col_list:
            self.data[i]=imputer.fit_transform(self.data[[i]])[0][0]

    def arbitrary_remove(self):
        #data=data.drop(columns=['LotFrontage','MasVnrArea','GarageYrBlt'])
        self.data=self.data.drop('Id',axis=1)

    def get_all_Null(self,dtype=""):
        x=self.data.isna().sum()>0
        l=[]
        for i in  list(x.index):
            thresh=self.data[i].isna().sum()/len(self.data)
            if(x[i]==True and (data[i].dtypes==dtype) ):
                print(i,data[i].isna().sum())
                l+=[i]
        return l

    def outlier_remove(self,col):

        q1=self.data[col].quantile(0.25)
        q3=self.data[col].quantile(0.75)
        iqr=q3-q1
        l_whis=q1-1.5*iqr
        u_whis=q3+1.5*iqr
        self.data= self.data[(self.data[col]>=l_whis)& (self.data[col]<=u_whis)]

    #Depricated  ....
    def outlier_remove_deprecated(self):
        for col in self.data.columns:
            if self.data[col].dtypes!='object':
```

```
                q1=self.data[col].quantile(0.25)
                q3=self.data[col].quantile(0.75)
                iqr=q3-q1
                l_whis=q1-1.5*iqr
                u_whis=q3+1.5*iqr
                self.data= self.data[(self.data[col]>=l_whis)& (self.
↪data[col]<=u_whis)]
        return self.data

    def one_hot_encoding(self):
        z=(self.data.dtypes=='object')
        k=pd.DataFrame(z)
        obj_list=list(k[k[0]==True].index)
        print(obj_list)
        for i in obj_list:
            dummy=pd.get_dummies(self.data[i],prefix=i,drop_first=True)
            #print(dummy)
            self.data=self.data.drop(i,axis=1)
            self.data=self.data.join(dummy)
            #self.data=pd.concat([self.data,dummy],axis=1)


    def StdScale(self):
        for i in self.data.columns:
            if self.data[i].dtypes!='object' and i!='SalePrice':
                scale = StandardScaler().fit(self.data[[i]])

                self.data[i] = scale.transform(self.data[[i]])


    ## DANGER ZONE Col Spare NEEDED To Keep y_pred.
    def drop_vif(self,thresh=5,col_Spare=['SalePrice','intercept']):


        vif=Utils_Suite(self.data).compute_vif()
        z1=vif[vif["vif"]>thresh]
        z1=z1.sort_values(by='vif', kind='mergesort',ascending=[False])
        while True:
            try:
                col=z1.iloc[0,0]
                if z1.empty:
                    break
                if col in col_Spare:
                    z1=z1.iloc[1:]
                    continue
                self.data=self.data.drop(col,axis=1)
                vif=Utils_Suite(self.data).compute_vif()
```

```
                z1=vif[vif["vif"]>thresh]
                z1=z1.sort_values(by='vif', kind='mergesort',ascending=[False])
            except IndexError:
                break


    def write_df(self):
        return self.data
```

[351]:
```
class Utils_Suite():
    def __init__(self,data):
        self.data=data
    def compute_correlation(self,threshold=0.3):
        matrix=self.data.corr(numeric_only=True)
        ␣
↪x=matrix[(matrix["SalePrice"]<threshold)&(matrix["SalePrice"]>-threshold)]["SalePrice"]
        return x
    def compute_mutual_information(self,thresh=0.1):
        enc = OrdinalEncoder()
        df_encoded = enc.fit_transform(self.data)
        mi_scores = mutual_info_regression(df_encoded, self.data['SalePrice'])
        mi_scores_df = pd.DataFrame(mi_scores, index=self.data.columns,␣
↪columns=['Score'])
        return mi_scores_df[mi_scores_df['Score']<thresh]
    def compute_vif(self):
        x=self.data.iloc[:,:-1]
        y=self.data.iloc[:,-1]
        x=pd.DataFrame(x)

        x['intercept']=1
        vif=pd.DataFrame()
        vif['variable']=x.columns
        vif['vif']=[variance_inflation_factor(x.values,i)for i in range(x.
↪shape[1])]
        return vif
```

[352]:
```
class Model():
    def __init__(self,x_train,y_train,x_test,y_test):
        self.x_train=x_train
        self.x_test=x_test
        self.y_train=y_train
        self.y_test=y_test
```

```
        self.y_pred=0

    def fit(self):
            self.reg = LinearRegression()
            self.reg.fit(self.x_train,self.y_train)
            return self.reg
    def predict(self):
         self.y_pred=self.reg.predict(self.x_test)
         return self.y_pred
    def score_metric(self):
         return r2_score(self.y_test,self.y_pred)
```

[353]:
```
k=PreProcess(data=data)
```

```
Alley 1369
PoolQC 1453
Fence 1179
MiscFeature 1406
LotFrontage 259
MasVnrArea 8
GarageYrBlt 81
['MSZoning', 'Street', 'LotShape', 'LandContour', 'Utilities', 'LotConfig',
'LandSlope', 'Neighborhood', 'Condition1', 'Condition2', 'BldgType',
'HouseStyle', 'RoofStyle', 'RoofMatl', 'Exterior1st', 'Exterior2nd',
'MasVnrType', 'ExterQual', 'ExterCond', 'Foundation', 'BsmtQual', 'BsmtCond',
'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2', 'Heating', 'HeatingQC',
'CentralAir', 'Electrical', 'KitchenQual', 'Functional', 'FireplaceQu',
'GarageType', 'GarageFinish', 'GarageQual', 'GarageCond', 'PavedDrive',
'SaleType', 'SaleCondition']
```

[354]:
```
data=k.write_df()
```

[355]:
```
col=list(data.columns)
col.remove('SalePrice')
col.append('SalePrice')
data=data[col]
```

[368]:
```
x=data.iloc[:,:-1]
y=data.iloc[:,-1]
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.
↪3,random_state=2024)
```

[357]:
```
u=Utils_Suite(data).compute_vif()
```

```
e:\anaconda\lib\site-packages\statsmodels\regression\linear_model.py:1752:
RuntimeWarning: invalid value encountered in scalar divide
  return 1 - self.ssr/self.centered_tss
```

[358]:
```
u[u['vif']>4]
```

[358]:

	variable	vif
6	BsmtFinSF1	4.060141
78	SaleType_WD	4.073770
80	SaleCondition_Partial	4.841633

[369]:
```
MR_Model=Model(x_train,y_train,x_test,y_test)
reg=MR_Model.fit()
```

[370]:
```
reg.score(x,y)
```

[370]:
```
0.7909355723036731
```

[373]:
```
y_pred=MR_Model.predict()
```

[372]:
```
MR_Model.score_metric()
```

[372]:
```
0.689393821581118
```

[376]:
```
from sklearn.metrics import mean_squared_error
np.sqrt(mean_squared_error(y_test,y_pred))
```

[376]:
```
36857.56429672354
```

Results:

Thus the sale price was predicted using multiple regression using multiple features

Ex. No: 3	Logistic Regression
24.01.2024	

Aim:

To Classify and Predict Whether a customer churn using logistic regression

Algorithm:

1. Perform Preprocessing (vif,correlation removal,null value exclusion etc..)
2. Once the data is pre-processed split the data into train-test sets of own propotion
3. Using the sklearn LogisticRegression fit the model and predict
4. Print the confusion Matrix and accuracy score

Code + Output:

[207]:
```
import pandas as pd
import numpy as np
from sklearn import preprocessing
import matplotlib.pyplot as plt
from sklearn.impute import SimpleImputer as s
from sklearn.impute import KNNImputer as knn
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import mutual_info_regression
from sklearn.preprocessing import OrdinalEncoder
from sklearn.metrics import r2_score
from sklearn.preprocessing import StandardScaler
from statsmodels.stats.outliers_influence import variance_inflation_factor
```

[208]:
```
data=pd.read_csv("telecom_customer_churn.csv")
```

[209]:
```
data.head()
```

[209]:

	Customer ID	Gender	Age	Married	Number of Dependents	City	\
0	0002-ORFBO	Female	37	Yes	0	Frazier Park	
1	0003-MKNFE	Male	46	No	0	Glendale	
2	0004-TLHLJ	Male	50	No	0	Costa Mesa	
3	0011-IGKFF	Male	78	Yes	0	Martinez	
4	0013-EXCHZ	Female	75	Yes	0	Camarillo	

	Zip Code	Latitude	Longitude	Number of Referrals	…	Payment Method	\
0	93225	34.827662	-118.999073	2	…	Credit Card	
1	91206	34.162515	-118.203869	0	…	Credit Card	
2	92627	33.645672	-117.922613	0	…	Bank Withdrawal	
3	94553	38.014457	-122.115432	1	…	Bank Withdrawal	
4	93010	34.227846	-119.079903	3	…	Credit Card	

	Monthly Charge	Total Charges	Total Refunds	Total Extra Data Charges	\
0	65.6	593.30	0.00	0	
1	-4.0	542.40	38.33	10	
2	73.9	280.85	0.00	0	
3	98.0	1237.85	0.00	0	
4	83.9	267.40	0.00	0	

```
   Total Long Distance Charges Total Revenue  Customer Status   Churn Category  \
0                       381.51        974.81                1              NaN
1                        96.21        610.28                1              NaN
2                       134.60        415.45                0         Competitor
3                       361.66       1599.51                0  Dissatisfaction
4                        22.14        289.54                0  Dissatisfaction

                     Churn Reason
0                             NaN
1                             NaN
2  Competitor had better devices
3        Product dissatisfaction
4             Network reliability

[5 rows x 38 columns]
```

[210]:
```
class Utils_Suite():
    def __init__(self,data):
        self.data=data
    def compute_correlation(self,threshold=0.3):
        matrix=self.data.corr()
        x=matrix[(matrix["Customer Status"]<threshold)&(matrix["Customer␣
↪Status"]>-threshold)]["Customer Status"]
        return x
    def compute_mutual_information(self,thresh=0.1):
        enc = OrdinalEncoder()
        df_encoded = enc.fit_transform(self.data)
        mi_scores = mutual_info_regression(df_encoded, self.data['Customer␣
↪Status'])
        mi_scores_df = pd.DataFrame(mi_scores, index=self.data.columns,␣
↪columns=['Score'])
        return mi_scores_df[mi_scores_df['Score']<thresh]
    def compute_vif(self):
        x=self.data.iloc[:,:-1]
        y=self.data.iloc[:,-1]
        x=pd.DataFrame(x)

        x['intercept']=1
        vif=pd.DataFrame()
        vif['variable']=x.columns
        vif['vif']=[variance_inflation_factor(x.values,i)for i in range(x.
↪shape[1])]
        return vif
```

[211]:
```
data['Customer Status'].dtypes
```

[212]:
```
class PreProcess():
    #Auto Run Upon Initiation .
    def __init__(self,data):
        self.data=data
        self.run()
    # PreProcessing  Schedules
    def run(self):
        self.ClearNull(threshold=0.5)
        l=self.get_all_Null(dtype='float64')
        #self.knn_impute(2,l)
        #self.outlier_remove()
        #self.data=self.data.dropna()
        #self.one_hot_encoding()
        self.StdScale()
        #self.outlier_remove('Customer Status')


        self.drop_correlation()
        self.drop_uniq_thresh(thresh=5)
        self.Label_Encoding()
        self.drop_vif(thresh=3)

    # Remove Correlation
    def drop_correlation(self):
        k=Utils_Suite(self.data).compute_correlation(0.3)
        f=pd.DataFrame(k)
        m=list(f[(f['Customer Status']<0.1) & (f['Customer Status']>-0.1)].
↪index)
        self.data=self.data.drop(columns=m)

    def ClearNull(self,threshold):
        x=self.data.isna().sum()>0

        for i in  list(x.index):
            thresh=self.data[i].isna().sum()/len(self.data)
            if(x[i]==True and thresh>threshold):
                print(i,self.data[i].isna().sum())
                self.data=self.data.drop(i,axis=1)


    def knn_impute(self,n_neighbors,col_list):
        imputer=knn(n_neighbors=n_neighbors)
        for i in col_list:
            self.data[i]=imputer.fit_transform(self.data[[i]])[0][0]
```

```
def get_all_Null(self,dtype=""):
    x=self.data.isna().sum()>0
    l=[]
    for i in  list(x.index):
        thresh=self.data[i].isna().sum()/len(self.data)
        if(x[i]==True and (data[i].dtypes==dtype) ):
            print(i,data[i].isna().sum())
            l+=[i]
    return l
# Drop Outlier Rows
def outlier_remove(self,col):

    q1=self.data[col].quantile(0.25)
    q3=self.data[col].quantile(0.75)
    iqr=q3-q1
    l_whis=q1-1.5*iqr
    u_whis=q3+1.5*iqr
    self.data= self.data[(self.data[col]>=l_whis)& (self.data[col]<=u_whis)]

# One hot Encoding using get_dummies
def one_hot_encoding(self):
    z=(self.data.dtypes=='object')
    k=pd.DataFrame(z)
    obj_list=list(k[k[0]==True].index)
    print(obj_list)
    for i in obj_list:
        dummy=pd.get_dummies(self.data[i],prefix=i,drop_first=True)
        #print(dummy)
        self.data=self.data.drop(i,axis=1)
        self.data=self.data.join(dummy)
        #self.data=pd.concat([self.data,dummy],axis=1)
# standardize data
def drop_uniq_thresh(self,thresh=5):
    col=data.columns
    x=pd.DataFrame(self.data.dtypes)
    ll=list(x[x[0]=="object"].index)
    droplist=[]
    for i in ll:
        print(i)
        if (len(self.data[i].unique())>thresh):
            droplist+=[i]
    print(droplist)
    self.data=self.data.drop(columns=droplist)


def Label_Encoding(self):
```

```
        label_encoder = preprocessing.LabelEncoder()
        x=pd.DataFrame(self.data.dtypes)
        ll=list(x[x[0]=="object"].index)
        for i in ll:
            self.data[i]= label_encoder.fit_transform(self.data[i])

    def StdScale(self):
        for i in self.data.columns:
            if self.data[i].dtypes!='object' and i!='Customer Status':
                scale = StandardScaler().fit(self.data[[i]])

                self.data[i] = scale.transform(self.data[[i]])


    ## DANGER ZONE Col Spare NEEDED To Keep y_pred. RAM HOGGING FUNCTION .
    #Use Wisely! Plus Parallize the operation for better efficacy? Maybe???

    def drop_vif(self,thresh=5,col_Spare=['Customer Status','intercept']):


        vif=Utils_Suite(self.data).compute_vif()
        z1=vif[vif["vif"]>thresh]
        z1=z1.sort_values(by='vif', kind='mergesort',ascending=[False])
        while True:
            try:
                col=z1.iloc[0,0]
                if z1.empty:
                    break
                if col in col_Spare:
                    z1=z1.iloc[1:]
                    continue
                self.data=self.data.drop(col,axis=1)
                vif=Utils_Suite(self.data).compute_vif()
                z1=vif[vif["vif"]>thresh]
                z1=z1.sort_values(by='vif', kind='mergesort',ascending=[False])
            except IndexError:
                break

    # Wrapper Function to dump data to a variable
    def write_df(self):
        return self.data
```

13]:
```
x=PreProcess(data)
```

```
Churn Category 4720
Churn Reason 4720
Avg Monthly Long Distance Charges 644
```

```
      Unlimited Data  Contract  Paperless Billing  Payment Method  \
0                  1         1                  1               1
1                  0         0                  0               1
2                  1         0                  1               0
3                  1         0                  1               0
4                  1         0                  1               1
…                …         …                  …             …
6584               2         0                  0               0
6585               1         1                  0               1
6586               1         0                  1               0
6587               1         2                  0               1
6588               1         2                  0               0

      Monthly Charge  Total Long Distance Charges  Customer Status
0           0.018307                    -0.487965                1
1          -2.219753                    -0.822156                1
2           0.285202                    -0.777187                0
3           1.060162                    -0.511217                0
4           0.606762                    -0.908919                0
…             …                           …             …
6584       -1.417460                    -0.925436                0
6585       -0.317724                    -0.224021                1
6586        0.645349                    -0.517378                0
6587        0.090658                    -0.768472                1
6588       -0.193923                    -0.934853                1

[6589 rows x 16 columns]
```

[198]:
```
data=x.write_df()
```

[199]:
```
data.columns
```

[199]:
```
Index(['Gender', 'Age', 'Married', 'Number of Dependents',
       'Number of Referrals', 'Multiple Lines', 'Internet Type',
       'Online Backup', 'Streaming TV', 'Unlimited Data', 'Contract',
       'Paperless Billing', 'Payment Method', 'Monthly Charge',
       'Total Long Distance Charges', 'Customer Status'],
      dtype='object')
```

[200]:
```
from sklearn.model_selection import train_test_split
```

[201]:
```
x=data.iloc[:,:-1].values
y=data.iloc[:,-1].values
```

```
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.3,random_state=0)
```

[202]:
```
from sklearn.linear_model import LogisticRegression
reg=LogisticRegression()
reg.fit(x_train,y_train)
```

[202]:
```
LogisticRegression()
```

[203]:
```
y_pred=reg.predict(x_test)
(y_test,y_pred)
from sklearn.metrics import
 ↪confusion_matrix,accuracy_score,classification_report

confusion_matrix(y_test,y_pred)
```

[203]:
```
array([[ 384,  169],
       [ 200, 1224]])
```

[204]:
```
accuracy_score(y_test,y_pred)
```

[204]:
```
0.8133535660091047
```

Result:

Thus Logistic Regression was used to predict customer churn and obtained an accuracy score of 81%

Ex. No: 4	Decision Tree
31.01.2024	

Aim:

To Compute Gini Score and create decision Tree From Scratch for a discrete data.

Algorithm:

1. For Each Columns encode the data and get the gini score
2. Compute the split points into array
3. Get all the leaf nodes
4. Construct the decision tree with pruned depth

Code + Output:

0.1 Version Changelog

1.0.0 -> Computed Part a) of Question for Gini Impurity for continuous Columns
Yet to comment and build tree

[2]:
```
import pandas as pd
import numpy as np
```

[3]:
```
class compute_gini_impurity:
    def __init__(self,data,column:str,target:str):
        self.data=data
        self.column=column
        self.target=target
        self.data=self.data.sort_values(ascending=True,by=column)
        self.idx=self.data.index
        self.l=list(data[target].unique())
        self._update_map()
        self.get_split()
        self.split_points()
        self.create_avg_split_continuous()
    def _update_map(self):
        self._map={}
        for i in range(len(self.data)):
            self._map.update({ i:self.data.index.get_loc(i) })
    def get_split(self):
        flag=self.l[self.data[self.target].iloc[self._map[self.idx[0]]]]
        self.arr=[]
        for i in self.idx:
            if (self.data[self.target].iloc[self._map[i]]==self.l[(~flag)]):
                flag=~flag
                self.arr.append(i)
        #return arr
    def split_points(self):
```

```
        self.split_points=[]
        for i in range(len(self.arr)):
            self.split_points.append(self._map[self.arr[i]]-1)
    def create_avg_split_continuous(self):
        #avf=pd.DataFrame(s)
        l2=[]
        for i in range(len(self.split_points)):
            l2.append((self.data[self.column].iloc[self.split_points[i]]+self.
↪data[self.column].iloc[self.split_points[i]+1])/2)
        s=pd.Series(l2)
        self.avf=pd.DataFrame(s)
        self.avf=self.avf.rename({0:"avg"},axis=1)
    def compute_gini_leaf(self):

        gini_score=[]
        gini_dict={}
        gs,ix=np.inf,0

        for i in range(len(self.avf)):

            no1=len(self.data[(self.data[self.column]<self.avf.iloc[i][0]␣
↪)&(self.data[self.target]==0)])
            yes1=len(self.data[(self.data[self.column]<self.avf.iloc[i][0]␣
↪)&(self.data[self.target]==1)])
            no=len(self.data[(self.data[self.column]>=self.avf.iloc[i][0]␣
↪)&(self.data[self.target]==0)])
            yes=len(self.data[(self.data[self.column]>=self.avf.iloc[i][0]␣
↪)&(self.data[self.target]==1)])
            p1=no1/(no1+yes1)
            p2=no/(no+yes)
            gini1=1-(1-p1)**2-p1**2
            gini2=1-(1-p2)**2-p2**2
            total=((no1+yes1)/(no1+yes1+no+yes))*gini1+((no+yes)/
↪(no1+yes1+no+yes))*gini2
            #gini_score.append(total)
            if(gs>total):
                gs=total
                ix=self.avf.iloc[i][0]
                #index=self.data.in
            gini_dict.update({self.avf.iloc[i][0]:total})

        return (ix,gs,gini_dict) #score,gini_dict_split,split_idx,split_feature,
```

[4]:
```
data=pd.read_csv(r"D:/classification.csv")
```

0.1.1 GINI Impurity for 'Age' Column

[5]:
```
s=compute_gini_impurity(data,'Age','Purchased')
```

[6]:
```
s1,ix,d=s.compute_gini_leaf()
```

[7]:
```
s1,ix
```

[7]:
```
(42.5, 0.268790236460717)
```

0.1.2 Gini Impurity for 'EstimatedSalary' Column

[8]:
```
s=compute_gini_impurity(data,'EstimatedSalary','Purchased')
s2,ix2,d2=s.compute_gini_leaf()
s2,ix2
```

[8]:
```
(89500.0, 0.32756555944055943)
```

0.2 Decision Tree From Scratch

[122]:
```
data_cat=pd.read_csv(r"D://toydata.csv")
```

[123]:
```
data_cat
```

[123]:

	Outlook	Temperature	Humidity	Windy	Play
0	Sunny	Hot	High	No	No
1	Sunny	Hot	High	Yes	No
2	Overcast	Hot	High	No	Yes
3	Rainy	Mild	High	No	Yes
4	Rainy	Cool	Normal	No	Yes
5	Rainy	Cool	Normal	Yes	No
6	Overcast	Cool	Normal	Yes	Yes
7	Sunny	Mild	High	No	No
8	Sunny	Cool	Normal	No	Yes
9	Rainy	Mild	Normal	No	Yes
10	Sunny	Mild	Normal	Yes	Yes
11	Overcast	Mild	High	Yes	Yes
12	Overcast	Hot	Normal	No	Yes
13	Rainy	Mild	High	Yes	No

[124]:
```
from category_encoders import OrdinalEncoder
maplist = [
    {'col': 'Outlook', 'mapping': {'Sunny': 0, 'Overcast': 1,'Rainy': 2}},
    {
        'col':'Temperature','mapping':{'Hot':0,'Mild':1,'Cool':2}

    },
    {
```

```
        'col':'Humidity','mapping':{'Normal':0,'High':1}

    },
    {
        'col':'Windy','mapping':{"No":0,"Yes":1}
    },
    {
        'col':'Play','mapping':{"No":0,"Yes":1}
    }




    ]
enc=OrdinalEncoder(mapping=maplist)

#data_cat.columns
data_cat[data_cat.columns] = enc.fit_transform(data_cat[data_cat.columns])
```

[141]:
```
data_cat
```

[141]:

	Outlook	Temperature	Humidity	Windy	Play
0	0	0	1	0	0
1	0	0	1	1	0
2	1	0	1	0	1
3	2	1	1	0	1
4	2	2	0	0	1
5	2	2	0	1	0
6	1	2	0	1	1
7	0	1	1	0	0
8	0	2	0	0	1
9	2	1	0	0	1
10	0	1	0	1	1
11	1	1	1	1	1
12	1	0	0	0	1
13	2	1	1	1	0

[12]:
```
data_cat['Outlook'].unique()
```

[12]:
```
array([0, 1, 2])
```

[ ]:

[ ]:

[13]:
```
data_cat
```

[13]:

	Outlook	Temperature	Humidity	Windy	Play
0	0	0	1	0	0
1	0	0	1	1	0
2	1	0	1	0	1
3	2	1	1	0	1
4	2	2	0	0	1
5	2	2	0	1	0
6	1	2	0	1	1
7	0	1	1	0	0
8	0	2	0	0	1
9	2	1	0	0	1
10	0	1	0	1	1
11	1	1	1	1	1
12	1	0	0	0	1
13	2	1	1	1	0

[14]:
```
class gini_cat:
    def __init__(self,data,column,target):
        self.data=data
        self.column=column
        self.target=target
        self.col_val=data[column].unique()
        self.target_val=data[target].unique()
    def compute(self):
        gini_dict={}
        gs=np.inf
        val=None
        for i in self.col_val:
            subset_data=[self.data[self.data[self.column]==i],self.data[self.
 ↪data[self.column]!=i]]
            gini_score=0

            for j in subset_data:
                size=len(j)
                if size==0:
                    continue
                score=0
                for k in self.target_val:
                    p=len(j[j[self.target]==k])/size
                    score+=p**2
                gini_score+=(1-score)*(size)/len(self.data)


            if (gini_score<gs):
                gs=gini_score
                val=i
            gini_dict[i]=gini_score
```

```
        return (gs,val,gini_dict)
```

[15]:
```
gini_cat(data_cat,'Outlook','Play').compute()
```

[15]:
```
(0.35714285714285715,
 1,
 {0: 0.3936507936507936, 1: 0.35714285714285715, 2: 0.45714285714285713})
```

[139]:
```
d1=data_cat[data_cat['Outlook']==0]
```

[32]:
```
d1
```

[32]:

	Outlook	Temperature	Humidity	Windy	Play
2	1	0	1	0	1
6	1	2	0	1	1
11	1	1	1	1	1
12	1	0	0	0	1

[19]:
```
class DecisionTreeClassifier:
    def __init__(self,data):
        pass
```

[126]:
```
def get_best_split(data,column):
    n_features=data[column].unique()
    #features={}
    _,_,features=gini_cat(data,column=column,target="Play").compute()
    return (min(features.items(), key=lambda x: x[1]) ,dict(sorted(features.
 ↪items(), key=lambda item: item[1])))

(get_best_split(data_cat,"Outlook"))
```

[126]:
```
((1, 0.35714285714285715),
 {1: 0.35714285714285715, 0: 0.3936507936507936, 2: 0.45714285714285713})
```

[23]:
```
def subset(data,column,feature):
    return data[data[column]==feature]
```

[275]:
```
class Node:
    def __init__(self,column,idx,data_subset,gini_score):
        self.gini_score=gini_score
        self.data=data_subset
        self.column=column
        self.idx=idx
        self.children=[]
```

```
def add_child(parent,node):
    new_node= node
    parent.children.append(new_node)
    return new_node
def print_tree(node, level=0):
    print('  ' * level + str(node.column))
    for child in node.children:
        print_tree(child, level + 1)
'''
root=Node("outlook",data_cat,gini_cat(data_cat,'Outlook','Play').compute()[0])
child1 = add_child(root,␣
 ↪'Sunny',data_cat[data_cat["Outlook"]==0],gini_cat(data_cat[data_cat["Outlook"]==0],'Outlooi
 ↪compute()[0])
child2 = add_child(root,␣
 ↪'Overcast',data_cat[data_cat["Outlook"]==1],gini_cat(data_cat[data_cat["Outlook"]==1],'Out
 ↪compute()[0])
child3 = add_child(root,␣
 ↪'Rainy',data_cat[data_cat["Outlook"]==2],gini_cat(data_cat[data_cat["Outlook"]==2],'Outlooi
 ↪compute()[0])
next=add_child(child1,'Humidity',data_cat[data_cat["Humidity"]==0],gini_cat(data_cat[data_cat
 ↪compute()[0])

print(root.children[0].children[0].column)
print_tree(root)
'''
```

[275]:
```
'\nroot=Node("outlook",data_cat,gini_cat(data_cat,\'Outlook\',\'Play\').compute(
)[0])\nchild1 = add_child(root, \'Sunny\',data_cat[data_cat["Outlook"]==0],gini_
cat(data_cat[data_cat["Outlook"]==0],\'Outlook\',\'Play\').compute()[0])\nchild2
= add_child(root, \'Overcast\',data_cat[data_cat["Outlook"]==1],gini_cat(data_ca
t[data_cat["Outlook"]==1],\'Outlook\',\'Play\').compute()[0])\nchild3 =
add_child(root, \'Rainy\',data_cat[data_cat["Outlook"]==2],gini_cat(data_cat[dat
a_cat["Outlook"]==2],\'Outlook\',\'Play\').compute()[0])\nnext=add_child(child1,
\'Humidity\',data_cat[data_cat["Humidity"]==0],gini_cat(data_cat[data_cat["Humid
ity"]==0],\'Outlook\',\'Play\').compute()[0])\n\nprint(root.children[0].children
[0].column)\nprint_tree(root)\n'
```

[260]:
```
def build(data,features_col):
    bf,gs,best_col=None,np.inf,None
    for i in features_col:
        if(get_best_split(data,i)[0][-1]<gs):
            gs=get_best_split(data,i)[0][-1]
            bf=get_best_split(data,i)[0]
            best_col=i
    if gs==0:
        return best_col
    else:
```

```
        return best_col
col=build(d1,set(d1.columns)-{"Play"})
col
```

[260]: 'Humidity'

[206]:
```
root.column
```

[206]: 'Outlook'

[210]:
```
get_best_split(data_cat[data_cat['Outlook']==2],'Humidity')
```

[210]: ((1, 0.4666666666666667), {1: 0.4666666666666667, 0: 0.4666666666666667})

[276]:
```
def build_decision(node, features):
    col = build(node.data, features_col=features)
    print(col)
    used =[col,"Play"]
    if col != "":
        j, d = get_best_split(node.data, col)
        #j=j[0]
        while len(d)!=0:
            j= min(d.items(), key=lambda x: x[1])[0]
            print("ji:",j)
            gs = min(d.items(), key=lambda x: x[1])[1]
            print(gs)


            nn = Node(col,j, subset(node.data, col, j), gs)

            print(f"Adding child with column {j} and gini score {gs}")
            node.children.append(nn)
            print(node.children)
            print(d)

            d.pop(j)
    return node

def add_child(parent, child):
    print(f"Adding child {child.column} to parent {parent.column}")
    parent.children.append(child)
```

[240]:
```
data_cat[data_cat['Outlook']==2],
```

[240]: (
	Outlook	Temperature	Humidity	Windy	Play
3	2	1	1	0	1
4	2	2	0	0	1
5	2	2	0	1	0

[241]:
```
subset(data_cat, 'Outlook', 2)
```

[241]:

	Outlook	Temperature	Humidity	Windy	Play
3	2	1	1	0	1
4	2	2	0	0	1
5	2	2	0	1	0
9	2	1	0	0	1
13	2	1	1	1	0

[237]:
```
gini_cat(data_cat[data_cat['Outlook']==2],"Humidity","Play").compute()
```

[237]:
```
(0.4666666666666667, 1, {1: 0.4666666666666667, 0: 0.4666666666666667})
```

[263]:
```
build(d1, features_col=set(subset(data_cat, 'Outlook', 1).columns)-{"Play"})
```

[263]:
```
'Humidity'
```

[258]:
```
d1=subset(data_cat, 'Outlook',1)
```

[277]:
```
root=Node('Outlook',None,data_cat,gini_cat(data_cat,'Outlook','Play').
↪compute()[0])
```

[110]:
```
root.data.columns.tolist()
```

[110]:
```
['Outlook', 'Temperature', 'Humidity', 'Wind', 'Play']
```

[278]:
```
build_decision(root, set(root.data.columns.tolist())-{'Play'})
```

```
Outlook
ji: 1
0.35714285714285715
Adding child with column 1 and gini score 0.35714285714285715
[<__main__.Node object at 0x000001B4E1CBB0D0>]
{1: 0.35714285714285715, 0: 0.3936507936507936, 2: 0.45714285714285713}
ji: 0
0.3936507936507936
Adding child with column 0 and gini score 0.3936507936507936
[<__main__.Node object at 0x000001B4E1CBB0D0>, <__main__.Node object at
0x000001B4E2E3D820>]
{0: 0.3936507936507936, 2: 0.45714285714285713}
ji: 2
0.45714285714285713
Adding child with column 2 and gini score 0.45714285714285713
[<__main__.Node object at 0x000001B4E1CBB0D0>, <__main__.Node object at
0x000001B4E2E3D820>, <__main__.Node object at 0x000001B4E2EAFCA0>]
{2: 0.45714285714285713}
```

[278]:
```
<__main__.Node at 0x1b4e2ec52e0>
```

[279]:
```
build(data_cat[data_cat['Outlook']==2],data_cat.columns.tolist())
```

[279]:
```
'Windy'
```

[202]:
```
build_decision(root.children[0], set(root.children[0].data.columns.
↪tolist())-{'Play'})
```

[202]:
```
<__main__.Node at 0x1b4e2ae61c0>
```

[ ]:
```
for i in root.children:
    next=i
    build(data_cat[data_cat['Outlook']==2],data_cat.columns.tolist())
    Node()
    add_child(i,)
```

[290]:
```
gini_cat(data_cat[(data_cat['Outlook']==1)&␣
↪(data_cat['Play']==0)],"Outlook","Play").compute()[0]
```

[290]:
```
inf
```

[248]:
```
def build_tree(node, features):
    if node.gini_score == 0.0:
        return node
    else:
        if not node.children:
            node = build_decision(node, features)  # update the node with the␣
↪returned node from build_decision
        for child in node.children:
            (build_tree(child, set(node.data.columns.tolist())-{'Play'}))
    return node

tree = build_decision(root, set(root.data.columns.tolist())-{'Play'})
print_tree(tree)
#print(tree.column)
print(tree.children)
```

```
Outlook
ji: 1
0.35714285714285715
hi:
Adding child with column 1 and gini score 0.35714285714285715
[<__main__.Node object at 0x000001B4E2E783A0>, <__main__.Node object at
0x000001B4E2E78550>, <__main__.Node object at 0x000001B4E2E7EFA0>,
<__main__.Node object at 0x000001B4E2AF3910>, <__main__.Node object at
0x000001B4E2AE7790>, <__main__.Node object at 0x000001B4E2E3D580>,
<__main__.Node object at 0x000001B4E2E99490>, <__main__.Node object at
```


Result:

Thus Decision tree was constructed using gini impurity

Ex. No: 5	Customer Churn Prediction
03.02.2024	

Aim:

To classify customer churn using the teleco dataset using KNN,Naïve Bayes,SVM ,Logistic Regression, DTree

Algorithm:

1.Preprocess the dataset using suitable methods
2. Once it is done Split it into train and test data
3. Apply Logistic Regression ,Decision Tree, KNN, NB,SVM and compute the accuracy
4. Choose the best model

Code + Output:

[67]:
```
from sklearn import preprocessing
import matplotlib.pyplot as plt
from sklearn.impute import SimpleImputer as s
from sklearn.impute import KNNImputer as knn
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import mutual_info_regression
from sklearn.preprocessing import OrdinalEncoder
from sklearn.metrics import r2_score
from sklearn.preprocessing import StandardScaler
from statsmodels.stats.outliers_influence import variance_inflation_factor
```

[68]:
```
import pandas as pd
import numpy as np
```

[69]:
```
data=pd.read_csv(r"D:\\Telco-Customer-Churn.csv")
```

[70]:
```
class Utils_Suite():
    def __init__(self,data):
        self.data=data
    def compute_correlation(self,threshold=0.3):
        matrix=self.data.corr(numeric_only=False)
        
↪x=matrix[(matrix["Churn"]<threshold)&(matrix["Churn"]>-threshold)]["Churn"]
        return x
    def compute_mutual_information(self,thresh=0.1):
        enc = OrdinalEncoder()
        df_encoded = enc.fit_transform(self.data)
        mi_scores = mutual_info_regression(df_encoded, self.data['Churn'])
        mi_scores_df = pd.DataFrame(mi_scores, index=self.data.columns,␣
↪columns=['Score'])
        return mi_scores_df[mi_scores_df['Score']<thresh]
    def compute_vif(self):
        x=self.data.iloc[:,:-1]
        y=self.data.iloc[:,-1]
        x=pd.DataFrame(x)

        x['intercept']=1
```

```
        vif=pd.DataFrame()
        vif['variable']=x.columns
        vif['vif']=[variance_inflation_factor(x.values,i)for i in range(x.
↪shape[1])]
        return vif
```

[71]:
```
class PreProcess():
    #Auto Run Upon Initiation .
    def __init__(self,data):
        self.data=data

        self.run()
    # PreProcessing  Schedules
    def run(self):
        self.ClearNull(threshold=0.5)
        self.data=self.data.drop(data[data['TotalCharges'] == ' '].index)
        self.data['TotalCharges']=self.data['TotalCharges'].astype(float)
        l=self.get_all_Null(dtype='float64')

        self.outlier_remove('TotalCharges')
        self.outlier_remove('MonthlyCharges')

        self.StdScale()

        #self.drop_uniq_thresh(thresh=5)

        #self.Label_Encoding()
        self.one_hot_encoding()
        #self.drop_correlation()
        #self.drop_vif(thresh=4)

    # Remove Correlation
    def drop_correlation(self):
        k=Utils_Suite(self.data).compute_correlation(0.3)
        f=pd.DataFrame(k)
        m=list(f[(f['Churn']<0.15) & (f['Churn']>-0.15)].index)
        self.data=self.data.drop(columns=m)

    def ClearNull(self,threshold):
        x=self.data.isna().sum()>0

        for i in  list(x.index):
            thresh=self.data[i].isna().sum()/len(self.data)
            if(x[i]==True and thresh>threshold):
                print(i,self.data[i].isna().sum())
                self.data=self.data.drop(i,axis=1)
```

```
def knn_impute(self,n_neighbors,col_list):
    imputer=knn(n_neighbors=n_neighbors)
    for i in col_list:
        self.data[i]=imputer.fit_transform(self.data[[i]])[0][0]


def get_all_Null(self,dtype=""):
    x=self.data.isna().sum()>0
    l=[]
    for i in  list(x.index):
        thresh=self.data[i].isna().sum()/len(self.data)
        if(x[i]==True and (data[i].dtypes==dtype) ):
            print(i,data[i].isna().sum())
            l+=[i]
    return l
# Drop Outlier Rows
def outlier_remove(self,col):

    q1=self.data[col].quantile(0.25)
    q3=self.data[col].quantile(0.75)
    iqr=q3-q1
    l_whis=q1-1.5*iqr
    u_whis=q3+1.5*iqr
    self.data= self.data[(self.data[col]>=l_whis)& (self.data[col]<=u_whis)]

# One hot Encoding using get_dummies
def one_hot_encoding(self):
    z=(self.data.dtypes=='object')
    k=pd.DataFrame(z)
    obj_list=list(k[k[0]==True].index)
    print(obj_list)
    for i in obj_list:
        dummy=pd.get_dummies(self.data[i],prefix=i,drop_first=True)

        self.data=self.data.drop(i,axis=1)
        self.data=self.data.join(dummy)

# standardize data
def drop_uniq_thresh(self,thresh=5):
    col=data.columns
    x=pd.DataFrame(self.data.dtypes)
    ll=list(x[x[0]=="object"].index)
    droplist=[]
    for i in ll:
        print(i)
```

```
            if (len(self.data[i].unique())>thresh):
                droplist+=[i]
        print(droplist)
        self.data=self.data.drop(columns=droplist)


    def Label_Encoding(self):
        label_encoder = preprocessing.LabelEncoder()
        x=pd.DataFrame(self.data.dtypes)
        ll=list(x[x[0]=="object"].index)
        for i in ll:
            self.data[i]= label_encoder.fit_transform(self.data[i])

    def StdScale(self):
        for i in self.data.columns:
            if self.data[i].dtypes!='object' and i!='Churn':
                scale = StandardScaler().fit(self.data[[i]])

                self.data[i] = scale.transform(self.data[[i]])


    ## DANGER ZONE Col Spare NEEDED To Keep y_pred. RAM HOGGING FUNCTION .
    #Use Wisely! Plus Parallize the operation for better efficacy? Maybe???

    def drop_vif(self,thresh=5,col_Spare=['Churn','intercept']):


        vif=Utils_Suite(self.data).compute_vif()
        z1=vif[vif["vif"]>thresh]
        z1=z1.sort_values(by='vif', kind='mergesort',ascending=[False])
        while True:
            try:
                col=z1.iloc[0,0]
                if z1.empty:
                    break
                if col in col_Spare:
                    z1=z1.iloc[1:]
                    continue
                self.data=self.data.drop(col,axis=1)
                vif=Utils_Suite(self.data).compute_vif()
                z1=vif[vif["vif"]>thresh]
                z1=z1.sort_values(by='vif', kind='mergesort',ascending=[False])
            except IndexError:
                break

    # Wrapper Function to dump data to a variable
    def write_df(self):
```

[ ]:

[66]:
```
data.dtypes
```

[66]:
```
customerID                                object
gender                                    object
SeniorCitizen                            float64
Partner                                   object
Dependents                                object
                                           ...
PaperlessBilling_Yes                       uint8
PaymentMethod_Credit card (automatic)      uint8
PaymentMethod_Electronic check             uint8
PaymentMethod_Mailed check                 uint8
Churn_Yes                                  uint8
Length: 7079, dtype: object
```

[72]:
```
data=data.drop(data[data['TotalCharges'] == ' '].index)
```

[73]:
```
data['TotalCharges']=data['TotalCharges'].astype(float)
```

[74]:
```
x=PreProcess(data)
data=x.write_df()
```

```
['customerID', 'gender', 'Partner', 'Dependents', 'PhoneService',
'MultipleLines', 'InternetService', 'OnlineSecurity', 'OnlineBackup',
'DeviceProtection', 'TechSupport', 'StreamingTV', 'StreamingMovies', 'Contract',
'PaperlessBilling', 'PaymentMethod', 'Churn']
```

[41]:
```
data=data.drop('Churn_Yes',axis=1)
```

[38]:
```
dummy=pd.get_dummies(data['Churn'],prefix='Churn')
dummy
```

[38]:

	Churn_No	Churn_Yes
0	1	0
1	1	0
2	0	1
3	1	0
4	0	1
…	…	…
7038	1	0
7039	1	0
7040	1	0
7041	0	1

[78]:
```
x=data.iloc[:,:-1].values
y=data.iloc[:,-1].values
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.
↪3,random_state=2024)
```

0.1 Using Logistic Regression

[83]:
```
from sklearn.linear_model import LogisticRegression
reg=LogisticRegression(solver='liblinear')
reg.fit(x_train,y_train)
```

[83]:
```
LogisticRegression(solver='liblinear')
```

[84]:
```
y_pred=reg.predict(x_test)
(y_test,y_pred)
from sklearn.metrics import␣
↪confusion_matrix,accuracy_score,classification_report

conf=confusion_matrix(y_test,y_pred)
```

[85]:
```
accuracy_score(y_test,y_pred)
```

[85]:
```
1.0
```

0.2 Using Decision Tree Classifier

[87]:
```
from sklearn.tree import DecisionTreeClassifier
tree=DecisionTreeClassifier(criterion="entropy",max_depth=4)
tree.fit(x_train,y_train)
y_pred=tree.predict(x_test)
(y_test,y_pred)

confusion_matrix(y_test,y_pred)
accuracy_score(y_test,y_pred)
```

[87]:
```
1.0
```

[494]:
```
print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

           0       0.80      0.93      0.86      1552
           1       0.66      0.37      0.47       558

    accuracy                           0.78      2110
   macro avg       0.73      0.65      0.67      2110
weighted avg       0.77      0.78      0.76      2110
```

0.3 Using Naive Bayes

[495]:
```
from sklearn.naive_bayes  import GaussianNB
reg=GaussianNB()
reg.fit(x_train,y_train)
y_pred=reg.predict(x_test)
(y_test,y_pred)

confusion_matrix(y_test,y_pred)
```

[495]:
```
array([[1183,  369],
       [ 160,  398]], dtype=int64)
```

[496]:
```
accuracy_score(y_test,y_pred)
```

[496]:
```
0.7492890995260664
```

[497]:
```
print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

           0       0.88      0.76      0.82      1552
           1       0.52      0.71      0.60       558

    accuracy                           0.75      2110
   macro avg       0.70      0.74      0.71      2110
weighted avg       0.79      0.75      0.76      2110
```

0.4 Using KNN Classifier

[498]:
```
from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier(n_neighbors=5)
classifier.fit(x_train, y_train)
```

[498]:
```
KNeighborsClassifier()
```

[499]:
```
y_pred = classifier.predict(x_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
[[1304  248]
 [ 288  270]]
              precision    recall  f1-score   support

           0       0.82      0.84      0.83      1552
           1       0.52      0.48      0.50       558

    accuracy                           0.75      2110
   macro avg       0.67      0.66      0.67      2110
weighted avg       0.74      0.75      0.74      2110
```

[500]:
```
accuracy_score(y_test,y_pred)
```

[500]:
```
0.7459715639810427
```

0.5 Using Support Vector Classifier

[23]:
```
from sklearn import svm

#Create a svm Classifier
clf = svm.SVC(kernel='poly') # Linear Kernel

#Train the model using the training sets
clf.fit(x_train, y_train)

#Predict the response for test dataset
y_pred = clf.predict(x_test)
accuracy_score(y_test, y_pred)
```

[23]:
```
0.8023696682464455
```

[502]:
```
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
[[1371  181]
 [ 267  291]]
              precision    recall  f1-score   support

           0       0.84      0.88      0.86      1552
           1       0.62      0.52      0.57       558

    accuracy                           0.79      2110
   macro avg       0.73      0.70      0.71      2110
```

```
weighted avg       0.78      0.79      0.78      2110
```

0.5.1 Inference

Logistic Regression :

```
Accuracy: 80%
```

Naive Bayes:

```
Accuracy: 74.9%
```

SVM :

```
Accuracy: 78.7%
```

Decision Tree Classifier :

```
Accuracy: 78.2%
```

KNN Classifier :

```
Accuracy: 74.5%
```

0.5.2 Interpretations

From All of the Above we see that Logistic Regression Has performed the best ~ 80% Accuracy And KNN Classifier the worst ~74% With all other algorithms being close to each other in terms of accuracy

Decision Tree is not the most accurate because it could have overfitted the data while training and due to this we see a dip in accuracy score and are biased towards branches with more levels as here in this case

While KNN Does not perform well in higher dimensions here we have almost 21 features

On the other hand

Naive bayes does not perform well with continuous data [Total Salary]and may be the accuracy would be low because of existence of small correlation

SVM Needs Proper Hyper parameter tuning and selection of kernel … here i chose linear .. which was not the case here so it performs poorly SVM Linear ~78% SVM Poly Kernel ~ 81 %

The best performing model Logistic regression has the comparitively high accuracy of 80% might be because of specificity of classification wrt churn and provides a probablistic likelihood of churning and learns the features ….

One Interesting observation with respect to overfitting is that even when only one partof the prediction set is revealed say (Churn_Yes) the Model Learns this specific feature and clearly overfits all data to this with a accuracy of 100% which is in no way accurate.

Result: Thus Multiple Classifiers were fit and Logistic regression performed better compared to others

Ex. No: 6	Hyper Parameter Tuning
28.02.2024	

Aim:

To Perform Hyper Parameter tuning using GridSearch and Randomised Search Strategy.

Algorithm:

1.Perform Necessary Pre-processing
2.Apply Multiple Algorithms on the dataset
3. After that Select some parameters and apply gridsearch and randomized search
4. Compute the accuracy after selection of parameters to train on given data

Code + Output:

0.0.1 Hyper Parameter Tuning

[19]:
```
import pandas as pd
import numpy as np
from sklearn import preprocessing
import matplotlib.pyplot as plt
from sklearn.impute import SimpleImputer as s
from sklearn.impute import KNNImputer as knn
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import mutual_info_regression
from sklearn.preprocessing import OrdinalEncoder
from sklearn.metrics import r2_score
from sklearn.preprocessing import StandardScaler
from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
```

[3]:
```
data=pd.read_csv(r"D:\\tcc.csv")
```

[4]:
```
class Utils_Suite():
    def __init__(self,data):
        self.data=data
    def compute_correlation(self,threshold=0.3):
        matrix=self.data.corr(numeric_only=False)
        
 ↪x=matrix[(matrix["Churn"]<threshold)&(matrix["Churn"]>-threshold)]["Churn"]
        return x
    def compute_mutual_information(self,thresh=0.1):
        enc = OrdinalEncoder()
        df_encoded = enc.fit_transform(self.data)
        mi_scores = mutual_info_regression(df_encoded, self.data['Churn'])
        mi_scores_df = pd.DataFrame(mi_scores, index=self.data.columns,
 ↪columns=['Score'])
```

```
        return mi_scores_df[mi_scores_df['Score']<thresh]
    def compute_vif(self):
        x=self.data.iloc[:,:-1]
        y=self.data.iloc[:,-1]
        x=pd.DataFrame(x)

        x['intercept']=1
        vif=pd.DataFrame()
        vif['variable']=x.columns
        vif['vif']=[variance_inflation_factor(x.values,i)for i in range(x.
↪shape[1])]
        return vif
```

[5]:
```
class PreProcess():
    #Auto Run Upon Initiation .
    def __init__(self,data):
        self.data=data

        self.run()
    # PreProcessing  Schedules
    def run(self):
        self.ClearNull(threshold=0.5)
        self.data=self.data.drop(data[data['TotalCharges'] == ' '].index)
        self.data['TotalCharges']=self.data['TotalCharges'].astype(float)
        l=self.get_all_Null(dtype='float64')

        self.outlier_remove('TotalCharges')
        self.outlier_remove('MonthlyCharges')

        self.StdScale()

        self.drop_uniq_thresh(thresh=5)

        #self.Label_Encoding()
        self.one_hot_encoding()
        #self.drop_correlation()
        #self.drop_vif(thresh=4)

    # Remove Correlation
    def drop_correlation(self):
        k=Utils_Suite(self.data).compute_correlation(0.3)
        f=pd.DataFrame(k)
        m=list(f[(f['Churn']<0.15) & (f['Churn']>-0.15)].index)
        self.data=self.data.drop(columns=m)

    def ClearNull(self,threshold):
        x=self.data.isna().sum()>0
```

```
        for i in  list(x.index):
            thresh=self.data[i].isna().sum()/len(self.data)
            if(x[i]==True and thresh>threshold):
                print(i,self.data[i].isna().sum())
                self.data=self.data.drop(i,axis=1)


    def knn_impute(self,n_neighbors,col_list):
        imputer=knn(n_neighbors=n_neighbors)
        for i in col_list:
            self.data[i]=imputer.fit_transform(self.data[[i]])[0][0]


    def get_all_Null(self,dtype=""):
        x=self.data.isna().sum()>0
        l=[]
        for i in  list(x.index):
            thresh=self.data[i].isna().sum()/len(self.data)
            if(x[i]==True and (data[i].dtypes==dtype) ):
                print(i,data[i].isna().sum())
                l+=[i]
        return l
    # Drop Outlier Rows
    def outlier_remove(self,col):

        q1=self.data[col].quantile(0.25)
        q3=self.data[col].quantile(0.75)
        iqr=q3-q1
        l_whis=q1-1.5*iqr
        u_whis=q3+1.5*iqr
        self.data= self.data[(self.data[col]>=l_whis)& (self.data[col]<=u_whis)]

    # One hot Encoding using get_dummies
    def one_hot_encoding(self):
        z=(self.data.dtypes=='object')
        k=pd.DataFrame(z)
        obj_list=list(k[k[0]==True].index)
        print(obj_list)
        for i in obj_list:
            dummy=pd.get_dummies(self.data[i],prefix=i,drop_first=True)

            self.data=self.data.drop(i,axis=1)
            self.data=self.data.join(dummy)

    # standardize data
    def drop_uniq_thresh(self,thresh=5):
```

```
        col=data.columns
        x=pd.DataFrame(self.data.dtypes)
        ll=list(x[x[0]=="object"].index)
        droplist=[]
        for i in ll:
            print(i)
            if (len(self.data[i].unique())>thresh):
                droplist+=[i]
        print(droplist)
        self.data=self.data.drop(columns=droplist)


    def Label_Encoding(self):
        label_encoder = preprocessing.LabelEncoder()
        x=pd.DataFrame(self.data.dtypes)
        ll=list(x[x[0]=="object"].index)
        for i in ll:
            self.data[i]= label_encoder.fit_transform(self.data[i])

    def StdScale(self):
        for i in self.data.columns:
            if self.data[i].dtypes!='object' and i!='Churn':
                scale = StandardScaler().fit(self.data[[i]])

                self.data[i] = scale.transform(self.data[[i]])


    ## DANGER ZONE Col Spare NEEDED To Keep y_pred. RAM HOGGING FUNCTION .
    #Use Wisely! Plus Parallize the operation for better efficacy? Maybe???

    def drop_vif(self,thresh=5,col_Spare=['Churn','intercept']):


        vif=Utils_Suite(self.data).compute_vif()
        z1=vif[vif["vif"]>thresh]
        z1=z1.sort_values(by='vif', kind='mergesort',ascending=[False])
        while True:
            try:
                col=z1.iloc[0,0]
                if z1.empty:
                    break
                if col in col_Spare:
                    z1=z1.iloc[1:]
                    continue
                self.data=self.data.drop(col,axis=1)
                vif=Utils_Suite(self.data).compute_vif()
                z1=vif[vif["vif"]>thresh]
```

```
                z1=z1.sort_values(by='vif', kind='mergesort',ascending=[False])
            except IndexError:
                break

    # Wrapper Function to dump data to a variable
    def write_df(self):
        return self.data
```

[10]:
```
x=PreProcess(data)
data=x.write_df()
```

```
customerID
gender
Partner
Dependents
PhoneService
MultipleLines
InternetService
OnlineSecurity
OnlineBackup
DeviceProtection
TechSupport
StreamingTV
StreamingMovies
Contract
PaperlessBilling
PaymentMethod
Churn
['customerID']
['gender', 'Partner', 'Dependents', 'PhoneService', 'MultipleLines',
'InternetService', 'OnlineSecurity', 'OnlineBackup', 'DeviceProtection',
'TechSupport', 'StreamingTV', 'StreamingMovies', 'Contract', 'PaperlessBilling',
'PaymentMethod', 'Churn']
```

[6]:
```
from sklearn.model_selection import train_test_split
```

[11]:
```
x=data.iloc[:,:-1].values
y=data.iloc[:,-1].values
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.
 ↪3,random_state=2024)
```

0.0.2 Support Vector Classifier

[13]:
```
from sklearn import svm
from sklearn.metrics import␣
 ↪confusion_matrix,accuracy_score,classification_report
#Create a sum Classifier
```

```
clf = svm.SVC(kernel='poly') # Linear Kernel

#Train the model using the training sets
clf.fit(x_train, y_train)

#Predict the response for test dataset
y_pred = clf.predict(x_test)
accuracy_score(y_test, y_pred)
```

[13]:
```
0.8023696682464455
```

[45]:
```
clf.get_params().keys()
```

[45]:
```
dict_keys(['C', 'break_ties', 'cache_size', 'class_weight', 'coef0',
'decision_function_shape', 'degree', 'gamma', 'kernel', 'max_iter',
'probability', 'random_state', 'shrinking', 'tol', 'verbose'])
```

Grid SearchCV (SVC)

[52]:
```
param={

    'kernel':['linear','poly','rbf'],
    'gamma':['scale','auto'],
    'shrinking':[True,False],

    'probability':[True,False],
    'break_ties':[True,False],



  }
cls_grid=svm.SVC()
gr=GridSearchCV(cls_grid,param_grid=param,cv=5)  # class model,param,crossval=5
gr.fit(x_train,y_train)
gr.best_params_
```

[52]:
```
{'kernel': 'linear'}
```

0.0.3 Random Search CV(SVC)

[53]:
```
gr=RandomizedSearchCV(cls_grid,param_distributions=param,cv=5)
gr.fit(x_train,y_train)
gr.best_params_
```

```
e:\anaconda\lib\site-packages\sklearn\model_selection\_search.py:292:
UserWarning: The total space of parameters 3 is smaller than n_iter=10. Running
3 iterations. For exhaustive searches, use GridSearchCV.
  warnings.warn(
```

[53]:
```
{'kernel': 'linear'}
```

0.0.4 Decision Tree Classifier

[14]:
```
from sklearn.tree import DecisionTreeClassifier
tree=DecisionTreeClassifier(criterion="entropy",max_depth=4)
tree.fit(x_train,y_train)
y_pred=tree.predict(x_test)
(y_test,y_pred)

confusion_matrix(y_test,y_pred)
accuracy_score(y_test,y_pred)
```

[14]:
```
0.7909952606635071
```

0.0.5 Grid Search CV (Decision Tree Classifier)

[41]:
```
param={

    'criterion':['gini','entropy'],
    'splitter':['best','random'],

    'max_depth':[i for i in range(10,20,1)],
    'min_samples_split':[i for i in range(2,10)],
    'max_features':['sqrt','log2'],


  }
cls_grid=DecisionTreeClassifier()
gr=GridSearchCV(cls_grid,param_grid=param,cv=5)  # class model,param,crossval=5
gr.fit(x_train,y_train)
gr.best_params_
```

[41]:
```
{'criterion': 'entropy',
 'max_depth': 10,
 'max_features': 'sqrt',
 'min_samples_split': 7,
 'splitter': 'random'}
```

0.0.6 Random SearchCV(Decision Tree)

[42]:
```
gr=RandomizedSearchCV(cls_grid,param_distributions=param,cv=5)
gr.fit(x_train,y_train)
gr.best_params_
```

[42]:
```
{'splitter': 'random',
 'min_samples_split': 9,
```

```
'max_features': 'sqrt',
'max_depth': 11,
'criterion': 'entropy'}
```

0.0.7 Random Forest Classifier

[15]:
```
from sklearn.preprocessing import StandardScaler


ss=StandardScaler()
x_train=ss.fit_transform(x_train)
x_test=ss.transform(x_test)
```

[24]:
```
from sklearn.ensemble import RandomForestClassifier
rfc=RandomForestClassifier(n_estimators=75,criterion='gini',bootstrap=True,random_state=
rfc.fit(x_train,y_train)
y_pred=rfc.predict(x_test)
accuracy_score(y_pred,y_test)
```

[24]:
```
0.8023696682464455
```

GridSearchCV(RandomForest)

[21]:
```
param={
    'n_estimators':[i for i in range(50,100,25)],
    'criterion':['gini','entropy'],
    'bootstrap':[True,False]

  }
cls_grid=RandomForestClassifier()
gr=GridSearchCV(cls_grid,param_grid=param,cv=5)  # class model,param,crossval=5
gr.fit(x_train,y_train)
gr.best_params_
```

[21]:
```
{'bootstrap': True, 'criterion': 'entropy', 'n_estimators': 75}
```

0.0.8 Random SearchCV(Random Forest)

[23]:
```
gr=RandomizedSearchCV(cls_grid,param_distributions=param,cv=5)
gr.fit(x_train,y_train)
gr.best_params_
```

```
e:\anaconda\lib\site-packages\sklearn\model_selection\_search.py:292:
UserWarning: The total space of parameters 8 is smaller than n_iter=10. Running
8 iterations. For exhaustive searches, use GridSearchCV.
  warnings.warn(
```

[23]:
```
{'n_estimators': 75, 'criterion': 'gini', 'bootstrap': True}
```

0.0.9 Adaboost

[33]:
```
from sklearn.ensemble import AdaBoostClassifier
cl=AdaBoostClassifier(n_estimators=25, algorithm="SAMME.R",learning_rate=1.0,␣
 ↪random_state=0)
cl.fit(x_train,y_train)
y_pred=cl.predict(x_test)
accuracy_score(y_pred,y_test)
```

[33]:
```
0.8118483412322275
```

GridSearchCV(AdaBoost)

[32]:
```
param={
    'n_estimators':[i for i in range(0,100,25)],
    'learning_rate':[i for i in np.arange(0,10,0.5)],
    'algorithm':['SAMME','SAMME.R'],
  }
cls_grid=AdaBoostClassifier()
gr=GridSearchCV(cls_grid,param_grid=param,cv=5)  # class model,param,crossval=5
gr.fit(x_train,y_train)
gr.best_params_
```

Result:

Thus the Grid Search and Randomised Search was used to optimize the parameters and increase accuracy

Ex. No: 7	KMeans From Scratch
06.03.2024	

Aim:

To perform KMeans from scratch and compare the silhouette score with the actual implemntation of sklearn .

Algorithm:

1. Using the given data compute random centroid and compute distances from the centroid
2. Update Distances such that dist (centroid,point) is minimum
3. Once that's done Update new centroid
4. Assign points to new centroid
5. Continue until the algorithm converges. That is till there is no change in the previous centroid and the new centroid
6. Compute the silhouette score and conpare with the actual implementation with sklearn

Code+Output:

[36]:
```
import pandas as pd
import numpy as np
import seaborn as sns
```

0.0.1 K-Means From Scratch

[37]:
```
data=pd.read_csv(r'D:\data.csv')
```

[38]:
```
data
```

[38]:

	CustomerID	Gender	Age	Annual Income	Spending Score
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40
..	…	…	…	…	…
195	196	Female	35	120	79
196	197	Female	45	126	28
197	198	Male	32	126	74
198	199	Male	32	137	18
199	200	Male	30	137	83

[200 rows x 5 columns]

[73]:
```
class KMeans:
    def __init__(self,data_n,n_clusters=5):
        self.data=data_n.values
        self.n_clusters=n_clusters

        self.create_centroid()
        print(self.centroid)
        #self._run()
```

```
    def dist(self,centroid,dt):
        return np.sqrt(np.sum((centroid-dt)**2))
    def create_centroid(self):
        self.centroid= [[i,np.random.uniform(self.data.min(),self.data.
↪max()),np.random.uniform(self.data.min(),self.data.max())] for i in␣
↪range(self.n_clusters)]
    def _run(self):
        previous_centroid=None
        current_centroid=self.centroid
        points={}
        for i in range(len(self.centroid)):
            points.update({self.centroid[i][0]:[]})
        while(previous_centroid!=current_centroid):

            for k in range(len(self.data)):
                min_dist=np.inf
                centroid_k=None
                for i in range(len(current_centroid)):
                    tmp=self.dist(current_centroid[i][1:],self.data[k])
                    if(tmp<min_dist):
                        min_dist=tmp
                        centroid_k=i
                #print(centroid_k)
                points[current_centroid[centroid_k][0]].append(self.data[k])
            previous_centroid=current_centroid
            print(len(points[0]),len(points[1]),len(points[2]))
            for i in range(len(current_centroid)):
                avg=0
                avg=sum(points[i])/len(points[i])
                print("avg",avg)
                current_centroid[i][1:]=avg

            dup=points
            points={current_centroid[i][0]:dup[current_centroid[i][0]] for i in␣
↪range(len(current_centroid))}


        self.clusters=points
        l=[]
        for i in range(len( self.data)):
            for j in range(self.n_clusters):
                #tmp=
                for k in points[j]:
                    if((self.data[i]==k).all()):
                        l.append(j)
                        break
```

```
        self.KMeans_labels=l
        print(l)
    def  silhouette_score(self):
        return silhouette_score(self.data, self.KMeans_labels,␣
↪metric='euclidean')
```

[49]:
```
data=data[['Annual Income','Spending Score']]
```

[83]:
```
z=KMeans(data[['Annual Income','Spending Score']],n_clusters=3)
z._run()
```

```
[[0, 124.52879949507005, 11.183561099111675], [1, 33.4127386685388,
40.60764305226854], [2, 134.1543940297911, 74.45315067737035]]
30 142 28
avg [90.63333333 15.96666667]
avg [48.00704225 50.85915493]
avg [92.         83.53571429]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
0, 1, 0, 1, 0, 1, 0, 2, 1, 2, 0, 2, 1, 1, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 1, 1,
2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2,
0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2]
```

[84]:
```
z.silhouette_score()
```

[84]: 0.4027896475824334

[68]:
```
z.KMeans_labels
```

[68]: [1, 1, 1, 1, 1, 1]

[22]:
```
data[['Annual Income','Spending Score']]
```

[22]:

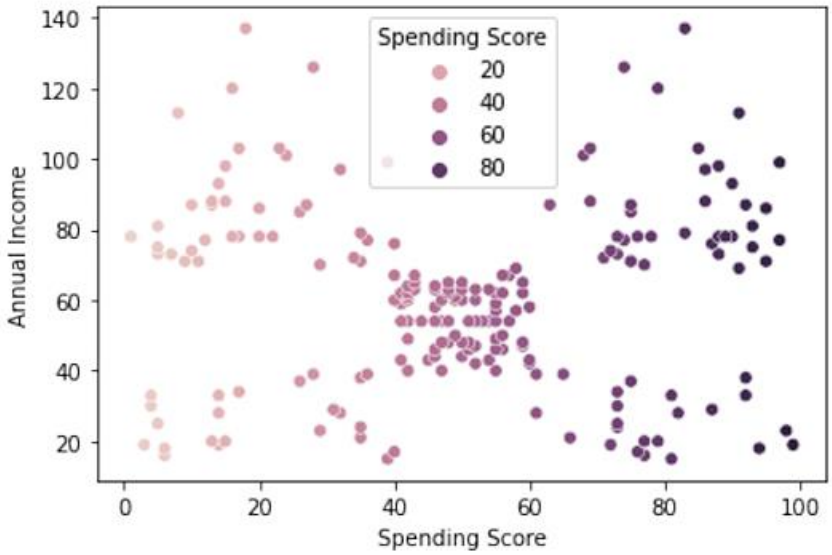
	Annual Income	Spending Score
0	15	39
1	15	81
2	16	6
3	16	77
4	17	40
..	…	…

```
197             126             74
198             137             18
199             137             83

[200 rows x 2 columns]
```

0.0.2 KMeans Using skLearn

[27]:
```
sns.scatterplot(data = data, x = 'Spending Score', y = 'Annual Income', hue =␣
↪'Spending Score')
```

[27]:
```
<AxesSubplot:xlabel='Spending Score', ylabel='Annual Income'>
```



0.1 Performing Scaling (Unnecessary in this case but anyways....)

[ ]:
```
```

[26]:
```
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test=train_test_split(data[['Age','Annual␣
↪Income','Spending Score']],data['Gender'])
x_train_n=preprocessing.normalize(x_train)
```

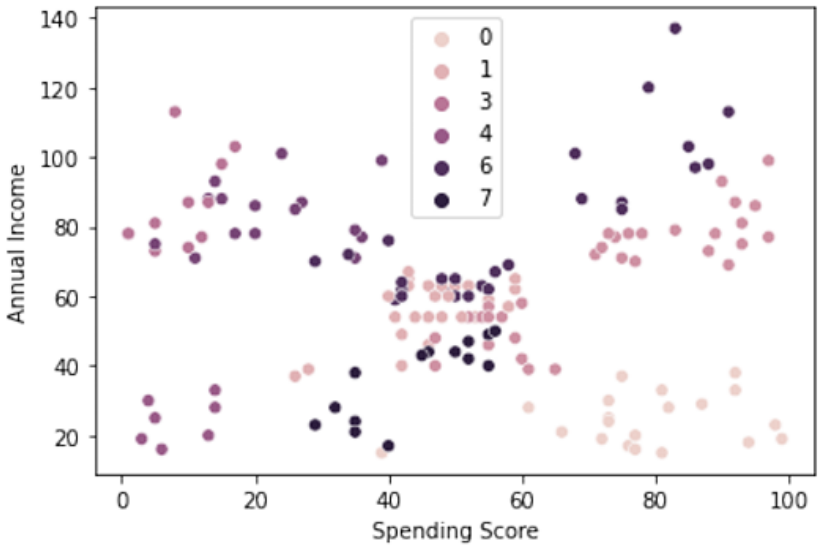
[27]:
```
from sklearn.cluster import KMeans
kmeans=KMeans(n_clusters=8,random_state=0)
```

[29]:
```
kmeans.fit(x_train_n)
```

[29]:
```
KMeans(random_state=0)
```

[30]:
```
sns.scatterplot(data = x_train, x = 'Spending Score', y = 'Annual Income', hue
↪= kmeans.labels_)
```

[30]:
```
<AxesSubplot:xlabel='Spending Score', ylabel='Annual Income'>
```



[31]:
```
kmeans.labels_
```

[31]:
```
array([1, 5, 2, 2, 3, 2, 1, 6, 1, 3, 2, 1, 6, 0, 2, 6, 0, 1, 1, 3, 0, 6,
       2, 4, 2, 6, 5, 2, 2, 5, 0, 1, 6, 0, 7, 6, 0, 0, 1, 5, 5, 6, 3, 2,
       2, 5, 2, 1, 7, 3, 6, 2, 6, 3, 4, 1, 3, 2, 2, 1, 7, 2, 2, 2, 5, 4,
       3, 2, 2, 1, 6, 6, 7, 2, 6, 1, 5, 0, 2, 0, 0, 0, 0, 0, 2, 4, 6, 7,
       2, 6, 6, 1, 3, 5, 1, 3, 0, 1, 2, 5, 1, 1, 0, 1, 7, 1, 4, 6, 6, 5,
       7, 1, 2, 1, 6, 5, 7, 6, 2, 6, 7, 6, 6, 7, 1, 6, 6, 1, 0, 2, 5, 0,
       2, 0, 2, 7, 0, 5, 5, 2, 7, 0, 6, 7, 4, 3, 1, 2, 4, 7])
```

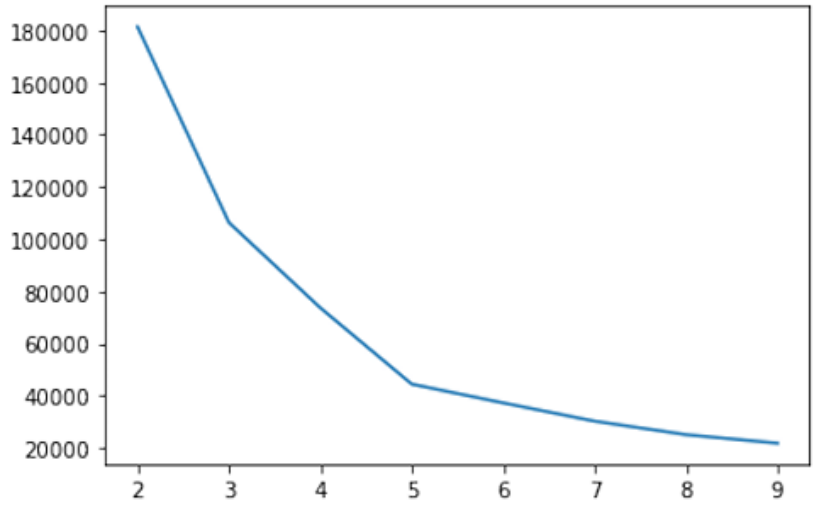
[32]:
```
from sklearn.metrics import silhouette_score

silhouette_score(x_train_n, kmeans.labels_, metric='euclidean')
```

[32]:
```
0.40563594215217225
```

[33]:
```
wcss=[]
for i in range(2,10):
    kmeans=KMeans(n_clusters=i,random_state=0)
    kmeans.fit(data[[ 'Annual Income', 'Spending Score']].values)
    wcss.append(kmeans.inertia_)
```

[34]:
```
import matplotlib.pyplot as plt
plt.plot(range(2,10),wcss)
```

[34]:
```
[<matplotlib.lines.Line2D at 0x1e285fe5190>]
```



[ ]:

0.1.1 Interpretation

Therefore the sklearn implementation needed 5 clusters to explain the data but in my from the scratch implementation 3 clusters is enough for achieving a high silhouette score However There is a caveat of running into zero division error once we reach >6 clusters in my implementation as sometimes some data is biased towards one cluster... Plus Clusters By themselves do not mean anything Unless we specify the dimension Here 2d that explains the relation between Annual Income

Result:

Thus the KMeans algorithm was implemented from scratch and their scores compared with existing sklearn implementation

Ex. No: 8	PCA MNIST From Scratch
19.03.2024	

Aim:

To Compute PCA From Scratch and apply PCA on MNIST Digit dataset

Algorithm:

1.Compute the covariance matrix with the data
2.Compute eigen values and eigen vectors
3.Sort the data using the eigenvalues and corresponding eigenvectors in descending order
4.Fit and transform N Dimensional data into N-K dimensions
5.Explain the variance between n Principle components

Code + Output:

[36]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Generic Class to perform PCA

Note : I am not implementing Cov /eig from scratch to save time!

[2]:
```
class PCA:

    def __init__(self,data,n_components):
        self.data=data
        self.n_components=n_components
        self.compute()

    def compute(self):
        cov=np.cov(self.data,rowvar=False)
        self.eig_val,self.eig_v=np.linalg.eigh(cov)
        sorted_idx=np.argsort(self.eig_val)[::-1]
        self.eig_val=self.eig_val[sorted_idx]
        self.eig_v=self.eig_v[:,sorted_idx]

    def fit_transform(self):
        tmp= np.dot(self.data,self.eig_v[:,:self.n_components])
        return tmp/np.sqrt(self.eig_val[:self.n_components])
```

Importing MNIST Digit Data

[3]:
```
train=pd.read_csv(r'E:\Machine Learning Algorithms\Ex7 PCA\train.csv')
y=y=train.iloc[:,0]
```

0.0.1 Data Normalised to 0->1 dividing by 255.0 (val changes from 0 to 255)

[4]:
```
df=train.iloc[:,1:]/255.0
```

[33]:
```
x=PCA(df,2)
dp=x.fit_transform()
```

[34]:
```
dp
```

[34]:
```
array([[-0.17929209, -1.41068277],
       [-4.26335393, -0.72751754],
       [ 0.21009223, -0.59283296],
       …,
       [-1.9039064 ,  1.91545772],
       [-1.88857534,  0.12466016],
       [-0.69319706,  0.84436166]])
```

[28]:
```
data_label
```

[28]:

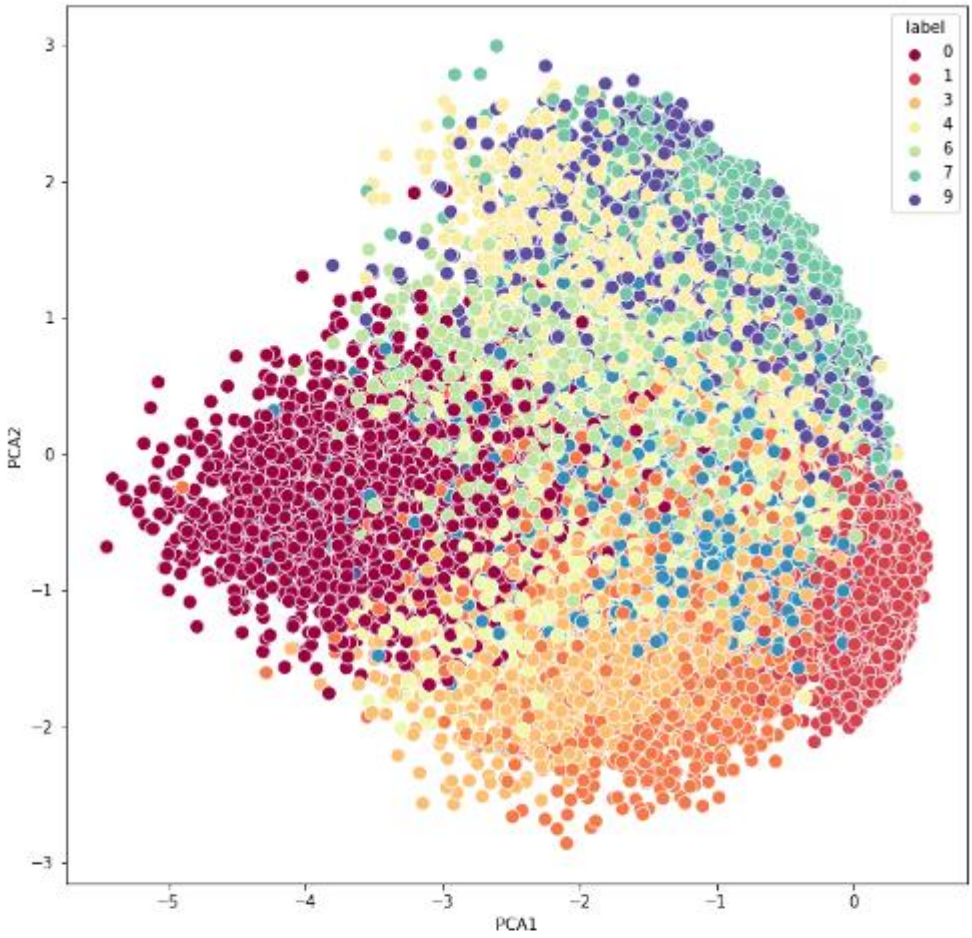
	PCA1	PCA2	label
0	-0.179292	-1.410683	1
1	-4.263354	-0.727518	0
2	0.210092	-0.592833	1
3	-1.036254	0.604963	4
4	-4.647483	-0.906197	0
…	…	…	…
41995	-2.636854	-0.504278	0
41996	0.509052	-0.610028	1
41997	-1.903906	1.915458	7
41998	-1.888575	0.124660	6
41999	-0.693197	0.844362	9

[42000 rows x 3 columns]

Visualizing the 2d PCA Features using Seaborn

[35]:
```
data=pd.DataFrame(dp,columns=['PCA1','PCA2'])
data_label=pd.concat([data,pd.DataFrame(y)],axis=1)# Concat along columns
plt.figure(figsize=(10,10))
sns.scatterplot(data=data_label,x='PCA1',y='PCA2',hue=y,s=80,palette='Spectral')
```

[35]:
```
<AxesSubplot:xlabel='PCA1', ylabel='PCA2'>
```



References: https://www.askpython.com/python/examples/principal-component-analysis

https://www.kaggle.com/c/digit-recognizer/data

https://numpy.org/doc/stable/reference/generated/numpy.linalg.eigh.html

https://seaborn.pydata.org/generated/seaborn.scatterplot.html

Result:

Thus PCA was Implemented from scratch and plotted on a 2-d graph space

Ex. No: 9	Neural Networks
23.03.2024	

Aim:

To implement Neural Networks from scratch for performing Regression on the boston dataset

Algorithm:

1. Create a Neural Net Class that accepts n_layers,n_neurons etc
2. Initially initialize the weights and bias randomly
3. For Forward Propogation use the necessary Equation to feed forward
4. Define loss function (cross entropy loss).
5. For Backward propogation use the necessary equations to update weights
6. Perform Gradient descent optimization to update weights for the next epoch
7. Load the dataset and train using the NNFS and compute MSE (Regression)
8. Using Keras NN compute the same and compare the MSE for both

Code + Output :

[3]:
```
import numpy as np
import pandas as pd
```

0.0.1 From the Scratch Implementation of Neural Networks

Self Note: I am using only Simple Activation Function (Sigmoid) whose derivative was done in class. This data set is used as advised in the Question (load_boston Data set). Ideally the code quality can be better and Must have made it even more modular (n layers). But here for demonstration purposes and time concerns I am limiting the same and sticking to the equations to backprop . I am using this simple NN for Regression purposes

[4]:
```
class NeuralNet:
    def __init__(self,n_input,n_neurons,n_output):
        self.n_input=n_input
        self.n_neurons=n_neurons
        self.n_output=n_output
        self.W_hidden=np.random.randn(n_input,n_neurons)
        self.b_hidden=np.zeros((1,n_neurons))
        self.W_output=np.random.randn(n_neurons,n_output)
        self.b_output=np.zeros((1,n_output))
    def sigmoid(self,x):
        return 1/(1+np.exp(-x))
    def forward(self,X):
        self.Z=np.dot(X,self.W_hidden)+self.b_hidden
        self.A_h=self.sigmoid(self.Z)
        self.Z_output=np.dot(self.A_h,self.W_output)+ self.b_output
        return self.Z_output
    def loss(self,y):
        m=y.shape[0]
        self._loss=(1/m)*(np.sum(y-self.Z_output)**2)
        return self._loss

    def backward(self,X,y):
```

```
        m=y.shape[0]
        self.d_Z=self.Z_output-y
        self.dW=(1/m)*(np.dot(self.A_h.T,self.d_Z))
        self.dB=(1/m)*(np.sum(self.d_Z))
        #tmp=np.dot(self.d_Z,self.W_output.T)
        self.d_Z1=np.dot(self.d_Z,self.W_output.T)*(self.A_h*(1-self.A_h))
        self.d_w1=(1/m)*np.dot(X.T,self.d_Z1)
        self.d_b1=(1/m)*(np.sum(self.d_Z1))
    def parameter_update(self,learn_rate):
        self.W_output=self.W_output-learn_rate*self.dW
        self.b_output=self.b_output-learn_rate*self.dB
        self.W_hidden=self.W_hidden-learn_rate*self.d_w1
        self.b_hidden=self.b_hidden-learn_rate*self.d_b1
    def train(self,n_epochs,X,y,learn_rate):
        y=y.reshape(-1,1)
        for i in range(n_epochs+1):
            self.Z_output=self.forward(X)
            loss=self.loss(y)
            self.backward(X,y)
            self.parameter_update(learn_rate=learn_rate)
            if(i%10==0):
                print(f'Epoch{i},loss:{loss}')
```

0.0.2 Loading the Load_Boston Dataset

[7]:
```
import numpy as np
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
data=load_boston()
X=data.data
y=data.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.34,␣
 ↪random_state=42)
sc=StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

```
e:\anaconda\lib\site-packages\sklearn\utils\deprecation.py:87: FutureWarning:
Function load_boston is deprecated; `load_boston` is deprecated in 1.0 and will
be removed in 1.2.
```

```
    The Boston housing prices dataset has an ethical problem. You can refer to
    the documentation of this function for further details.

    The scikit-learn maintainers therefore strongly discourage the use of this
    dataset unless the purpose of the code is to study and educate about
    ethical issues in data science and machine learning.

    In this special case, you can fetch the dataset from the original
    source::

        import pandas as pd
        import numpy as np


        data_url = "http://lib.stat.cmu.edu/datasets/boston"
        raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)
        data = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :2]])
        target = raw_df.values[1::2, 2]

    Alternative datasets include the California housing dataset (i.e.
    :func:`~sklearn.datasets.fetch_california_housing`) and the Ames housing
    dataset. You can load the datasets as follows::

        from sklearn.datasets import fetch_california_housing
        housing = fetch_california_housing()

    for the California housing dataset and::

        from sklearn.datasets import fetch_openml
        housing = fetch_openml(name="house_prices", as_frame=True)

    for the Ames housing dataset.

  warnings.warn(msg, category=FutureWarning)
```

0.0.3 Training the Simple NN

```
With 13 Inputs / Features (load_boston) has 13 features I'm not selective of specific featur
100 Neurons Per Layer
1 Output Layer
200 Epochs
```

[8]:
```
x=NeuralNet(13,100,1)
x.train(200,X_train,y_train,0.01)
```

```
Epoch0,loss:241725.0073386548
Epoch10,loss:47.257043972638215
```

```
Epoch20,loss:25.666594000089724
Epoch30,loss:18.39902061649305
Epoch40,loss:12.088855408793751
Epoch50,loss:7.6661803792062075
Epoch60,loss:4.845340367733449
Epoch70,loss:3.101441181839372
Epoch80,loss:2.0236609377332564
Epoch90,loss:1.3509187466909074
Epoch100,loss:0.9257739038458271
Epoch110,loss:0.6530420987658092
Epoch120,loss:0.47482957755109617
Epoch130,loss:0.3558657519983528
Epoch140,loss:0.2745941117907178
Epoch150,loss:0.21774267096331576
Epoch160,loss:0.1770476798685013
Epoch170,loss:0.1472897353693773
Epoch180,loss:0.12511680659789576
Epoch190,loss:0.10833552079261284
Epoch200,loss:0.09548113624670233
```

[36]:

[36]:
```
14.706232512585812
```

[35]:
```
y_pred
```

[35]:
```
array([[27.31746447],
       [32.58694043],
       [14.56668538],
       [24.40046708],
       [16.57291892],
       [21.53781959],
       [16.08170568],
       [15.46889132],
       [20.80686285],
       [20.00900021],
       [26.56980231],
       [19.21340724],
       [ 8.30163996],
       [21.02685668],
       [16.95404491],
       [25.5384165 ],
       [18.95824666],
       [ 7.28473322],
       [36.76522298],
       [15.60315729],
       [28.71439474],
       [30.35501432],
```

[20]:
```
y_test
```

[20]:
```
array([23.6, 32.4, 13.6, 22.8, 16.1, 20. , 17.8, 14. , 19.6, 16.8, 21.5,
       18.9,  7. , 21.2, 18.5, 29.8, 18.8, 10.2, 50. , 14.1, 25.2, 29.1,
       12.7, 22.4, 14.2, 13.8, 20.3, 14.9, 21.7, 18.3, 23.1, 23.8, 15. ,
       20.8, 19.1, 19.4, 34.7, 19.5, 24.4, 23.4, 19.7, 28.2, 50. , 17.4,
       22.6, 15.1, 13.1, 24.2, 19.9, 24. , 18.9, 35.4, 15.2, 26.5, 43.5,
       21.2, 18.4, 28.5, 23.9, 18.5, 25. , 35.4, 31.5, 20.2, 24.1, 20. ,
       13.1, 24.8, 30.8, 12.7, 20. , 23.7, 10.8, 20.6, 20.8,  5. , 20.1,
       48.5, 10.9,  7. , 20.9, 17.2, 20.9,  9.7, 19.4, 29. , 16.4, 25. ,
       25. , 17.1, 23.2, 10.4, 19.6, 17.2, 27.5, 23. , 50. , 17.9,  9.6,
       17.2, 22.5, 21.4, 12. , 19.9, 19.4, 13.4, 18.2, 24.6, 21.1, 24.7,
        8.7, 27.5, 20.7, 36.2, 31.6, 11.7, 39.8, 13.9, 21.8, 23.7, 17.6,
       24.4,  8.8, 19.2, 25.3, 20.4, 23.1, 37.9, 15.6, 45.4, 15.7, 22.6,
       14.5, 18.7, 17.8, 16.1, 20.6, 31.6, 29.1, 15.6, 17.5, 22.5, 19.4,
       19.3,  8.5, 20.6, 17. , 17.1, 14.5, 50. , 14.3, 12.6, 28.7, 21.2,
       19.3, 23.1, 19.1, 25. , 33.4,  5. , 29.6, 18.7, 21.7, 23.1, 22.8,
       21. , 48.8, 14.6, 16.6, 27.1, 20.1, 19.8, 21. ])
```

[38]:
```
print("Total MSE(Test Data):",np.sum(y_pred-y_test.reshape(-1,1))**2/y_test.
 ↪shape[0])
```

```
Total MSE(Test Data): 14.706232512585812
```

0.0.4 Using Keras NN

[9]:
```
from keras.models import Sequential
from keras.layers import Dense
model=Sequential()
model.add(Dense(100,activation='sigmoid',input_shape=(X_train.shape[1],)))
model.add(Dense(100, activation='sigmoid'))
model.add(Dense(1))# 1 output layer
model.compile( loss='mse', metrics=['mae'])
model.fit(X_train, y_train, epochs=200, batch_size=1,verbose=1)
test_mse_score, _ = model.evaluate(X_test,y_test )
print('Test MSE:', test_mse_score)
```

```
333/333 [==============================] - 1s 3ms/step - loss: 5.3626 - mae:
1.7050
Epoch 194/200
333/333 [==============================] - 1s 3ms/step - loss: 5.3488 - mae:
1.7380
Epoch 195/200
333/333 [==============================] - 1s 3ms/step - loss: 5.4716 - mae:
1.7205
Epoch 196/200
333/333 [==============================] - 1s 3ms/step - loss: 5.2964 - mae:
1.7565
Epoch 197/200
333/333 [==============================] - 1s 3ms/step - loss: 5.2040 - mae:
1.6812
Epoch 198/200
333/333 [==============================] - 1s 3ms/step - loss: 5.3459 - mae:
1.6418
Epoch 199/200
333/333 [==============================] - 1s 3ms/step - loss: 5.2799 - mae:
1.6944
Epoch 200/200
333/333 [==============================] - 1s 3ms/step - loss: 5.3147 - mae:
1.7082
6/6 [==============================] - 0s 2ms/step - loss: 12.7061 - mae: 2.5144
Test MSE: 12.706128120422363
```

Interpretations

```
My Implementation uses simple gradient descent that tunes the weights and bias (Initial
The Loss Metric is MSE (Mean Square Error)
For Keras NN: 12.78
For Scratch NN: Loss -> 14.706 for test data
Keras NN also has various optimizers which i am not using here but at the end of 200 it
But keras Performs better than my model as it is more optimized and has batching and we
```

Result:

Thus simple ANN was implemented from scratch and performed regression and compared the accuracy (MSE) with the standard Keras NN

Ex. No: 10	Association Rule Mining
03.04.2024	

Aim:

To perform Associative Rule mining using Apriori / FPtree Algorithm

Algorithm:

1.For this specific case I am using Apriori Algorithm
2. Choose minimum support and confidence threshold,
3. PreProcess the data into a nested list associating each row with a transaction
4.Compute the frequency of each item in the data frame
5. Using Itertools generate unique combinations within data(CrossProduct)
6. Now for 2 Combination perform the same frequency and eliminate items < min support
7. Perform Until reaching {} or all frequencies = min support
8. Compute the confidences using the same and filter using the threshold

Code + Output:

0.0.1 Association Rule Mining

[4]:
```
import numpy as  np
import pandas as pd
import pickle
import itertools
from collections import Counter
```

[115]:
```
class Apriori:
    def __init__(self,minimum_support,confidence_thresh):
        self.data=pd.read_csv(r'E:\Machine Learning Algorithms\Ex9 Association␣
↪Rule Mining\Market_Basket_Optimisation.csv')
        self.minimum_support=minimum_support
        self.confidence_thresh=confidence_thresh

    def _lazyread(self):
        dataset=self.data.fillna(0)
        self.transactions=[]
        for i in range(0,7500):
            self.transactions.append([str(dataset.values[i,j]) for j in␣
↪range(0, 20)])

    def _pkl_write_(self):
        with open(r'E:\Machine Learning Algorithms\Ex9 Association Rule␣
↪Mining\transactions.pkl','wb') as f:
            pickle.dump(self.transactions,f)
    def _pkl_read(self):
        with open(r'E:\Machine Learning Algorithms\Ex9 Association Rule␣
↪Mining\transactions.pkl','rb') as f:
            self.transactions=pickle.load(f)
        self.transactions=self.transactions[:100]
    def candidate_generation(self):
        d={}
        for i in range(len(self.transactions)):
            d.update({i:0})
        tmp=[]
        for i in ((self.transactions)):
```

```
            for j in ((i)):
                if(j!='0' and j!='0.0'):
                    tmp.append(j)
        d={}
        for i in tmp:
            d.update({i:0})
        for j in range(len(self.transactions)):
            _dict=dict(Counter(self.transactions[j]))
            print(_dict)
            try:
                _dict.pop('0')
                _dict.pop('0.0')
            except KeyError:
                continue
            for k in _dict:
                d[k]+=_dict.get(k)
        self.d=d
        self.data=pd.DataFrame(d.items(),columns=['items','frequency'])
        return self.data
    def cartesian_product(self,number):
        s=0
        d={}
        l=list(itertools.combinations(self.data['items'],number ))
        for i in range(len(l)):
            for j in range(len(self.transactions)):
                #tmp=list(itertools.product(list(z.transactions[j]),list(z.
↪transactions[j])))
                if (set(l[i]).intersection(set(self.
↪transactions[j])))==set(l[i]):
                    s+=1

            if (s>=1):
                d.update({l[i]:s})
            s=0
        self._candidate_combination=d
        return d
```

[116]:
```
z=Apriori(3,50)
```

[89]:
```
z.transactions
```

[89]:
```
['chutney',
 '0',
 '0',
 '0',
```

[118]:
```
df=z.candidate_generation()
```

```
{'burgers': 1, 'meatballs': 1, 'eggs': 1, '0': 16, '0.0': 1}
{'chutney': 1, '0': 18, '0.0': 1}
{'turkey': 1, 'avocado': 1, '0': 17, '0.0': 1}
{'mineral water': 1, 'milk': 1, 'energy bar': 1, 'whole wheat rice': 1, 'green
tea': 1, '0': 14, '0.0': 1}
{'low fat yogurt': 1, '0': 18, '0.0': 1}
{'whole wheat pasta': 1, 'french fries': 1, '0': 17, '0.0': 1}
{'soup': 1, 'light cream': 1, 'shallot': 1, '0': 16, '0.0': 1}
{'frozen vegetables': 1, 'spaghetti': 1, 'green tea': 1, '0': 16, '0.0': 1}
{'french fries': 1, '0': 18, '0.0': 1}
{'eggs': 1, 'pet food': 1, '0': 17, '0.0': 1}
{'cookies': 1, '0': 18, '0.0': 1}


{'turkey': 1, 'burgers': 1, 'mineral water': 1, 'eggs': 1, 'cooking oil': 1,
'0': 14, '0.0': 1}
{'spaghetti': 1, 'champagne': 1, 'cookies': 1, '0': 16, '0.0': 1}
{'mineral water': 1, 'salmon': 1, '0': 17, '0.0': 1}
{'mineral water': 1, '0': 18, '0.0': 1}
{'shrimp': 1, 'chocolate': 1, 'chicken': 1, 'honey': 1, 'oil': 1, 'cooking oil':
1, 'low fat yogurt': 1, '0': 12, '0.0': 1}
{'turkey': 1, 'eggs': 1, '0': 17, '0.0': 1}
{'turkey': 1, 'fresh tuna': 1, 'tomatoes': 1, 'spaghetti': 1, 'mineral water':
1, 'black tea': 1, 'salmon': 1, 'eggs': 1, 'chicken': 1, 'extra dark chocolate':
1, '0': 9, '0.0': 1}
{'meatballs': 1, 'milk': 1, 'honey': 1, 'french fries': 1, 'protein bar': 1,
'0': 14, '0.0': 1}
{'red wine': 1, 'shrimp': 1, 'pasta': 1, 'pepper': 1, 'eggs': 1, 'chocolate': 1,
'shampoo': 1, '0': 12, '0.0': 1}
{'rice': 1, 'sparkling water': 1, '0': 17, '0.0': 1}
{'spaghetti': 1, 'mineral water': 1, 'ham': 1, 'body spray': 1, 'pancakes': 1,
'green tea': 1, '0': 13, '0.0': 1}
{'burgers': 1, 'grated cheese': 1, 'shrimp': 1, 'pasta': 1, 'avocado': 1,
'honey': 1, 'white wine': 1, 'toothpaste': 1, '0': 11, '0.0': 1}
{'eggs': 1, '0': 18, '0.0': 1}
{'parmesan cheese': 1, 'spaghetti': 1, 'soup': 1, 'avocado': 1, 'milk': 1,
'fresh bread': 1, '0': 13, '0.0': 1}
{'ground beef': 1, 'spaghetti': 1, 'mineral water': 1, 'milk': 1, 'energy bar':
1, 'black tea': 1, 'salmon': 1, 'frozen smoothie': 1, 'escalope': 1, '0': 10,
'0.0': 1}
{'sparkling water': 1, '0': 18, '0.0': 1}
{'mineral water': 1, 'eggs': 1, 'chicken': 1, 'chocolate': 1, 'french fries': 1,
'0': 14, '0.0': 1}
```

```
{'eggs': 1, 'cake': 1, 'french fries': 1, '0': 16, '0.0': 1}
{'burgers': 1, 'spaghetti': 1, 'milk': 1, 'french fries': 1, 'green tea': 1,
'0': 14, '0.0': 1}
{'mineral water': 1, 'energy bar': 1, 'butter': 1, 'french fries': 1, '0': 15,
'0.0': 1}
{'burgers': 1, 'grated cheese': 1, 'herb & pepper': 1, 'mineral water': 1,
'eggs': 1, 'cooking oil': 1, '0': 13, '0.0': 1}
{'energy bar': 1, '0': 18, '0.0': 1}
{'energy bar': 1, 'energy drink': 1, '0': 17, '0.0': 1}
{'chocolate': 1, 'frozen smoothie': 1, '0': 17, '0.0': 1}
{'low fat yogurt': 1, '0': 18, '0.0': 1}
{'asparagus': 1, 'salad': 1, '0': 17, '0.0': 1}
{'burgers': 1, 'french fries': 1, 'low fat yogurt': 1, 'green tea': 1, '0': 15,
'0.0': 1}
{'low fat yogurt': 1, '0': 18, '0.0': 1}
{'burgers': 1, 'herb & pepper': 1, 'shrimp': 1, 'pasta': 1, 'spaghetti': 1,
'mineral water': 1, 'meatballs': 1, 'olive oil': 1, 'energy bar': 1, 'french
wine': 1, 'eggs': 1, 'salt': 1, '0': 7, '0.0': 1}
{'champagne': 1, 'low fat yogurt': 1, '0': 17, '0.0': 1}
{'champagne': 1, '0': 18, '0.0': 1}
{'burgers': 1, 'almonds': 1, 'eggs': 1, 'french fries': 1, 'cookies': 1, 'green
tea': 1, '0': 13, '0.0': 1}
{'ham': 1, 'soup': 1, 'escalope': 1, 'body spray': 1, '0': 15, '0.0': 1}
{'turkey': 1, 'ham': 1, 'frozen vegetables': 1, 'pepper': 1, 'oil': 1, 'extra
dark chocolate': 1, 'tea': 1, 'magazines': 1, '0': 11, '0.0': 1}
{'muffins': 1, 'eggs': 1, 'cookies': 1, '0': 16, '0.0': 1}
{'cookies': 1, '0': 18, '0.0': 1}
{'frozen vegetables': 1, 'whole wheat pasta': 1, 'ground beef': 1, 'spaghetti':
1, 'chocolate': 1, 'green tea': 1, '0': 13, '0.0': 1}
{'mineral water': 1, 'barbecue sauce': 1, 'chocolate': 1, '0': 16, '0.0': 1}
```

[86]:
```
df
```

[86]:

	items	frequency
0	c	0
1	h	0
2	u	0
3	t	0
4	n	0
5	e	0
6	y	0
7	.	0

[120]:
```
z.cartesian_product(2)
```

[120]:
```
{('burgers', 'meatballs'): 2,
 ('burgers', 'eggs'): 5,
 ('burgers', 'turkey'): 1,
```

```
('turkey', 'strawberries'): 1,
('turkey', 'olive oil'): 1,
('turkey', 'cider'): 1,
('turkey', 'tea'): 1,
('avocado', 'mineral water'): 2,
('avocado', 'milk'): 1,
('avocado', 'whole wheat rice'): 1,
('avocado', 'green tea'): 1,
('avocado', 'whole wheat pasta'): 1,
('avocado', 'french fries'): 1,
('avocado', 'soup'): 2,
('avocado', 'spaghetti'): 1,
('avocado', 'shrimp'): 1,
('avocado', 'chocolate'): 1,
('avocado', 'honey'): 1,
('avocado', 'pasta'): 1,
('avocado', 'body spray'): 1,
('avocado', 'pancakes'): 1,
('avocado', 'grated cheese'): 1,
('avocado', 'white wine'): 1,
('avocado', 'toothpaste'): 1,
('avocado', 'parmesan cheese'): 1,
('avocado', 'fresh bread'): 1,
('avocado', 'ground beef'): 1,
('avocado', 'herb & pepper'): 1,
('avocado', 'hot dogs'): 1,
('avocado', 'brownies'): 1,
('avocado', 'muffins'): 1,
('avocado', 'cider'): 1,
('mineral water', 'milk'): 3,
('mineral water', 'energy bar'): 4,
('mineral water', 'whole wheat rice'): 3,
('mineral water', 'green tea'): 3,
('mineral water', 'low fat yogurt'): 1,
('mineral water', 'whole wheat pasta'): 1,
('mineral water', 'french fries'): 2,
('mineral water', 'soup'): 1,
('mineral water', 'frozen vegetables'): 4,
('mineral water', 'spaghetti'): 8,
('mineral water', 'cooking oil'): 3,
('mineral water', 'salmon'): 3,
('mineral water', 'shrimp'): 1,
('mineral water', 'chocolate'): 5,
('mineral water', 'chicken'): 3,
('mineral water', 'honey'): 1,
('mineral water', 'fresh tuna'): 4,
('mineral water', 'tomatoes'): 2,
```

```
4                 turkey          5
..                   ...        ...
61              brownies          2
62               cereals          1
63  clothes accessories           1
64            bug spray           1
65              muffins           1

[66 rows x 2 columns]
```

[291]:
```
df_1=df[df['frequency']>=2] #min support 2
```

[243]:
```
df_1
```

[243]:

	items	frequency
0	burgers	4
1	meatballs	3
2	eggs	9
4	turkey	5
5	avocado	6
6	mineral water	15
7	milk	4
8	energy bar	3
9	whole wheat rice	2
10	green tea	7
11	low fat yogurt	2
13	french fries	7
14	soup	4
15	light cream	2
17	frozen vegetables	4
18	spaghetti	12
20	cookies	3
21	cooking oil	2
22	champagne	2
23	salmon	4
24	shrimp	3
25	chocolate	6
26	chicken	4
27	honey	4
29	fresh tuna	2
31	black tea	3
34	red wine	2
35	pasta	2
39	sparkling water	3
41	body spray	2
42	pancakes	2
44	white wine	2

```
48        ground beef          2
49    frozen smoothie          2
50           escalope          4
57            pickles          2
60           hot dogs          2
61           brownies          2
```

[292]:
```
z.data=df_1
```

[293]:
```
k=z.cartesian_product(2)
```

[294]:
```
df_2=pd.DataFrame(k.items(),columns=['items','freq'])
```

[295]:
```
df_2=df_2[df_2['freq']>=2]
```

[299]:
```
z.data=df_2
```

[296]:
```
df_1[df_1['items']=='eggs']
```

[296]:
```
  items  frequency
2  eggs          9
```

[297]:
```
df_1[df_1['items']=='burgers']['frequency'].values[0] #denominator
```

[297]:
```
4
```

[266]:
```
df_1
```

[266]:
```
                items  frequency
0             burgers          4
1           meatballs          3
2                eggs          9
4              turkey          5
5             avocado          6
6       mineral water         15
7                milk          4
8          energy bar          3
9    whole wheat rice          2
10          green tea          7
11     low fat yogurt          2
13       french fries          7
14               soup          4
15        light cream          2
17  frozen vegetables          4
18          spaghetti         12
20            cookies          3
21        cooking oil          2
22          champagne          2
```

```
23              salmon          4
24              shrimp          3
25           chocolate          6
26             chicken          4
27               honey          4
29          fresh tuna          2
31           black tea          3
34            red wine          2
35               pasta          2
39     sparkling water          3
41          body spray          2
42            pancakes          2
44          white wine          2
48         ground beef          2
49     frozen smoothie          2
50            escalope          4
57             pickles          2
60            hot dogs          2
61            brownies          2
```

[301]:
```
s1=z.data['items'].reset_index(drop=True)
```

[300]:
```
z.data
```

[300]:
```
                                items  freq
1                      (burgers, eggs)     2
19                      (eggs, turkey)     3
21                (eggs, mineral water)     4
27                    (eggs, chocolate)     3
28                      (eggs, chicken)     2
34              (turkey, mineral water)     2
46                      (avocado, soup)     2
57                (mineral water, milk)     2
58         (mineral water, energy bar)     2
59    (mineral water, whole wheat rice)     2
60           (mineral water, green tea)     2
62   (mineral water, frozen vegetables)     2
63           (mineral water, spaghetti)     5
65              (mineral water, salmon)     3
66           (mineral water, chocolate)     2
67             (mineral water, chicken)     3
69          (mineral water, fresh tuna)     2
70           (mineral water, black tea)     2
74         (mineral water, ground beef)     2
75     (mineral water, frozen smoothie)     2
76             (mineral water, escalope)     2
77                     (milk, energy bar)     2
```

```
82                    (milk, spaghetti)     2
108             (green tea, spaghetti)     3
112            (green tea, body spray)     2
113              (green tea, pancakes)     2
117              (green tea, brownies)     2
136                   (soup, hot dogs)     2
138     (frozen vegetables, spaghetti)     3
146                (spaghetti, salmon)     3
150           (spaghetti, fresh tuna)     2
151            (spaghetti, black tea)     2
156      (spaghetti, frozen smoothie)     2
157             (spaghetti, escalope)     3
168               (salmon, black tea)     2
171                (salmon, escalope)     2
173               (shrimp, chocolate)     2
175                   (shrimp, honey)     2
177                   (shrimp, pasta)     2
179              (chocolate, chicken)     2
203            (body spray, pancakes)     2
211       (frozen smoothie, escalope)     2
```

[307]:
```
s1=pd.DataFrame(s1,columns=['items'])
```

[346]:
```
df_2=pd.DataFrame(df_2.reset_index(drop=True),columns=['items','freq'])
```

```
76            (mineral water, escalope)     2
77                   (milk, energy bar)     2
82                    (milk, spaghetti)     2
108             (green tea, spaghetti)      3
112            (green tea, body spray)      2
113              (green tea, pancakes)      2
117              (green tea, brownies)      2
136                   (soup, hot dogs)      2
138     (frozen vegetables, spaghetti)      3
146               (spaghetti, salmon)       3
150            (spaghetti, fresh tuna)      2
151             (spaghetti, black tea)      2
156       (spaghetti, frozen smoothie)      2
157              (spaghetti, escalope)      3
168                (salmon, black tea)      2
171                 (salmon, escalope)      2
173                (shrimp, chocolate)      2
175                    (shrimp, honey)      2
177                    (shrimp, pasta)      2
179               (chocolate, chicken)      2
203            (body spray, pancakes)       2
211         (frozen smoothie, escalope)     2
```

[349]:
```
df_1=pd.DataFrame(df_1.reset_index(drop=True),columns=['items','frequency'])
```

[375]:
```
df=pd.DataFrame()
l=[]
for i in range(1,len(df_1)):
    t=s1['items'][i]
    #print(t[0])
    #for j in t:
    #print(df_1[df_1['items']=='burgers'])
    #print(t[0],df_1['items'][0])
    for j in range(len(df_1)):
        if(df_1['items'][j]==t[0]):
            #df.concat(df_2['freq'][j]/df_1['frequency'][i])
            #print('a')
            l.append(df_2['freq'][j]/df_1['frequency'][i])


    #print(df_1[df_1['items']][0])
    #df.append()
```

[376]:
```
z=pd.DataFrame(l,columns=['Confidence'])
```

[379]:
```
df_2=df_2.append(z)
```

```
C:\Users\Admin\AppData\Local\Temp\ipykernel_37620\4253449237.py:1:
```

[352]:

	items	freq
0	(burgers, eggs)	2
1	(eggs, turkey)	3
2	(eggs, mineral water)	4
3	(eggs, chocolate)	3
4	(eggs, chicken)	2
5	(turkey, mineral water)	2
6	(avocado, soup)	2
7	(mineral water, milk)	2
8	(mineral water, energy bar)	2
9	(mineral water, whole wheat rice)	2
10	(mineral water, green tea)	2
11	(mineral water, frozen vegetables)	2
12	(mineral water, spaghetti)	5
13	(mineral water, salmon)	3
14	(mineral water, chocolate)	2
15	(mineral water, chicken)	3
16	(mineral water, fresh tuna)	2
17	(mineral water, black tea)	2
18	(mineral water, ground beef)	2
19	(mineral water, frozen smoothie)	2
20	(mineral water, escalope)	2
21	(milk, energy bar)	2
22	(milk, spaghetti)	2
23	(green tea, spaghetti)	3
24	(green tea, body spray)	2
25	(green tea, pancakes)	2
26	(green tea, brownies)	2
27	(soup, hot dogs)	2
28	(frozen vegetables, spaghetti)	3
29	(spaghetti, salmon)	3
30	(spaghetti, fresh tuna)	2
31	(spaghetti, black tea)	2
32	(spaghetti, frozen smoothie)	2
33	(spaghetti, escalope)	3
34	(salmon, black tea)	2
35	(salmon, escalope)	2
36	(shrimp, chocolate)	2
37	(shrimp, honey)	2
38	(shrimp, pasta)	2
39	(chocolate, chicken)	2
40	(body spray, pancakes)	2
41	(frozen smoothie, escalope)	2

[251]:
```
k2=z.cartesian_product(3)
```

[252]:
```
k2
```

[252]:
```
{}
```

[253]:
```
g=pd.DataFrame(k.items(),columns=['items','freq'])
```

[254]:
```
g[g['freq']>=2]
```

[254]:

	items	freq
1	(burgers, eggs)	2
19	(eggs, turkey)	3
21	(eggs, mineral water)	4
27	(eggs, chocolate)	3
28	(eggs, chicken)	2
34	(turkey, mineral water)	2
46	(avocado, soup)	2
57	(mineral water, milk)	2
58	(mineral water, energy bar)	2
59	(mineral water, whole wheat rice)	2
60	(mineral water, green tea)	2
62	(mineral water, frozen vegetables)	2
63	(mineral water, spaghetti)	5
65	(mineral water, salmon)	3
66	(mineral water, chocolate)	2
67	(mineral water, chicken)	3
69	(mineral water, fresh tuna)	2
70	(mineral water, black tea)	2
74	(mineral water, ground beef)	2
75	(mineral water, frozen smoothie)	2
76	(mineral water, escalope)	2
77	(milk, energy bar)	2
82	(milk, spaghetti)	2
108	(green tea, spaghetti)	3
112	(green tea, body spray)	2
113	(green tea, pancakes)	2
117	(green tea, brownies)	2
136	(soup, hot dogs)	2
138	(frozen vegetables, spaghetti)	3
146	(spaghetti, salmon)	3
150	(spaghetti, fresh tuna)	2
151	(spaghetti, black tea)	2
156	(spaghetti, frozen smoothie)	2
157	(spaghetti, escalope)	3
168	(salmon, black tea)	2
171	(salmon, escalope)	2
173	(shrimp, chocolate)	2
175	(shrimp, honey)	2
177	(shrimp, pasta)	2
179	(chocolate, chicken)	2

```
FutureWarning: The frame.append method is deprecated and will be removed from
pandas in a future version. Use pandas.concat instead.
  df_2=df_2.append(z)
```

[382]:
```
df_2
```

[382]:
```
                    items  freq  Confidence
0         (burgers, eggs)   2.0         NaN
1          (eggs, turkey)   3.0         NaN
2   (eggs, mineral water)   4.0         NaN
3       (eggs, chocolate)   3.0         NaN
4         (eggs, chicken)   2.0         NaN
..                    ...   ...         ...
32                    NaN   NaN         1.5
33                    NaN   NaN         0.5
34                    NaN   NaN         1.0
35                    NaN   NaN         1.0
36                    NaN   NaN         1.0

[79 rows x 3 columns]
```

[373]:
```
z=z[z['Confidence']<1]
```

[374]:
```
z
```

[374]:
```
    Confidence
1     0.444444
2     0.800000
3     0.666667
4     0.200000
5     0.500000
6     0.666667
8     0.285714
10    0.285714
11    0.500000
13    0.500000
14    0.166667
15    0.666667
18    0.500000
19    0.666667
20    0.333333
21    0.500000
22    0.500000
24    0.666667
27    0.666667
33    0.500000
```

[355]:
```
s1
```


Result :

Thus Apriori algorithm was implemented and the confidence was computed accordingly.