**Q2. Design and Implement at least four Un-Supervised learning algorithms using Python Scikit -learn for any domain like finance, health care or area of interest of your choice.(III Year Section –A Students)**

**1.) Hierarchical Clustering in Python with scikit-learn**

**About the algorithm:**

Hierarchical clustering starts by treating each observation as a separate cluster. Then, it repeatedly executes the following two steps: (1) identify the two clusters that are closest together

(2) merge the two most similar clusters. This iterative process continues until all the clusters are merged together.

**About the dataset:**

The dataset is the mall customer dataset from Kaggle. The dataset has some basic data about customers like Customer ID, age, gender, annual income and spending score. Spending Score of the customer is based on parameters like customer behaviour and purchasing data. Market basket analysis needs understanding the customers like who can be easily converge [Target Customers] so that the sense can be given to marketing team and plan the strategy accordingly.

**Code:**

# Hierarchical Clustering

# Importing the libraries

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

# Importing the dataset

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

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

# Using the dendrogram to find the optimal number of clusters

import scipy.cluster.hierarchy as sch

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

plt.title('Dendrogram')

plt.xlabel('Customers')

plt.ylabel('Euclidean distances')

plt.show()

# Training the Hierarchical Clustering model on the dataset

from sklearn.cluster import AgglomerativeClustering

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

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

# Visualising the clusters

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

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

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

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

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

plt.title('Clusters of customers')

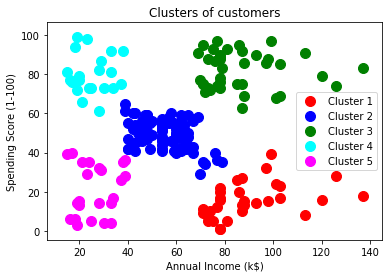
plt.xlabel('Annual Income (k$)')

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

plt.legend()

plt.show()

**Output:**



**2.) K-Means Clustering in Python with scikit-learn**

**About the algorithm:**

This algorithm will help to tackle unlabelled datasets (i.e. the datasets that do not have any class-labels). K-Means falls under the category of centroid-based clustering. So, the algorithm works by:

1. Taking any two centroids or data points initially.

2. After choosing the centroids, (say C1 and C2) the data points are assigned to any of the Clusters (let us take centroids = clusters for the time being) depending upon the distance between them and the centroids.

3. The algorithm will continue updating cluster centroids until they cannot be updated anymore. The updating takes place in the following manner:



(where n = number of objects belonging to that particular cluster)

**About the dataset:**

The dataset is used is the Loan Prediction dataset. There are altogether 13 columns in our data set. Of them Loan\_Status is the response variable and rest all are the variables /factors that decide the approval of the loan or not. The variables are, Loan ID, Gender,Married, Dependents, Education, Self\_Employed, Applicant Income, Co Applicant income, Loan Amount, Loan\_Amount\_Term, Credit\_History, Property\_Area,Loan\_Status.

**Code:**

import pandas as pd

import numpy as np

import random as rd

import matplotlib.pyplot as plt

data = pd.read\_csv('clustering.csv')

data.head()

X = data[["LoanAmount","ApplicantIncome"]]

#Visualise data points

plt.scatter(X["ApplicantIncome"],X["LoanAmount"],c='black')

plt.xlabel('AnnualIncome')

plt.ylabel('Loan Amount (In Thousands)')

plt.show()

#number of clusters

K=3

# Select random observation as centroids

Centroids = (X.sample(n=K))

plt.scatter(X["ApplicantIncome"],X["LoanAmount"],c='black')

plt.scatter(Centroids["ApplicantIncome"],Centroids["LoanAmount"],c='red')

plt.xlabel('AnnualIncome')

plt.ylabel('Loan Amount (In Thousands)')

plt.show()

# Step 3 - Assign all the points to the closest cluster centroid

# Step 4 - Recompute centroids of newly formed clusters

# Step 5 - Repeat step 3 and 4

diff = 1

j=0

while(diff!=0):

XD=X

i=1

for index1,row\_c in Centroids.iterrows():

ED=[]

for index2,row\_d in XD.iterrows():

d1=(row\_c["ApplicantIncome"]-row\_d["ApplicantIncome"])\*\*2

d2=(row\_c["LoanAmount"]-row\_d["LoanAmount"])\*\*2

d=np.sqrt(d1+d2)

ED.append(d)

X[i]=ED

i=i+1

C=[]

for index,row in X.iterrows():

min\_dist=row[1]

pos=1

for i in range(K):

if row[i+1] < min\_dist:

min\_dist = row[i+1]

pos=i+1

C.append(pos)

X["Cluster"]=C

Centroids\_new = X.groupby(["Cluster"]).mean()[["LoanAmount","ApplicantIncome"]]

if j == 0:

diff=1

j=j+1

else:

diff = (Centroids\_new['LoanAmount'] - Centroids['LoanAmount']).sum() + (Centroids\_new['ApplicantIncome'] - Centroids['ApplicantIncome']).sum()

print(diff.sum())

Centroids = X.groupby(["Cluster"]).mean()[["LoanAmount","ApplicantIncome"]]

color=['blue','green','cyan']

for k in range(K):

data=X[X["Cluster"]==k+1]

plt.scatter(data["ApplicantIncome"],data["LoanAmount"],c=color[k])

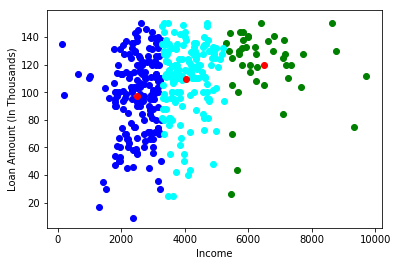
plt.scatter(Centroids["ApplicantIncome"],Centroids["LoanAmount"],c='red')

plt.xlabel('Income')

plt.ylabel('Loan Amount (In Thousands)')

plt.show()

**Output:**

****

**3.) OPTICS Clustering in Python with scikit-learn**

**About the algorithm:**

This algorithm produces a visualization of Reachability distances and uses this visualization to cluster the data. Its basic idea is similar to DBSCAN, but it addresses one of DBSCAN's major weaknesses: the problem of detecting meaningful clusters in data of varying density. To do so, the points of the database are (linearly) ordered such that spatially closest points become neighbours in the ordering.

**About the dataset:**

The dataset is the mall customer dataset from Kaggle. The dataset has some basic data about customers like Customer ID, age, gender, annual income and spending score. Spending Score of the customer is based on parameters like customer behaviour and purchasing data. Market basket analysis needs understanding the customers like who can be easily converge [Target Customers] so that the sense can be given to marketing team and plan the strategy accordingly.

**Code:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from matplotlib import gridspec

from sklearn.cluster import OPTICS, cluster\_optics\_dbscan

from sklearn.preprocessing import normalize, StandardScaler

X = pd.read\_csv('Customers.csv')

# Dropping irrelevant columns

drop\_features = ['CustomerID', 'Gender']

X = X.drop(drop\_features, axis = 1)

# Handling the missing values if any

X.fillna(method ='ffill', inplace = True)

# Scaling the data to bring all the attributes to a comparable level

scaler = StandardScaler()

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

# Normalizing the data so that the data

# approximately follows a Gaussian distribution

X\_normalized = normalize(X\_scaled)

# Converting the numpy array into a pandas DataFrame

X\_normalized = pd.DataFrame(X\_normalized)

# Renaming the columns

X\_normalized.columns = X.columns

# Building the OPTICS Clustering model

optics\_model = OPTICS(min\_samples = 10, xi = 0.05, min\_cluster\_size = 0.05)

# Training the model

optics\_model.fit(X\_normalized)

# Producing the labels according to the DBSCAN technique with eps = 0.5

labels1 = cluster\_optics\_dbscan(reachability = optics\_model.reachability\_, core\_distances = optics\_model.core\_distances\_, ordering = optics\_model.ordering\_, eps = 0.5)

# Producing the labels according to the DBSCAN technique with eps = 2.0

labels2 = cluster\_optics\_dbscan(reachability = optics\_model.reachability\_, core\_distances = optics\_model.core\_distances\_, ordering = optics\_model.ordering\_, eps = 2)

# Creating a numpy array with numbers at equal spaces till

# the specified range

space = np.arange(len(X\_normalized))

# Storing the reachability distance of each point

reachability = optics\_model.reachability\_[optics\_model.ordering\_]

# Storing the cluster labels of each point

labels = optics\_model.labels\_[optics\_model.ordering\_]

# Defining the framework of the visualization

plt.figure(figsize =(10, 7))

G = gridspec.GridSpec(2, 3)

ax1 = plt.subplot(G[0, :])

ax2 = plt.subplot(G[1, 0])

# Plotting the Reachability-Distance Plot

colors = ['c.', 'b.', 'r.', 'y.', 'g.']

for Class, colour in zip(range(0, 5), colors):

Xk = space[labels == Class]

Rk = reachability[labels == Class]

ax1.plot(Xk, Rk, colour, alpha = 0.3)

ax1.plot(space[labels == -1], reachability[labels == -1], 'k.', alpha = 0.3)

ax1.plot(space, np.full\_like(space, 2., dtype = float), 'k-', alpha = 0.5)

ax1.plot(space, np.full\_like(space, 0.5, dtype = float), 'k-.', alpha = 0.5)

ax1.set\_ylabel('Reachability Distance')

ax1.set\_title('Reachability Plot')

# Plotting the OPTICS Clustering

colors = ['c.', 'b.', 'r.', 'y.', 'g.']

for Class, colour in zip(range(0, 5), colors):

Xk = X\_normalized[optics\_model.labels\_ == Class]

ax2.plot(Xk.iloc[:, 0], Xk.iloc[:, 1], colour, alpha = 0.3)

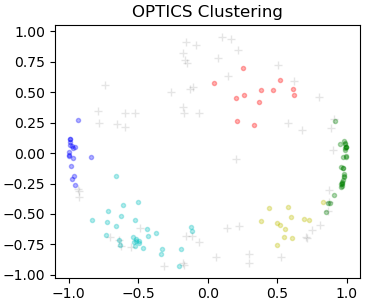
ax2.plot(X\_normalized.iloc[optics\_model.labels\_ == -1, 0],

X\_normalized.iloc[optics\_model.labels\_ == -1, 1],

'k+', alpha = 0.1)

ax2.set\_title('OPTICS Clustering')

**Output:**



**4.) Principle Component analysis (PCA) in Python with scikit-learn**

**About the algorithm:**

Principal component analysis (PCA) is an unsupervised technique used to pre-process and reduce the dimensionality of high-dimensional datasets while preserving the original structure and relationships inherent to the original dataset so that machine learning models can still learn from them and be used to make accurate predictions. PCA can be thought of as a clustering algorithm not unlike other clustering methods, such as k-means clustering. Once the principal components are computed, a low-dimensional view of the data can be generated by plotting the first principal component (PC1) against the second principal component (PC2).

A simple algorithm for PCA is as follows:

* Normalize the Data - This is often necessary if the features in your feature set are measured in different units. A common way of normalizing the features in your data set is to convert them to z-scores.
* Calculate the covariance matrix.
* Compute the eigen values and vectors.
* Re-orient the data.
* Plot the data (PC1 against PC2)

**About the dataset:**

The dataset used is the Wine Dataset. The dataset has information related to wine like Alcohol, Malic\_Acid, Ash, Ash\_Alcanity, Magnesium, Total\_Phenols, Flavanoids, Nonflavanoid\_Phenols, Proanthocyanins, Color\_Intensity, Hue, OD280, Proline, Customer\_Segment. Dropping the 'Customer\_Segment' column from the dataset makes it unlabeled. It's the task of PCA to cluster the records of the datasets into different segements.

**Code:**

# Principal Component Analysis (PCA)

# Importing the libraries

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

# Importing the dataset

dataset = pd.read\_csv('Wine.csv')

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

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

# Splitting the dataset into the Training set and Test set

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

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

# Feature Scaling

from sklearn.preprocessing import StandardScaler

sc = StandardScaler()

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

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

# Applying PCA

from sklearn.decomposition import PCA

pca = PCA(n\_components = 2)

X\_train = pca.fit\_transform(X\_train)

X\_test = pca.transform(X\_test)

# Training the Logistic Regression model on the Training set

from sklearn.linear\_model import LogisticRegression

classifier = LogisticRegression(random\_state = 0)

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

# Making the Confusion Matrix

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

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

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

print(cm)

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

# Visualising the Training set results

from matplotlib.colors import ListedColormap

X\_set, y\_set = X\_train, y\_train

X1, X2 = np.meshgrid(np.arange(start = X\_set[:, 0].min() - 1, stop = X\_set[:, 0].max() + 1, step = 0.01),

np.arange(start = X\_set[:, 1].min() - 1, stop = X\_set[:, 1].max() + 1, step = 0.01))

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

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

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

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

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

plt.scatter(X\_set[y\_set == j, 0], X\_set[y\_set == j, 1],

c = ListedColormap(('red', 'green', 'blue'))(i), label = j)

plt.title('Logistic Regression (Training set)')

plt.xlabel('PC1')

plt.ylabel('PC2')

plt.legend()

plt.show()

# Visualising the Test set results

from matplotlib.colors import ListedColormap

X\_set, y\_set = X\_test, y\_test

X1, X2 = np.meshgrid(np.arange(start = X\_set[:, 0].min() - 1, stop = X\_set[:, 0].max() + 1, step = 0.01),

np.arange(start = X\_set[:, 1].min() - 1, stop = X\_set[:, 1].max() + 1, step = 0.01))

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

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

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

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

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

plt.scatter(X\_set[y\_set == j, 0], X\_set[y\_set == j, 1],

c = ListedColormap(('red', 'green', 'blue'))(i), label = j)

plt.title('Logistic Regression (Test set)')

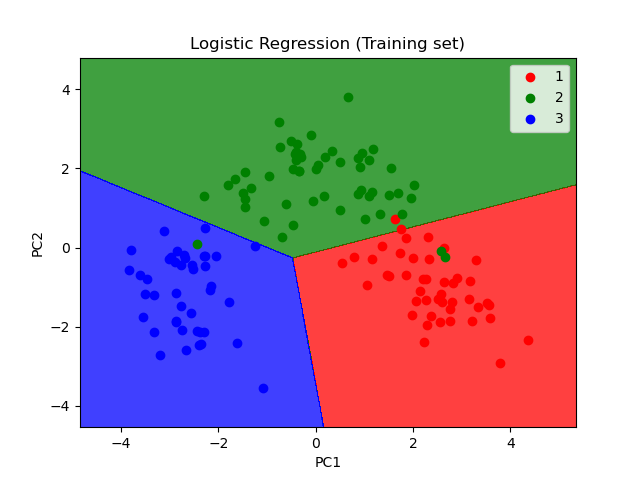
plt.xlabel('PC1')

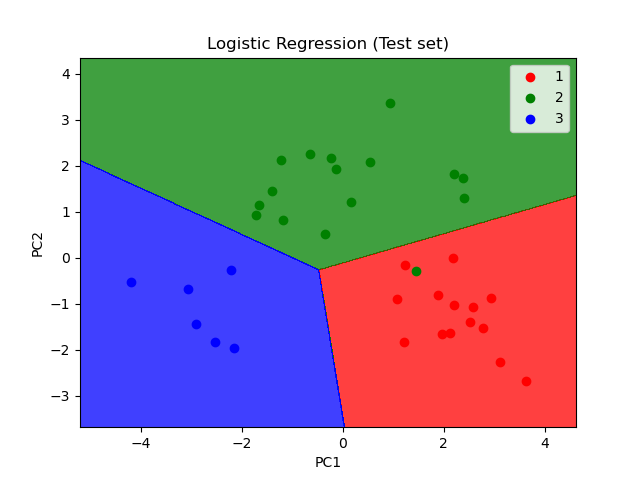
plt.ylabel('PC2')

plt.legend()

plt.show()

**Output:**

****



**Q3. Design and Implement at least two algorithms each for Time series analysis and Predictive analytics (Mandatory assignment for both sections).**

**1.) Time Series Analysis:**

Time series analysis is a method for analyzing time series data, which is a [sequence](https://en.wikipedia.org/wiki/Sequence) taken at successive equally spaced points in time, in order to extract meaningful statistics and other characteristics of the data. Time series forecasting is the use of a [model](https://en.wikipedia.org/wiki/Model_(abstract)) to predict future values based on previously observed values.

**First Algorithm: ARIMA:**

ARIMA, short for 'AutoRegressive Integrated Moving Average', is a forecasting algorithm based on the idea that the information in the past values of the time series can alone be used to predict the future values. In our code we are using the arima model to predict the number of passengers for a particular airline from the year 1949 to 1960.

**About the Dataset:**

The dataset used is the AirPassengers Dataset. The dataset is a csv file with two columns. The first column includes the months of all years from 1949 to 1960 and the second column includes the number of passengers that travelled by that airline in that month of the year.

**Code:**

# -\*- coding: utf-8 -\*-

import numpy as np

import pandas as pd

#import matplotlib.pylot as plt

from statsmodels.tsa.seasonal import seasonal\_decompose

# Read the AirPassengers dataset

airline = pd.read\_csv('AirPassengers.csv',

index\_col ='Month',

parse\_dates = True)

# Print the first five rows of the dataset

airline.head()

# ETS Decomposition

result = seasonal\_decompose(airline['#Passengers'],

model ='multiplicative')

# ETS plot

result.plot()

# Import the library

from pmdarima import auto\_arima

# Ignore harmless warnings

import warnings

warnings.filterwarnings("ignore")

# Fit auto\_arima function to AirPassengers dataset

stepwise\_fit = auto\_arima(airline['#Passengers'], start\_p = 1, start\_q = 1,

max\_p = 3, max\_q = 3, m = 12,

start\_P = 0, seasonal = True,

d = None, D = 1, trace = True,

error\_action ='ignore', # we don't want to know if an order does not work

suppress\_warnings = True, # we don't want convergence warnings

stepwise = True) # set to stepwise

# To print the summary

stepwise\_fit.summary()

# Split data into train / test sets

train = airline.iloc[:len(airline)-12]

test = airline.iloc[len(airline)-12:] # set one year(12 months) for testing

# Fit a SARIMAX(0, 1, 1)x(2, 1, 1, 12) on the training set

from statsmodels.tsa.statespace.sarimax import SARIMAX

model = SARIMAX(train['#Passengers'],

order = (0, 1, 1),

seasonal\_order =(2, 1, 1, 12))

result = model.fit()

result.summary()

start = len(train)

end = len(train) + len(test) - 1

# Predictions for one-year against the test set

predictions = result.predict(start, end,

typ = 'levels').rename("Predictions")

# plot predictions and actual values

predictions.plot(legend = True)

test['#Passengers'].plot(legend = True)

from sklearn.metrics import mean\_squared\_error

from statsmodels.tools.eval\_measures import rmse

# Calculate root mean squared error

rmse(test["#Passengers"], predictions)

# Calculate mean squared error

mean\_squared\_error(test["#Passengers"], predictions)

# Train the model on the full dataset

model = model = SARIMAX(airline['#Passengers'],

order = (0, 1, 1),

seasonal\_order =(2, 1, 1, 12))

result = model.fit()

# Forecast for the next 3 years

forecast = result.predict(start = len(airline),

end = (len(airline)-1) + 3 \* 12,

typ = 'levels').rename('Forecast')

# Plot the forecast values

airline['#Passengers'].plot(figsize = (12, 5), legend = True)

forecast.plot(legend = True)

**Output:**

Performing stepwise search to minimize aic

Fit ARIMA(1,1,1)x(0,1,1,12) [intercept=True]; AIC=1024.824, BIC=1039.200, Time=0.703 seconds

Fit ARIMA(0,1,0)x(0,1,0,12) [intercept=True]; AIC=1033.479, BIC=1039.229, Time=0.018 seconds

Fit ARIMA(1,1,0)x(1,1,0,12) [intercept=True]; AIC=1022.316, BIC=1033.817, Time=0.510 seconds

Fit ARIMA(0,1,1)x(0,1,1,12) [intercept=True]; AIC=1022.904, BIC=1034.405, Time=0.605 seconds

Fit ARIMA(0,1,0)x(0,1,0,12) [intercept=False]; AIC=1031.508, BIC=1034.383, Time=0.016 seconds

Fit ARIMA(1,1,0)x(0,1,0,12) [intercept=True]; AIC=1022.343, BIC=1030.968, Time=0.100 seconds

Fit ARIMA(1,1,0)x(2,1,0,12) [intercept=True]; AIC=1021.137, BIC=1035.513, Time=1.886 seconds

Fit ARIMA(1,1,0)x(2,1,1,12) [intercept=True]; AIC=1017.167, BIC=1034.418, Time=6.589 seconds

Near non-invertible roots for order (1, 1, 0)(2, 1, 1, 12); setting score to inf (at least one inverse root too close to the border of the unit circle: 0.998)

Fit ARIMA(1,1,0)x(1,1,1,12) [intercept=True]; AIC=1022.410, BIC=1036.786, Time=0.992 seconds

Fit ARIMA(0,1,0)x(2,1,0,12) [intercept=True]; AIC=1034.067, BIC=1045.568, Time=1.330 seconds

Fit ARIMA(2,1,0)x(2,1,0,12) [intercept=True]; AIC=1023.007, BIC=1040.258, Time=2.355 seconds

Fit ARIMA(1,1,1)x(2,1,0,12) [intercept=True]; AIC=1022.905, BIC=1040.156, Time=2.841 seconds

Fit ARIMA(0,1,1)x(2,1,0,12) [intercept=True]; AIC=1021.017, BIC=1035.393, Time=1.355 seconds

Fit ARIMA(0,1,1)x(1,1,0,12) [intercept=True]; AIC=1022.314, BIC=1033.815, Time=0.339 seconds

Fit ARIMA(0,1,1)x(2,1,1,12) [intercept=True]; AIC=1015.842, BIC=1033.093, Time=7.258 seconds

Near non-invertible roots for order (0, 1, 1)(2, 1, 1, 12); setting score to inf (at least one inverse root too close to the border of the unit circle: 0.998)

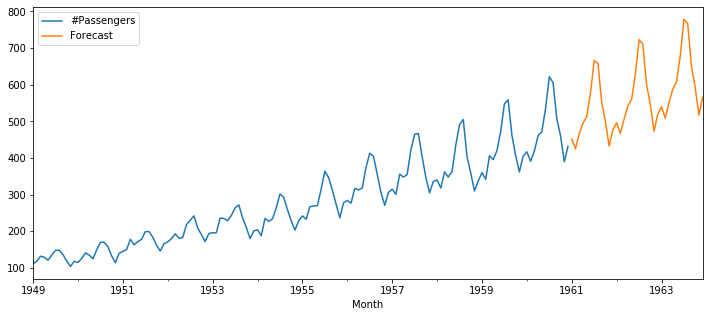
Fit ARIMA(0,1,1)x(1,1,1,12) [intercept=True]; AIC=1022.207, BIC=1036.583, Time=1.196 seconds

Fit ARIMA(0,1,2)x(2,1,0,12) [intercept=True]; AIC=1022.996, BIC=1040.247, Time=1.989 seconds

Fit ARIMA(1,1,2)x(2,1,0,12) [intercept=True]; AIC=1024.668, BIC=1044.795, Time=3.767 seconds

Total fit time: 34.234 secondsA screenshot of a cell phone

Description automatically generated



A screenshot of a cell phone

Description automatically generated

**Second Algorithm: Random Forest:**

Random Forest algorithm is a supervised classification algorithm. We can see it from its name, which is to create a forest by some way and make it random. There is a direct relationship between the number of trees in the forest and the results it can get: the larger the number of trees, the more accurate the result.

**About the Dataset:**

The dataset used is the Open Power System Data, which is publicly available. It contains in csv format, the electricity consumption, wind power consumption and solar power production for 2006 – 2017.

**Code:**

### General import

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn import preprocessing

import sklearn.metrics as metrics

from sklearn.model\_selection import TimeSeriesSplit

from sklearn.model\_selection import GridSearchCV

from sklearn.linear\_model import LinearRegression

from sklearn.neural\_network import MLPRegressor

from sklearn.neighbors import KNeighborsRegressor

from sklearn.ensemble import RandomForestRegressor

from sklearn.svm import SVR

from sklearn.model\_selection import cross\_val\_score

from sklearn.metrics import make\_scorer

def rmse(actual, predict):

predict = np.array(predict)

actual = np.array(actual)

distance = predict - actual

square\_distance = distance \*\* 2

mean\_square\_distance = square\_distance.mean()

score = np.sqrt(mean\_square\_distance)

return score

def regression\_results(y\_true, y\_pred):

# Regression metrics

explained\_variance=metrics.explained\_variance\_score(y\_true, y\_pred)

mean\_absolute\_error=metrics.mean\_absolute\_error(y\_true, y\_pred)

mse=metrics.mean\_squared\_error(y\_true, y\_pred)

mean\_squared\_log\_error=metrics.mean\_squared\_log\_error(y\_true, y\_pred)

median\_absolute\_error=metrics.median\_absolute\_error(y\_true, y\_pred)

r2=metrics.r2\_score(y\_true, y\_pred)

print('explained\_variance: ', round(explained\_variance,4))

print('mean\_squared\_log\_error: ', round(mean\_squared\_log\_error,4))

print('r2: ', round(r2,4))

print('MAE: ', round(mean\_absolute\_error,4))

print('MSE: ', round(mse,4))

print('RMSE: ', round(np.sqrt(mse),4))

data = pd.read\_csv('daily.csv',sep=",")

# to explicitly convert the date column to type DATETIME

data['Date'] = pd.to\_datetime(data['Date'])

data = data.set\_index('Date')

# creating new dataframe from consumption column

data\_consumption = data[['Consumption']]

# inserting new column with yesterday's consumption values

data\_consumption.loc[:,'Yesterday'] = data\_consumption.loc[:,'Consumption'].shift()

# inserting another column with difference between yesterday and day before yesterday's consumption values.

data\_consumption.loc[:,'Yesterday\_Diff'] = data\_consumption.loc[:,'Yesterday'].diff()

# dropping NAs

data\_consumption = data\_consumption.dropna()

X\_train = data\_consumption[:'2016'].drop(['Consumption'], axis = 1)

y\_train = data\_consumption.loc[:'2016', 'Consumption']

X\_test = data\_consumption['2017'].drop(['Consumption'], axis = 1)

y\_test = data\_consumption.loc['2017', 'Consumption']

# Spot Check Algorithms

models = []

models.append(('LR', LinearRegression()))

models.append(('NN', MLPRegressor(solver = 'lbfgs'))) #neural network

models.append(('KNN', KNeighborsRegressor()))

models.append(('RF', RandomForestRegressor(n\_estimators = 10))) # Ensemble method - collection of many decision trees

models.append(('SVR', SVR(gamma='auto'))) # kernel = linear

# Evaluate each model in turn

results = []

names = []

for name, model in models:

# TimeSeries Cross validation

tscv = TimeSeriesSplit(n\_splits=10)

cv\_results = cross\_val\_score(model, X\_train, y\_train, cv=tscv, scoring='r2')

results.append(cv\_results)

names.append(name)

print('%s: %f (%f)' % (name, cv\_results.mean(), cv\_results.std()))

# Compare Algorithms

plt.boxplot(results, labels=names)

plt.title('Algorithm Comparison')

plt.show()

rmse\_score = make\_scorer(rmse, greater\_is\_better = False)

model = RandomForestRegressor()

param\_search = {

'n\_estimators': [20, 50, 100],

'max\_features': ['auto', 'sqrt', 'log2'],

'max\_depth' : [i for i in range(5,15)]

}

tscv = TimeSeriesSplit(n\_splits=10)

gsearch = GridSearchCV(estimator=model, cv=tscv, param\_grid=param\_search, scoring = rmse\_score)

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

best\_score = gsearch.best\_score\_

best\_model = gsearch.best\_estimator\_

y\_true = y\_test.values

y\_pred = best\_model.predict(X\_test)

print("Performance :")

regression\_results(y\_true, y\_pred)

print("Performance After Feature Engineering :")

# creating copy of original dataframe

data\_consumption\_2o = data\_consumption.copy()

# inserting column with yesterday-1 values

data\_consumption\_2o['Yesterday-1'] = data\_consumption\_2o['Yesterday'].shift()

# inserting column with difference in yesterday-1 and yesterday-2 values.

data\_consumption\_2o['Yesterday-1\_Diff'] = data\_consumption\_2o['Yesterday-1'].diff()

# dropping NAs

data\_consumption\_2o = data\_consumption\_2o.dropna()

X\_train\_2o = data\_consumption\_2o[:'2016'].drop(['Consumption'], axis = 1)

y\_train\_2o = data\_consumption\_2o.loc[:'2016', 'Consumption']

X\_test = data\_consumption\_2o['2017'].drop(['Consumption'], axis = 1)

y\_test = data\_consumption\_2o.loc['2017', 'Consumption']

model = RandomForestRegressor()

param\_search = {

'n\_estimators': [20, 50, 100],

'max\_features': ['auto', 'sqrt', 'log2'],

'max\_depth' : [i for i in range(5,15)]

}

tscv = TimeSeriesSplit(n\_splits=10)

gsearch = GridSearchCV(estimator=model, cv=tscv, param\_grid=param\_search, scoring = rmse\_score)

gsearch.fit(X\_train\_2o, y\_train\_2o)

best\_score = gsearch.best\_score\_

best\_model = gsearch.best\_estimator\_

y\_true = y\_test.values

y\_pred = best\_model.predict(X\_test)

regression\_results(y\_true, y\_pred)

print("Better Performance :")

data\_consumption\_2o\_solar = data\_consumption\_2o.join(data[['Solar']])

data\_consumption\_2o\_solar = data\_consumption\_2o\_solar.dropna()

X\_train\_2o\_solar = data\_consumption\_2o\_solar[:'2016'].drop(['Consumption'], axis = 1)

y\_train\_2o\_solar = data\_consumption\_2o\_solar.loc[:'2016', 'Consumption']

X\_test = data\_consumption\_2o\_solar['2017'].drop(['Consumption'], axis = 1)

y\_test = data\_consumption\_2o\_solar.loc['2017', 'Consumption']

model = RandomForestRegressor()

param\_search = {

'n\_estimators': [20, 50, 100],

'max\_features': ['auto', 'sqrt', 'log2'],

'max\_depth' : [i for i in range(5,15)]

}

tscv = TimeSeriesSplit(n\_splits=5)

gsearch = GridSearchCV(estimator=model, cv=tscv, param\_grid=param\_search, scoring = rmse\_score)

gsearch.fit(X\_train\_2o\_solar, y\_train\_2o\_solar)

best\_score = gsearch.best\_score\_

best\_model = gsearch.best\_estimator\_

y\_true = y\_test.values

y\_pred = best\_model.predict(X\_test)

regression\_results(y\_true, y\_pred)

**Output:**

LR: 0.343747 (0.029773)

NN: 0.488427 (0.107526)

KNN: 0.646307 (0.083185)

RF: 0.621847 (0.088466)

SVR: -0.117944 (0.157755)

Performance :

('explained\_variance: ', 0.7323)

('mean\_squared\_log\_error: ', 0.0041)

('r2: ', 0.73)

('MAE: ', 55.499)

('MSE: ', 7312.6016)

('RMSE: ', 85.5138)

Performance After Feature Engineering :

('explained\_variance: ', 0.8397)

('mean\_squared\_log\_error: ', 0.0025)

('r2: ', 0.8397)

('MAE: ', 40.6433)

('MSE: ', 4341.2756)

('RMSE: ', 65.8884)

Better Performance :

('explained\_variance: ', 0.8373)

('mean\_squared\_log\_error: ', 0.0025)

('r2: ', 0.837)

('MAE: ', 40.7168)

('MSE: ', 4415.4563)

('RMSE: ', 66.4489)

A screenshot of a cell phone

Description automatically generated

**2.) Predictive Analytics:**

Predictive analytics encompasses a variety of statistical techniques from data mining, predictive modelling, and machine learning, that analyze current and historical facts to make predictions about future or otherwise unknown events.

**First Algorithm: Linear Regression Model:**  
Linear regression is a statistical method that analyzes and finds relationships between two variables. In predictive analytics it can be used to predict a future numerical value of a variable. Linear regression is most suitable for linear data. But it is extremely sensitive toward outliers in the data points.

**About the Dataset:**

This dataset is originally from the National Institute of Diabetes and Digestive and Kidney Diseases. The objective of the dataset is to diagnostically predict whether a patient has diabetes, based on certain diagnostic measurements included in the dataset. Several constraints were placed on the selection of these instances from a larger database. All patients here are females at least 21 years old of Pima Indian heritage.

**Code:**

print(\_\_doc\_\_)

import matplotlib.pyplot as plt

import numpy as np

from sklearn import datasets, linear\_model

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

# Load the diabetes dataset

diabetes\_X, diabetes\_y = datasets.load\_diabetes(return\_X\_y=True)

# Use only one feature

diabetes\_X = diabetes\_X[:, np.newaxis, 2]

# Split the data into training/testing sets

diabetes\_X\_train = diabetes\_X[:-20]

diabetes\_X\_test = diabetes\_X[-20:]

# Split the targets into training/testing sets

diabetes\_y\_train = diabetes\_y[:-20]

diabetes\_y\_test = diabetes\_y[-20:]

# Create linear regression object

regr = linear\_model.LinearRegression()

# Train the model using the training sets

regr.fit(diabetes\_X\_train, diabetes\_y\_train)

# Make predictions using the testing set

diabetes\_y\_pred = regr.predict(diabetes\_X\_test)

# The coefficients

print('Coefficients: \n', regr.coef\_)

# The mean squared error

print('Mean squared error: %.2f'

% mean\_squared\_error(diabetes\_y\_test, diabetes\_y\_pred))

# The coefficient of determination: 1 is perfect prediction

print('Coefficient of determination: %.2f'

% r2\_score(diabetes\_y\_test, diabetes\_y\_pred))

# Plot outputs

plt.scatter(diabetes\_X\_test, diabetes\_y\_test, color='black')

plt.plot(diabetes\_X\_test, diabetes\_y\_pred, color='blue', linewidth=3)

plt.xticks(())

plt.yticks(())

plt.show()

**Output:**

Coefficients: [938.23786125]

Mean squared error: 2548.07

Coefficient of determination: 0.47

A picture containing text, photo, different

Description automatically generated

**Second Algorithm: K-Means:**

This algorithm will help to tackle unlabelled datasets (i.e. the datasets that do not have any class-labels). K-Means falls under the category of centroid-based clustering. So, the algorithm works by:

1. Taking any two centroids or data points initially.

2. After choosing the centroids, (say C1 and C2) the data points are assigned to any of the Clusters (let us take centroids = clusters for the time being) depending upon the distance between them and the centroids.

3. The algorithm will continue updating cluster centroids until they cannot be updated anymore. The updating takes place in the following manner:



(where n = number of objects belonging to that cluster)

**About the Dataset:**

The dataset used is the Titanic Dataset. The sinking of the RMS Titanic is one of the most infamous shipwrecks in history. Although there was some element of luck involved in surviving the sinking, some groups of people were more likely to survive than others, such as women, children, and the upper-class. In the dataset, the training set contains several records about the passengers of Titanic. It has 12 features capturing information about PassengerId, Survived, Passenger class, Name, Sex, Age, SibSp, Parch, Ticket, Fare, Cabin, Port of Embarkation. The dataset's label is ‘survival’ which denotes the survivial status of a passenger. The records can be clustered into two i.e. the ones who survived and the ones who did not. Dropping the 'survival' column from the dataset makes it unlabeled. It's the task of K-Means to cluster the records of the datasets if they survived or not.

**Code:**

# Dependencies

import pandas as pd

import numpy as np

from sklearn.cluster import KMeans

from sklearn.preprocessing import LabelEncoder

from sklearn.preprocessing import MinMaxScaler

import seaborn as sns

import matplotlib.pyplot as plt

# Load the train and test datasets to create two DataFrames

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

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

#checking presence of missing values

#print(train.isna().sum())

#print(test.isna().sum())

# Fill missing values with mean column values in the train set

train.fillna(train.mean(), inplace=True)

# Fill missing values with mean column values in the test set

test.fillna(test.mean(), inplace=True)

#see the survival count of passengers with respect to the features

#train[['Pclass', 'Survived']].groupby(['Pclass'],

as\_index=False).mean().sort\_values(by='Survived', ascending=False)

#train.info()

train = train.drop(['Name','Ticket', 'Cabin','Embarked'], axis=1)

test = test.drop(['Name','Ticket', 'Cabin','Embarked'], axis=1)

labelEncoder = LabelEncoder()

labelEncoder.fit(train['Sex'])

labelEncoder.fit(test['Sex'])

train['Sex'] = labelEncoder.transform(train['Sex'])

test['Sex'] = labelEncoder.transform(test['Sex'])

# Let's investigate if you have non-numeric data left

#train.info()

#test.info()

X = np.array(train.drop(['Survived'], 1).astype(float))

y = np.array(train['Survived'])

#print(X.shape)

#print(y)

#KMeans(algorithm='auto', copy\_x=True, init='k-means++',

max\_iter=600,n\_clusters=2, n\_init=10, n\_jobs=1,

precompute\_distances='auto',random\_state=None, tol=0.0001, verbose=0)

kmeans = KMeans(n\_clusters=2, max\_iter=600, algorithm = 'auto')

scaler = MinMaxScaler()

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

kmeans.fit(X\_scaled)

correct = 0

for i in range(len(X)):

predict\_me = np.array(X[i].astype(float))

predict\_me = predict\_me.reshape(-1, len(predict\_me))

prediction = kmeans.predict(predict\_me)

# print(prediction[0],y[i])

if (prediction[0] == y[i]):

correct += 1

#print(correct)

#print(len(X))

print("Accuracy:")

print(float(correct)/float(len(X)))

**Output:**

K-Means clustered the records of the datasets if they survived or not (0 or 1).

Accuracy : 0.626262626263