**CAPSTONE 1**

**Firstly, we import the required libraries:**

import numpy as np

import pandas as pd

import itertools

import matplotlib.pyplot as plt

import seaborn as sns

import sklearn

from sklearn.linear\_model import LinearRegression

from sklearn.linear\_model import LogisticRegression

from sklearn.svm import SVC

from sklearn.ensemble import RandomForestClassifier

from sklearn.cluster import KMeans

from sklearn.decomposition import PCA

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

from sklearn.model\_selection import GridSearchCV

from sklearn.metrics import mean\_absolute\_error

from sklearn.metrics import confusion\_matrix

from sklearn.preprocessing import StandardScaler

from sklearn.neighbors import KNeighborsClassifier

from sklearn.naive\_bayes import GaussianNB

from sklearn.tree import DecisionTreeClassifier

from xgboost import XGBClassifier

from sklearn.model\_selection import train\_test\_split, cross\_val\_score

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

**Now, Read the CSV file and load it into a DataFrame:**

df = pd.read\_csv('diabetes.csv')

**1. Find the distribution of outcome feature**

To find the distribution of the 'Outcome' feature in the dataset, we can use the value\_counts() function in pandas. This will show the count of each unique value in the 'Outcome' column, which represents the distribution of the outcome classes (0 and 1).

* outcome\_distribution = df['Outcome'].value\_counts()

print(outcome\_distribution)

This code will output the count of each unique value in the 'Outcome' column, which corresponds to the distribution of the outcome classes in the dataset.

A white background with black text

Description automatically generated

**2. Check whether it is imbalanced or balanced dataset**

A dataset is imbalanced if the classes within the dataset are not evenly distributed.

In this case, we can look at the distribution of the target variable (the 'Outcome' column) and compare the number of instances in each class.

We can calculate the imbalance ratio by dividing the count of the minority class by the count of the majority class.

* class\_distribution = df['Outcome'].value\_counts()

imbalance\_ratio = class\_distribution[1] / class\_distribution[0]

print("Imbalance Ratio:", imbalance\_ratio)

If the imbalance ratio is close to 1, the dataset is balanced.

If the imbalance ratio is significantly greater than 1 (e.g., 5, 10, or more), the dataset is considered imbalanced, with the class 1 being the minority class.

A screen shot of a computer

Description automatically generated

Here, the imbalance ratio (268 / 500) ≈ 0.536, indicating that the dataset is imbalanced, with class 1 being the minority class.

**3. Find the mean age as per the outcome**

df.groupby("Outcome").agg({"Age":"mean"})

A white background with a black border

Description automatically generated with medium confidence

**4. Find the max age as per the outcome**

df.groupby("Outcome").agg({"Age":"max"})

**A white rectangular object with a black border

Description automatically generated**

**5. Find the mean BMI as per the outcome**

df.groupby("Outcome").agg({"BMI":"mean"})

A white rectangular object with a black border

Description automatically generated with medium confidence

**6. Find the max BMI as per the outcome**

df.groupby("Outcome").agg({"BMI":"max"})

A white rectangular object with a black border

Description automatically generated

**7. Create a Pie Chart showing the distribution of Outcome Feature**

outcome\_distribution = df['Outcome'].value\_counts()

labels = ['No Diabetes (0)', 'Diabetes (1)']

sizes = outcome\_distribution.values

colors = ['lightskyblue', 'lightcoral']

explode = (0.1, 0) # explode the 1st slice (Diabetes)

plt.pie(sizes, explode=explode, labels=labels, colors=colors,

autopct='%1.1f%%', shadow=True, startangle=140)

plt.axis('equal') # Equal aspect ratio ensures that pie is drawn as a circle.

plt.title('Distribution of Outcome Feature')

plt.show()

A screenshot of a computer screen

Description automatically generated

**8. Find all the pairs of features having correlation greater than 0.3**

To calculate the correlation matrix we can use the .corr() method, and then iterates through the upper triangle of the matrix to find pairs of features that have an absolute correlation value greater than 0.3.

* correlation\_matrix = df.corr()

high\_corr\_pairs = []

for i in range(len(correlation\_matrix.columns)):

for j in range(i + 1, len(correlation\_matrix.columns)):

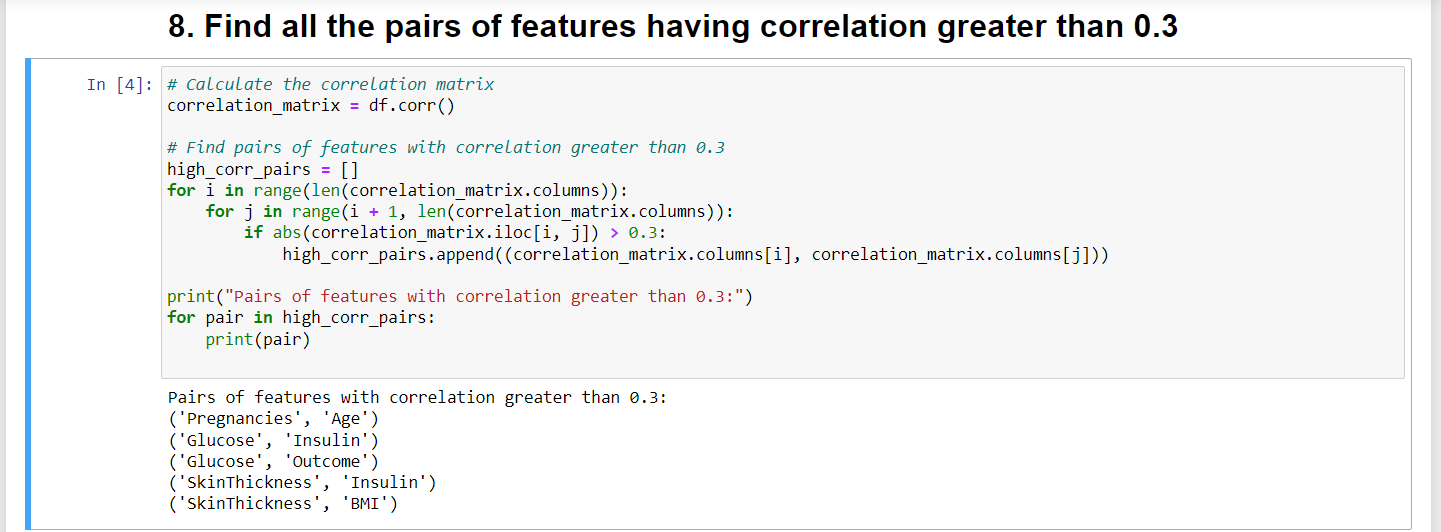
if abs(correlation\_matrix.iloc[i, j]) > 0.3:

high\_corr\_pairs.append((correlation\_matrix.columns[i], correlation\_matrix.columns[j]))

print("Pairs of features with correlation greater than 0.3:")

for pair in high\_corr\_pairs:

print(pair)



**9. Find the features in which missing values are present**

* missing\_values\_columns = df.columns[df.isnull().any()]

print("Columns with missing values:")

print(missing\_values\_columns)

A screenshot of a computer error

Description automatically generated

**10. Replace all the missing values with the mean or median value**

Here, we calculate the mean and median values for each column using the .mean() and .median() methods, respectively. After that, we use .fillna() method to replace the missing values in the dataset with either the mean or median value, creating two separate updated datasets: one filled with the mean and one filled with the median. Finally, we print the updated datasets.

* mean\_values = df.mean()

median\_values = df.median()

data\_filled\_mean = df.fillna(mean\_values)

data\_filled\_median = df.fillna(median\_values)

print("Dataset with missing values filled using mean:")

print(data\_filled\_mean)

print("\nDataset with missing values filled using median:")

print(data\_filled\_median)

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**11. Create a new Feature NewInsulinScore as per following criteria: if "Insulin" >= 16 and "Insulin" <= 166 => then It is Normal otherwise Abnormal**

* df['NewInsulinScore'] = df['Insulin'].apply(lambda x: 'Normal' if (x >= 16 and x <= 166) else 'Abnormal')

print(df)

A screenshot of a computer

Description automatically generated

**12. Create a Feature NewGlucose as per following Criteria**

* 1. **If "Glucose" <= 70 => Low**
  2. **If "Glucose" > 70 & "Glucose" <= 99 => Normal**
  3. **If "Glucose" > 99 & "Glucose" <= 126 => OverWeight**
  4. **If "Glucose" > 126 => Secret**

NewGlucose = pd.Series(["Low","Normal","OverWeight","Secret"],dtype="category")

df["NewGlucose"]= NewGlucose

df.loc[df['Glucose'] <= 70, "NewGlucose"] = NewGlucose[0]

df.loc[(df['Glucose'] > 70) &(df['Glucose'] <= 99), "NewGlucose"] = NewGlucose[1]

df.loc[(df['Glucose'] > 99) &(df['Glucose'] <= 126), "NewGlucose"] = NewGlucose[2]

df.loc[df['Glucose'] > 126, "NewGlucose"] = NewGlucose[3]

df.head()

A screenshot of a computer

Description automatically generated

**13. Create One hot encoding for NewInsulinScore and NewGlucose feature created earlier**

* data\_encoded = pd.get\_dummies(df, columns=['NewInsulinScore', 'NewGlucose'])

print(data\_encoded)

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**14. Create Training and Testing Dataset (Take 12% as test dataset)**

**a. Use following Algorithms to train on the dataset:**

* **'Logistic Regression', 'KNN', 'SVM',"Naive Bayes", 'Decision Tree Classifier', 'Random Forest Classifier', 'XgBoost'**

X = data\_encoded.drop('Outcome', axis=1)

y = data\_encoded['Outcome']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.12, random\_state=42)

models = {

'Logistic Regression': LogisticRegression(),

'KNN': KNeighborsClassifier(),

'SVM': SVC(),

'Naive Bayes': GaussianNB(),

'Decision Tree': DecisionTreeClassifier(),

'Random Forest': RandomForestClassifier(),

'XgBoost': XGBClassifier()

}

for name, model in models.items():

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

accuracy = model.score(X\_test, y\_test)

print(f'{name}: Accuracy = {accuracy:.2f}')

A white rectangular object with red border

Description automatically generated

**15. Find Value of k for which KNN Gives Best Accuracy**

* best\_accuracy = 0

best\_k = 0

for k in range(1, 21):

knn = KNeighborsClassifier(n\_neighbors=k)

accuracy\_scores = cross\_val\_score(knn, X\_train, y\_train, cv=5)

avg\_accuracy = accuracy\_scores.mean()

if avg\_accuracy > best\_accuracy:

best\_accuracy = avg\_accuracy

best\_k = k

print(f"Best value of k for KNN: {best\_k}, Best accuracy: {best\_accuracy:.2f}")

A screenshot of a computer

Description automatically generated

**16. Print all evaluation metrics for all the Algorithms(find it on Test Data) i.e Precision, Recall, Accuracy and F1 Score, Confusion Matrix**

* for name, model in models.items():

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

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

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

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

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

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

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

print(f'{name}:')

print(f'Precision: {precision:.2f}')

print(f'Recall: {recall:.2f}')

print(f'Accuracy: {accuracy:.2f}')

print(f'F1 Score: {f1:.2f}')

print(f'Confusion Matrix:')

print(cm)

print('---')

A white background with black border

Description automatically generated