المملكة العربية السعودية
وزارة التعليم
جامعة ام القرى
كلية الحاسبات

Project Title:

Predict Heart Disease Using Machine Learning Model **Logistic Regression** And **chatGPT**

Selected Topic I

Section:
1

Group:
4

Team Members:
Abdulhady Homoddah – 442006605
Azzam Alhothly – 442003546
Khalid Alharbi – 442001306
Mohammed Murad – 442005659

Project Supervisor:
Dr. Mohammed Alghamdi
maeghamdi@uqu.edu.sa

2024/1445

Ministry of Education
Umm Al-Qura University
College of Computers

Computer Science & Artificial Intelligence Department
Selected Topics I
Third Term 2024

Final Assignment

Project Title (Maximum 10 words):

Logistic Regression and chatGPT to predict heart disease.

Project Idea (Minimum 100 words):

In this project, our main goal is to build a mobile app with a machine learning model and openAI that can help us predict whether a person has heart disease or not based on their patient data. We want to develop a tool that can assist doctors in making accurate diagnoses and improving patient care. To achieve this, we'll be using a technique called Logistic Regression, which is a commonly used method in machine learning for classification tasks like ours. By analyzing various factors from the patient data, our model will learn patterns and make predictions about the presence or absence of heart disease. To bring our model to life, we'll be harnessing the power of Python and its helpful libraries. We'll start by organizing and cleaning the patient data to ensure it's in the best possible shape for training our model. Then, we'll train the model using this prepared data, fine-tuning it to make accurate predictions. It's important for us to evaluate the performance of our model to ensure its reliability. We'll use different metrics to measure its accuracy and effectiveness in identifying heart disease cases. By successfully building this predictive model, we hope to provide valuable support to healthcare professionals, enabling them to make informed decisions and potentially detect heart disease in its early stages. Ultimately, our aim is to improve patient outcomes and contribute to better healthcare practices.

Project Tools in details e.g. Machine Learning, Data Analysis, ChatGPT (Minimum 100 words):

In this machine learning project, we'll be using some popular libraries in Python to help us with different tasks. We'll start with the Pandas library, which will be our go-to tool for working with the dataset. It will assist us in loading the data, preparing it for analysis, and exploring its contents. To build our machine learning models, we'll rely on the Scikit-learn library. It's like our trusty toolbox, offering a wide range of classification algorithms.
We'll use these algorithms, such as Logistic Regression, to predict whether a patient has heart disease or not. Once we have our predictions, we'll want to visualize and present our findings. That's where the Matplotlib library comes in. It's like our artistic tool, allowing us to create different types of visualizations to better understand the data.
And to make our visualizations even more appealing, we'll also use Seaborn, a library that adds a touch of style and elegance to our statistical graphics. FastAPI is a modern, fast, and high-performance web framework for building APIs with Python. It is specifically designed for building APIs with high performance, scalability, and easy-to-use asynchronous capabilities. NGROK is a tool that provides secure tunnels to local host over the internet. NGROK allows you to expose your locally running web server to the internet, making it accessible by anyone with the provided URL. React Native is a powerful framework that enables developers to build mobile apps using familiar web technologies. It provides a balance between code reusability and native performance, making it a popular choice for cross-platform app development. We used ChatGPT by obtaining an API from the OpenAI website.

Project Features in details (Minimum 350 words):

Heart disease is a leading cause of mortality worldwide, and its early detection is critical for effective intervention and treatment. Medical professionals face the challenge of accurately diagnosing heart disease based on patient data, which often involves complex patterns and interactions among various factors. The objective of our project is to create an app that can assist doctors in making accurate diagnoses and improving patient care. By analyzing patient data, including factors such as age, gender, cholesterol levels, blood pressure, and other relevant medical indicators, our model will learn patterns associated with heart disease. By successfully building this app, we hope to provide valuable support to healthcare professionals, enabling them to make informed decisions and potentially detect heart disease in its early stages. Ultimately, our aim is to contribute to better healthcare practices, improve patient outcomes, and reduce the burden of heart disease on individuals and society as a whole.

Project Features:

1) The project involves building a mobile app that will serve as a platform for predicting whether a person has heart disease or not based on their patient data.

2) The core of the project is the implementation of a machine learning model using Logistic Regression. This model will analyze various factors from the patient data to learn patterns and make predictions about the presence or absence of heart disease.

3) The project utilizes Python programming language and its relevant libraries for implementing the machine learning model and developing the mobile app. Python offers a wide range of tools and resources for effective data analysis and app development.

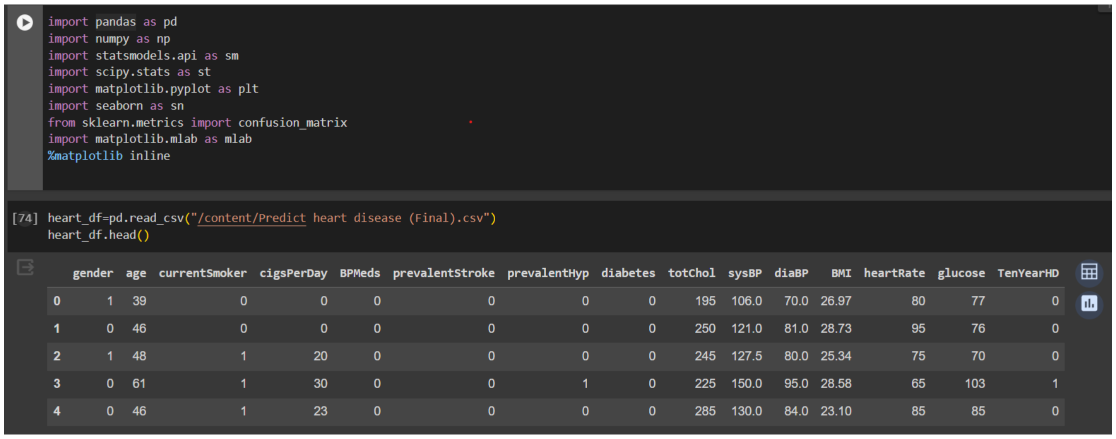
4) We will send patient data to ChatGPT, and it will respond to us whether the patient may have a heart disease or not.

5) The main objective of the project is to provide valuable support to healthcare professionals. The mobile app and predictive model aim to assist doctors in making accurate diagnoses, enabling them to make informed decisions regarding patient care.

6) By accurately predicting the presence or absence of heart disease, the project aims to contribute to the early detection of this condition. Early detection can lead to timely interventions and improved patient outcomes.

Project Screenshots with explanation for each of which:

- Model:



```
import pandas as pd
import numpy as np
import statsmodels.api as sm
import scipy.stats as st
import matplotlib.pyplot as plt
import seaborn as sn
from sklearn.metrics import confusion_matrix
import matplotlib.mlab as mlab
%matplotlib inline
```

```
[74] heart_df=pd.read_csv("/content/Predict heart disease (Final).csv")
     heart_df.head()
```

```
   gender  age  currentSmoker  cigsPerDay  BPMeds  prevalentStroke  prevalentHyp  diabetes  totChol  sysBP  diaBP    BMI  heartRate  glucose  TenYearHD
0       1   39              0           0       0                0             0         0      195  106.0   70.0  26.97         80       77          0
1       0   46              0           0       0                0             0         0      250  121.0   81.0  28.73         95       76          0
2       1   48              1          20       0                0             0         0      245  127.5   80.0  25.34         75       70          0
3       0   61              1          30       0                0             1         0      225  150.0   95.0  28.58         65      103          1
4       0   46              1          23       0                0             0         0      285  130.0   84.0  23.10         85       85          0
```

These lines import several libraries that are commonly used in data analysis and visualization tasks. Here's a brief explanation of each library:

1) **pandas (imported as pd):** is a powerful library for data manipulation and analysis. It provides data structures like DataFrames for efficient data handling.

2) **numpy (imported as np):** is a fundamental library for numerical computing in Python. It provides functions and tools for working with arrays and mathematical operations.

3) **statsmodels.api:** is a library that offers a wide range of statistical models and tests.

4) **scipy.stats:** provides various statistical functions and distributions for scientific computing.

5) **matplotlib.pyplot (imported as plt):** is a popular plotting library that allows you to create visualizations.

6) **Seaborn:** is a high-level interface for creating attractive statistical graphics.

7) **sklearn.metrics:** is a module within the scikit-learn library that includes various metrics for evaluating machine learning models.

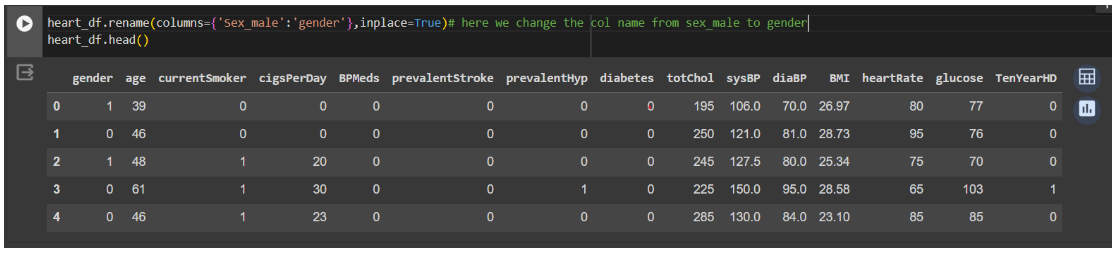
   matplotlib.mlab: provides a set of functions for data analysis and visualization.

- `heart_df=pd.read_csv("/content/Predict heart disease(Final).csv")`

This line reads the dataset from a file called "**Predict heart disease (Final).csv**" located in the "/**conten**t" directory. The dataset is loaded into a DataFrame object named "**heart_df**". **The pd.read_csv()** function is a pandas function that reads a CSV file and converts it into a DataFrame.

- `heart_df.head()`

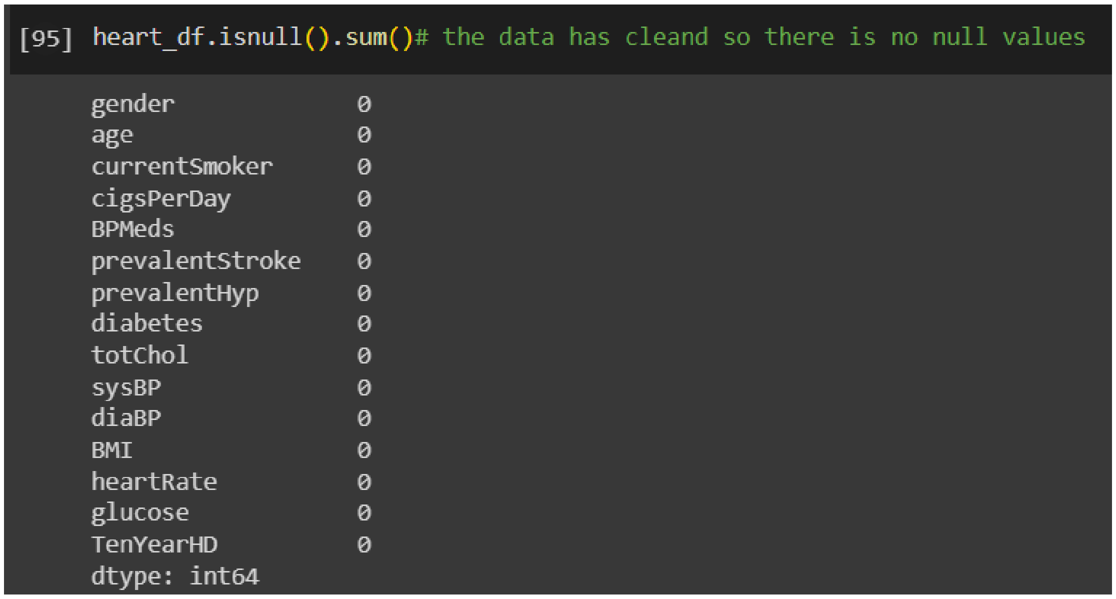
This line displays the first few rows of the DataFrame "**heart_df**" using the **head()** method. This gives you a preview of the dataset, allowing you to see the column names and a sample of the data.



```
heart_df.rename(columns={'Sex_male':'gender'},inplace=True)# here we change the col name from sex_male to gender
heart_df.head()
```

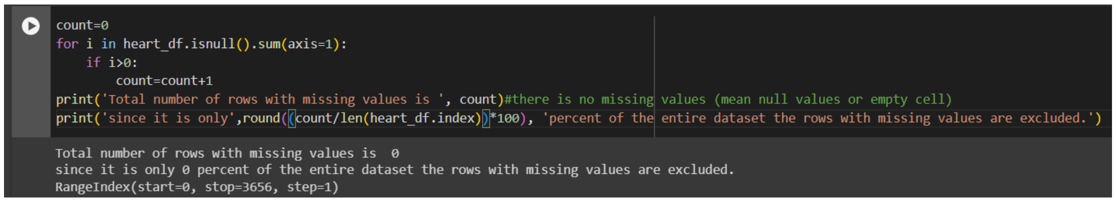
	gender	age	currentSmoker	cigsPerDay	BPMeds	prevalentStroke	prevalentHyp	diabetes	totChol	sysBP	diaBP	BMI	heartRate	glucose	TenYearHD
0	1	39	0	0	0	0	0	0	195	106.0	70.0	26.97	80	77	0
1	0	46	0	0	0	0	0	0	250	121.0	81.0	28.73	95	76	0
2	1	48	1	20	0	0	0	0	245	127.5	80.0	25.34	75	70	0
3	0	61	1	30	0	0	1	0	225	150.0	95.0	28.58	65	103	1
4	0	46	1	23	0	0	0	0	285	130.0	84.0	23.10	85	85	0

By executing this line of code, the column name **'Sex_male'** in heart_df is changed to **'gender'**. This can be useful for improving the clarity or consistency of column names, making them more descriptive or aligned with a specific naming convention.



```
[95] heart_df.isnull().sum()# the data has cleand so there is no null values

     gender             0
     age                0
     currentSmoker      0
     cigsPerDay         0
     BPMeds             0
     prevalentStroke    0
     prevalentHyp       0
     diabetes           0
     totChol            0
     sysBP              0
     diaBP              0
     BMI                0
     heartRate          0
     glucose            0
     TenYearHD          0
     dtype: int64
```

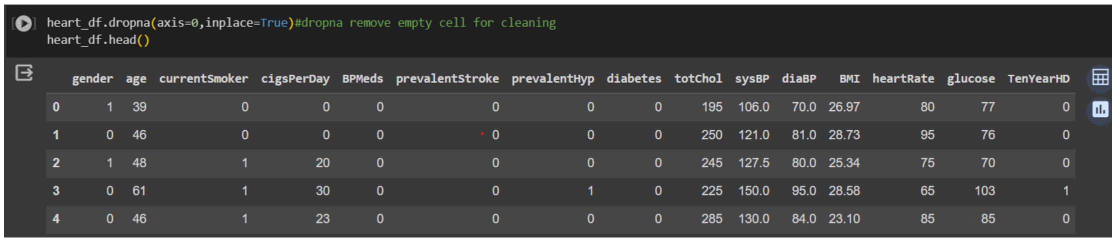
By executing this line of code, you will get a Series object that displays the count of missing values for each column in **'heart_df'**. The column names will be displayed as the index, and the corresponding values will indicate the number of missing values in each column.



```
count=0
for i in heart_df.isnull().sum(axis=1):
    if i>0:
        count=count+1
print('Total number of rows with missing values is ', count)#there is no missing values (mean null values or empty cell)
print('since it is only',round((count/len(heart_df.index))*100), 'percent of the entire dataset the rows with missing values are excluded.')
```

```
Total number of rows with missing values is  0
since it is only 0 percent of the entire dataset the rows with missing values are excluded.
RangeIndex(start=0, stop=3656, step=1)
```

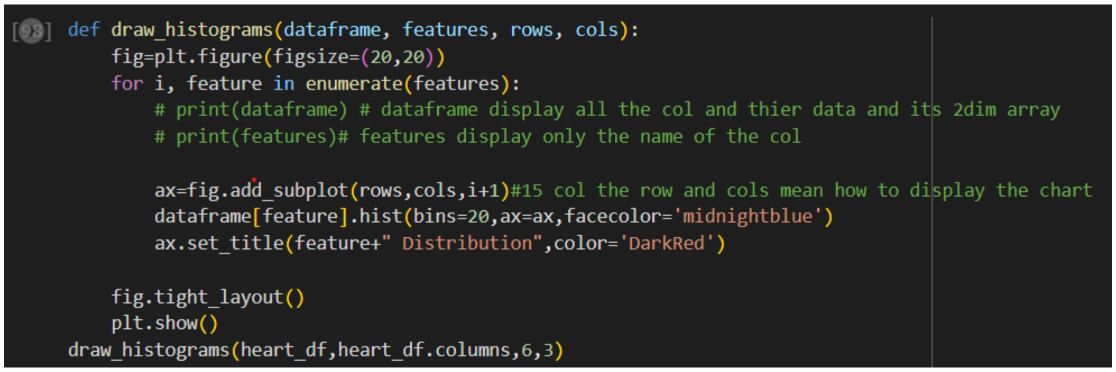
The code calculates the number of rows in the DataFrame **'heart_df'** that contain missing values and prints the count along with a percentage indicating the proportion of rows with missing values in the dataset.



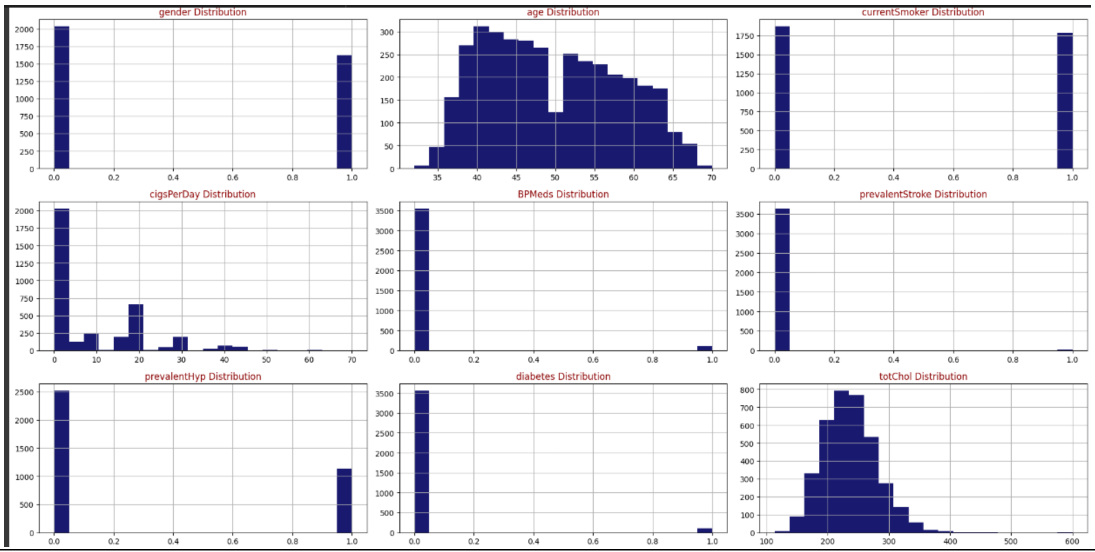
```
heart_df.dropna(axis=0,inplace=True)#dropna remove empty cell for cleaning
heart_df.head()
```

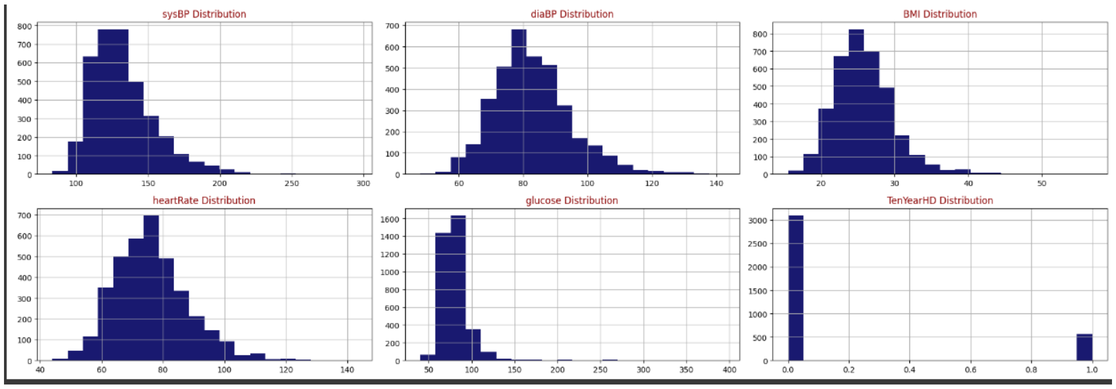
	gender	age	currentSmoker	cigsPerDay	BPMeds	prevalentStroke	prevalentHyp	diabetes	totChol	sysBP	diaBP	BMI	heartRate	glucose	TenYearHD
0	1	39	0	0	0	0	0	0	195	106.0	70.0	26.97	80	77	0
1	0	46	0	0	0	0	0	0	250	121.0	81.0	28.73	95	76	0
2	1	48	1	20	0	0	0	0	245	127.5	80.0	25.34	75	70	0
3	0	61	1	30	0	0	1	0	225	150.0	95.0	28.58	65	103	1
4	0	46	1	23	0	0	0	0	285	130.0	84.0	23.10	85	85	0

By executing this line of code, any row in **'heart_df'** that contains at least one missing value (NaN) will be dropped from the DataFrame. The changes will be made directly to **'heart_df'**.



```
def draw_histograms(dataframe, features, rows, cols):
    fig=plt.figure(figsize=(20,20))
    for i, feature in enumerate(features):
        # print(dataframe) # dataframe display all the col and thier data and its 2dim array
        # print(features)# features display only the name of the col

        ax=fig.add_subplot(rows,cols,i+1)#15 col the row and cols mean how to display the chart
        dataframe[feature].hist(bins=20,ax=ax,facecolor='midnightblue')
        ax.set_title(feature+" Distribution",color='DarkRed')

    fig.tight_layout()
    plt.show()
draw_histograms(heart_df,heart_df.columns,6,3)
```





```
def draw_histograms(dataframe, features, rows, cols):
        fig=plt.figure(figsize=(20,20))
```

This line defines the function `**draw_histograms**` that takes four parameters: `**dataframe**` (the DataFrame to plot), `**features**` (a list of column names to plot histograms for), `**rows**` (the number of rows in the grid), and `**cols**` (the number of columns in the grid). It also creates a new figure with a specified size using `**plt.figure()**`.

```
    for i, feature in enumerate(features):
        ax=fig.add_subplot(rows,cols,i+1)
        dataframe[feature].hist(bins=20,ax=ax,facecolor='midnightblue')
        ax.set_title(feature+" Distribution",color='DarkRed')
```

This loop iterates over each feature in the `**features**` list and creates a subplot for each feature in the grid. It uses `**fig.add_subplot()**` to add a new subplot to the figure at the current position specified by the `**rows**`, `**cols**`, and `**i+1**` values. The `**enumerate()**` function is used to obtain both the index (`**i**`) and value (`**feature**`) of each feature in the list.

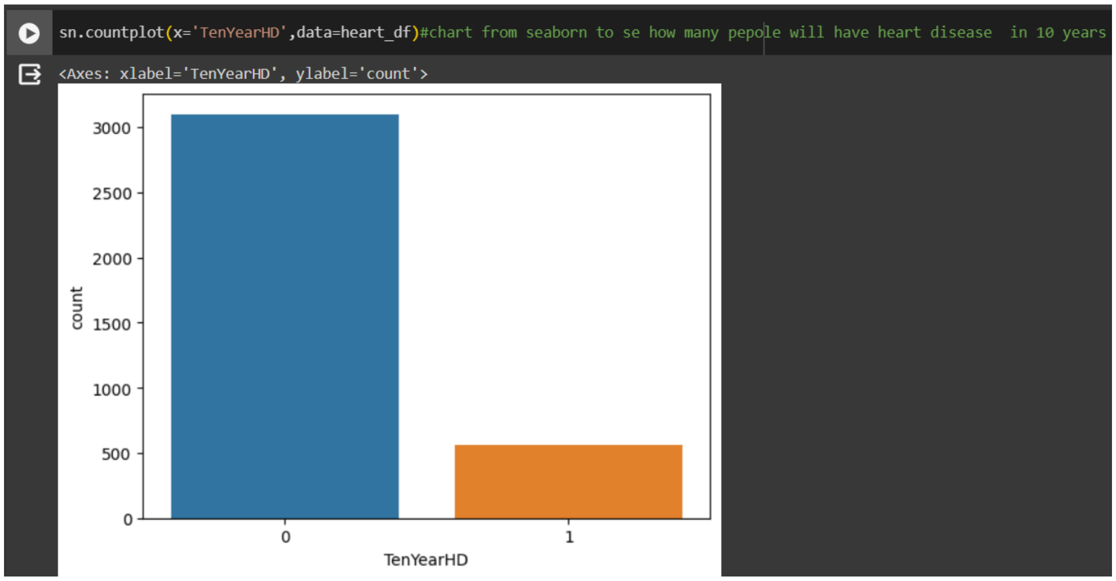
Inside each subplot, it plots a histogram for the corresponding feature using `**dataframe[feature].hist()**`. The `**bins=20**` parameter specifies the number of bins in the histogram, and `**ax=ax**` sets the current subplot as the axis for the histogram plot. The `**facecolor='midnightblue'**` parameter sets the color of the histogram bars.

Additionally, it sets a title for each subplot using `**ax.set_title()**`. The title includes the feature name followed by "**Distribution**", and the color is set to '**DarkRed**'.

```
    fig.tight_layout()
    plt.show()
```

After the loop, this code adjusts the layout of the subplots using `**fig.tight_layout()**` to ensure they are properly spaced. Finally, it displays the figure with all the histograms using `**plt.show()**`.

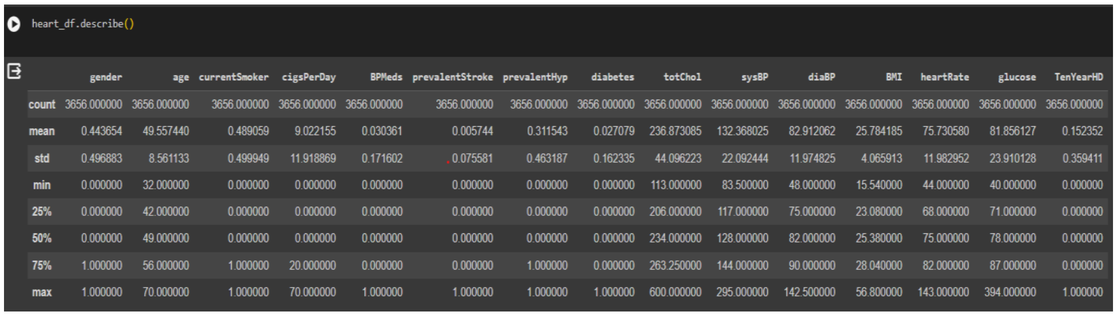
The function `**draw_histograms**` is then called with the DataFrame `**heart_df**`, all columns (`**heart_df.columns**`), and a grid size of 6 rows and 3 columns (`**6, 3**`) to generate the histogram grid.



```
sn.countplot(x='TenYearHD',data=heart_df)#chart from seaborn to se how many pepole will have heart disease  in 10 years
```

```
<Axes: xlabel='TenYearHD', ylabel='count'>
```

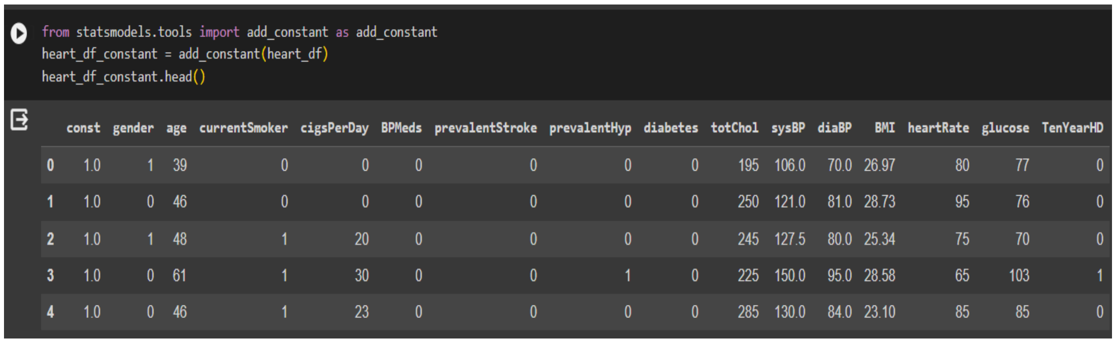
By executing this line of code, a **countplot** is generated based on the **'TenYearCHD'** column in **heart_df**. The **countplot** will have bars representing the count of observations in each unique category of the **'TenYearCHD'** variable.



```
heart_df.describe()
```

```
            gender          age  currentSmoker   cigsPerDay       BPMeds  prevalentStroke  prevalentHyp     diabetes      totChol        sysBP        diaBP          BMI    heartRate      glucose    TenYearHD
count  3656.000000  3656.000000    3656.000000  3656.000000  3656.000000      3656.000000   3656.000000  3656.000000  3656.000000  3656.000000  3656.000000  3656.000000  3656.000000  3656.000000  3656.000000
mean      0.443654    49.557440       0.489059     9.022155     0.030361         0.005744      0.311543     0.027079   236.873085   132.368025    82.912062    25.784185    75.730580    81.856127     0.152352
std       0.496883     8.561133       0.499949    11.918869     0.171602         0.075581      0.463187     0.162335    44.096223    22.092444    11.974825     4.065913    11.982952    23.910128     0.359411
min       0.000000    32.000000       0.000000     0.000000     0.000000         0.000000      0.000000     0.000000   113.000000    83.500000    48.000000    15.540000    44.000000    40.000000     0.000000
25%       0.000000    42.000000       0.000000     0.000000     0.000000         0.000000      0.000000     0.000000   206.000000   117.000000    75.000000    23.080000    68.000000    71.000000     0.000000
50%       0.000000    49.000000       0.000000     0.000000     0.000000         0.000000      0.000000     0.000000   234.000000   128.000000    82.000000    25.380000    75.000000    78.000000     0.000000
75%       1.000000    56.000000       1.000000    20.000000     0.000000         0.000000      1.000000     0.000000   263.250000   144.000000    90.000000    28.040000    82.000000    87.000000     0.000000
max       1.000000    70.000000       1.000000    70.000000     1.000000         1.000000      1.000000     1.000000   600.000000   295.000000   142.500000    56.800000   143.000000   394.000000     1.000000
```

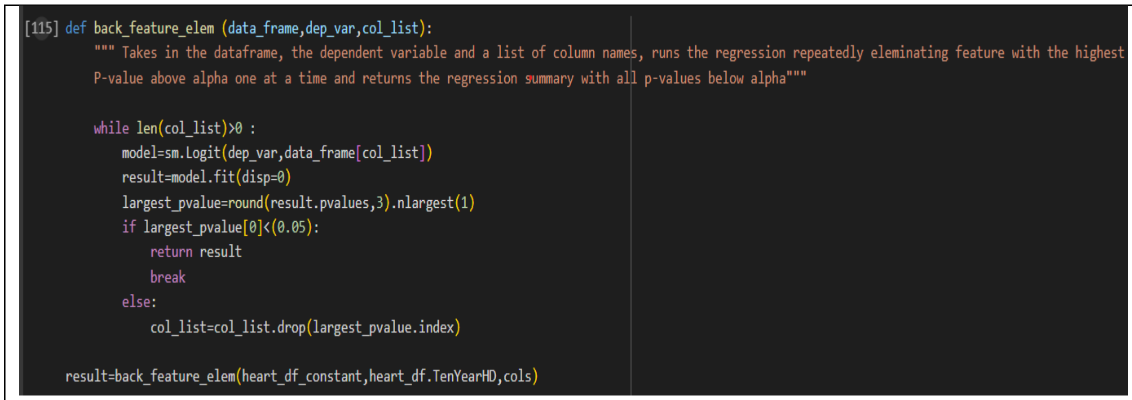
By executing this line of code, a summary of descriptive statistics will be generated for each numerical column in **heart_df**.



```
from statsmodels.tools import add_constant as add_constant
heart_df_constant = add_constant(heart_df)
heart_df_constant.head()
```

```
   const  gender  age  currentSmoker  cigsPerDay  BPMeds  prevalentStroke  prevalentHyp  diabetes  totChol  sysBP  diaBP    BMI  heartRate  glucose  TenYearHD
0    1.0       1   39              0           0       0                0             0         0      195  106.0   70.0  26.97         80       77          0
1    1.0       0   46              0           0       0                0             0         0      250  121.0   81.0  28.73         95       76          0
2    1.0       1   48              1          20       0                0             0         0      245  127.5   80.0  25.34         75       70          0
3    1.0       0   61              1          30       0                0             1         0      225  150.0   95.0  28.58         65      103          1
4    1.0       0   46              1          23       0                0             0         0      285  130.0   84.0  23.10         85       85          0
```

By adding a constant column, you are preparing the DataFrame for use in various statistical models, such as linear regression. The constant column represents the intercept term in the model equation. It allows the model to estimate the baseline value or starting point when all other independent variables are zero.



```
[115] def back_feature_elem (data_frame,dep_var,col_list):
        """ Takes in the dataframe, the dependent variable and a list of column names, runs the regression repeatedly eleminating feature with the highest
        P-value above alpha one at a time and returns the regression summary with all p-values below alpha"""

        while len(col_list)>0 :
            model=sm.Logit(dep_var,data_frame[col_list])
            result=model.fit(disp=0)
            largest_pvalue=round(result.pvalues,3).nlargest(1)
            if largest_pvalue[0]<(0.05):
                return result
                break
            else:
                col_list=col_list.drop(largest_pvalue.index)

    result=back_feature_elem(heart_df_constant,heart_df.TenYearHD,cols)
```

The **'back_feature_elem'** function takes three arguments: **data_frame**, which is the input **dataframe**, **dep_var**, which represents the dependent variable, and **col_list**, which is a list of column names.

Inside the function, a while loop is used to iterate until there are columns remaining in **col_list**. The loop performs the following steps:

1) It creates a logistic regression model using **sm.Logit** from the **statsmodels** library. The dependent variable (**dep_var**) and the subset of columns (**data_frame[col_list]**) are used as inputs.

2) The model is fitted using **model.fit(disp=0)**, which estimates the logistic regression coefficients and other model parameters.

3) The **p-values** of the coefficients are obtained using **result.pvalues**. The round function is used to round the **p-values** to three decimal places.

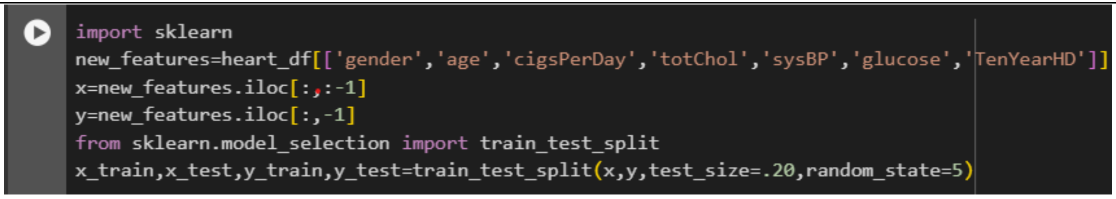
4) The largest **p-value** is extracted using **nlargest(1)** to identify the feature with the highest **p-value**.

5) If the largest **p-value** is less than 0.05 (the significance level commonly used), the function returns the regression result (result) and breaks out of the loop.

6) If the largest **p-value** is greater than or equal to 0.05, the corresponding feature is dropped from **col_list** using **col_list.drop(largest_pvalue.index)**.

Finally, the function **back_feature_elem** is called with specific arguments (**heart_df_constant, heart_df.TenYearCHD, cols**), and the result is assigned to the variable result.

The purpose of this code is to perform backward feature elimination for **logistic regression**. Starting with a set of features, the algorithm removes one feature at a time based on the highest **p-value** until all remaining features have **p-values** below the significance level. This approach helps identify the most significant features for predicting the dependent variable.



```
import sklearn
new_features=heart_df[['gender','age','cigsPerDay','totChol','sysBP','glucose','TenYearHD']]
x=new_features.iloc[:,:-1]
y=new_features.iloc[:,-1]
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=.20,random_state=5)
```

```
new_features=heart_df[['age','Sex_male','cigsPerDay','totChol','sysBP','glucose','TenYearCHD']]
```

This line creates a new **DataFrame** called **new_features** by selecting specific columns from **heart_df**.

```
x=new_features.iloc[:,:-1]
y=new_features.iloc[:,-1]
```

These two lines split the **'new_features'** DataFrame into input features (x) and target variable (y). The iloc indexing is used to select all rows (:) and all columns except the last column (:-1) for x, and only the last column (-1) for y. In this case, '**TenYearCHD**' is considered the target variable, and the other columns are the input features.

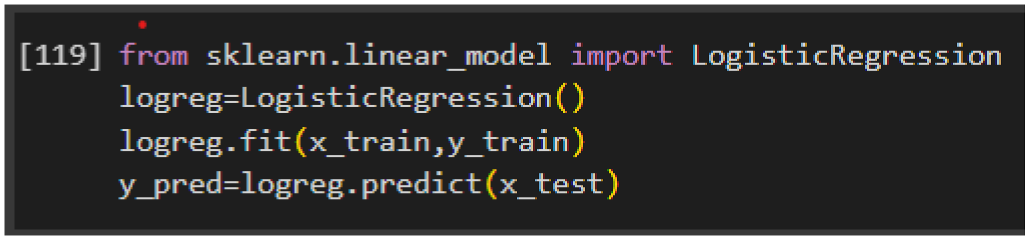
```
from sklearn.cross_validation import train_test_split
```

This line imports the **train_test_split** function from the **sklearn.model_selection** module. This function is used to split the dataset into training and testing sets.

```
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=.20,random_state=5)
```

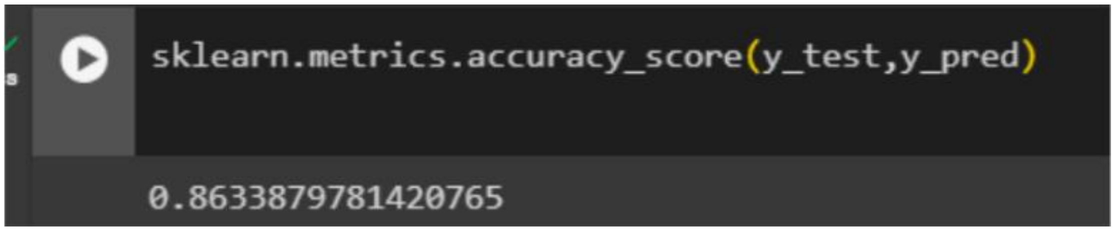
This line applies the **train_test_split** function to the input features (x) and target variable (y). It splits the data into training and testing sets, with 80% of the data used for training (**x_train, y_train**) and 20% for testing (**x_test, y_test**). The **test_size** parameter is set to 0.20, indicating the desired proportion for the testing set. The **random_state** parameter is set to 5, ensuring reproducibility by fixing the random seed for the splitting process.

By splitting the data into training and testing sets, you can evaluate the performance of machine learning models on unseen data. The training set is used to train the model, while the testing set is used to assess its performance and generalization ability.



```
[119] from sklearn.linear_model import LogisticRegression
      logreg=LogisticRegression()
      logreg.fit(x_train,y_train)
      y_pred=logreg.predict(x_test)
```

This code snippet performs **logistic regression** modeling using **scikit-learn**. It initializes a logistic regression model, fits it to the training data, and then uses the trained model to predict the class labels for the testing data. **Logistic regression** is commonly used for binary classification tasks, where the goal is to predict one of two possible classes based on the input features.

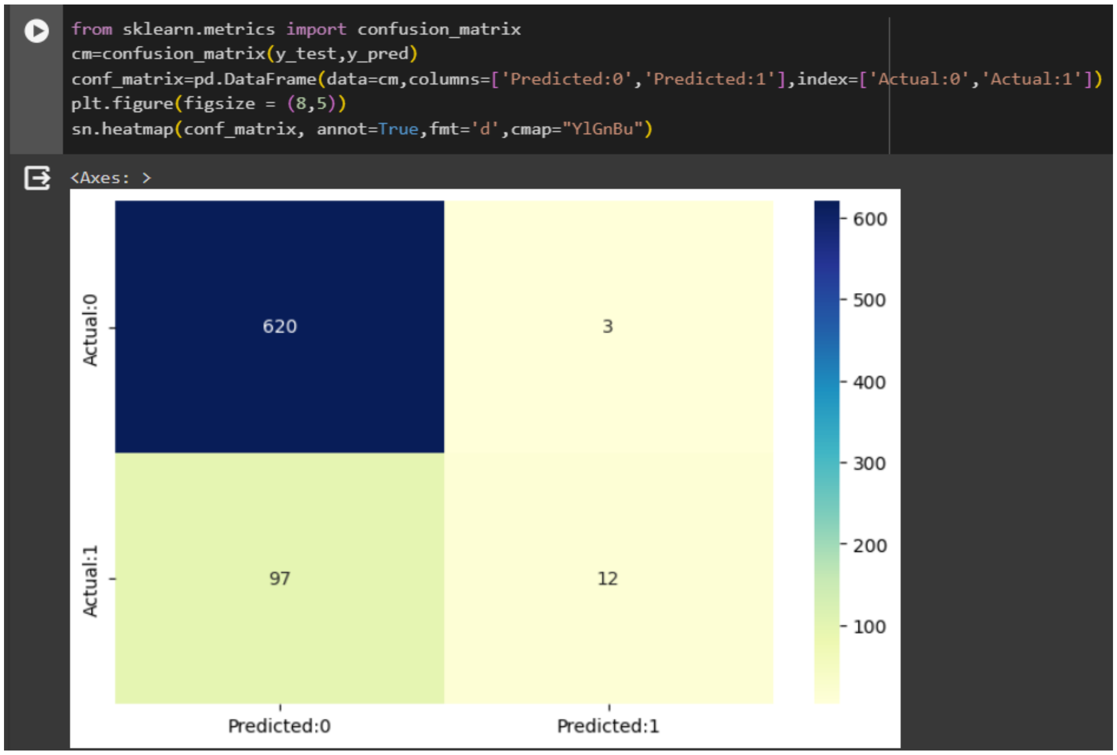


```
sklearn.metrics.accuracy_score(y_test,y_pred)
```

```
0.8633879781420765
```

The **accuracy_score** function compares the predicted labels (**y_pred**) with the true labels (**y_test**) and returns the accuracy score as a floating-point number between 0 and 1. The accuracy score is calculated as the ratio of the number of correctly predicted instances to the total number of instances.

A higher accuracy score indicates that the model has made more correct predictions on the testing dataset, while a lower score suggests that the model's predictions do not align well with the true labels. **Accuracy of the model is 0.86.**



```
from sklearn.metrics import confusion_matrix
cm=confusion_matrix(y_test,y_pred)
conf_matrix=pd.DataFrame(data=cm,columns=['Predicted:0','Predicted:1'],index=['Actual:0','Actual:1'])
plt.figure(figsize = (8,5))
sn.heatmap(conf_matrix, annot=True,fmt='d',cmap="YlGnBu")
```

```
<Axes: >
```

	Predicted:0	Predicted:1
Actual:0	620	3
Actual:1	97	12

```
from sklearn.metrics import confusion_matrix
```

This line imports the **confusion_matrix** function from the **sklearn.metrics** module. The **confusion_matrix** function is used to create a confusion matrix, which is a table that summarizes the performance of a classification model by comparing the predicted class labels to the true class labels.

```
cm=confusion_matrix(y_test,y_pred)
```

This line calculates the confusion matrix by calling the **confusion_matrix** function and passing the true labels (**y_test**) and predicted labels (**y_pred**) as arguments. The resulting confusion matrix, represented by cm, is a 2x2 array that shows the counts of true positive, true negative, false positive, and false negative predictions.

```
conf_matrix=pd.DataFrame(data=cm,columns=['Predicted:0','Predicted:1'],index=['Actual:0','Actual:1'])
```

This line creates a pandas DataFrame called **conf_matrix** using the pd.DataFrame function. The data parameter is set to the confusion matrix cm, and the columns and index parameters are set to the labels for the predicted and actual classes, respectively. This DataFrame organizes the confusion matrix data for better readability.

```
plt.figure(figsize = (8,5))
sn.heatmap(conf_matrix, annot=True,fmt='d',cmap="YlGnBu")
```

This code visualizes the confusion matrix using a heatmap plot. It sets the figure size to (8, 5) using **plt.figure(figsize=(8, 5))**. Then, it uses **sn.heatmap** from the seaborn library (sn) to create the heatmap plot. The **conf_matrix** DataFrame is passed as the data for the heatmap. The **annot=True** parameter displays the values inside the heatmap cells, and **fmt='d'** specifies that the cell values should be displayed as integers. The **cmap="YlGnBu"** parameter sets the color map for the heatmap.

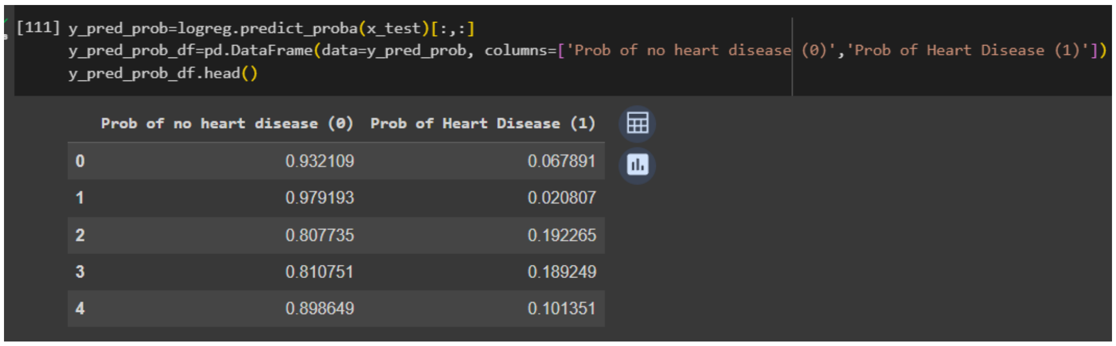
The confusion matrix shows 620+12 = 632 correct predictions and 97+3= 100 incorrect ones.

True Positives (TP): 12 - This represents the number of instances that are actually positive (actual: 1) and are correctly predicted as positive (predicted: 1).

True Negatives (TN): 620 - This represents the number of instances that are actually negative (actual: 0) and are correctly predicted as negative (predicted: 0).

False Positives (FP): 3 - This represents the number of instances that are actually negative (actual: 0) but are incorrectly predicted as positive (predicted: 1).

False Negatives (FN): 97 - This represents the number of instances that are actually positive (actual: 1) but are incorrectly predicted as negative (predicted: 0).



```
[111] y_pred_prob=logreg.predict_proba(x_test)[:,:]
      y_pred_prob_df=pd.DataFrame(data=y_pred_prob, columns=['Prob of no heart disease (0)','Prob of Heart Disease (1)'])
      y_pred_prob_df.head()
```

	Prob of no heart disease (0)	Prob of Heart Disease (1)
0	0.932109	0.067891
1	0.979193	0.020807
2	0.807735	0.192265
3	0.810751	0.189249
4	0.898649	0.101351

```
y_pred_prob=logreg.predict_proba(x_test)[:,:]
```

In this code snippet, **logreg** refers to a logistic regression model. The **predict_proba()** function is called on the logistic regression model with **x_test** as the input. This function calculates the predicted probabilities of the input samples belonging to each class. The resulting probabilities are stored in the **y_pred_prob** variable.

The [:, :] indexing is used to select all rows and all columns from the array of predicted probabilities. This ensures that the dimensions of **y_pred_prob** match the shape of the test dataset.

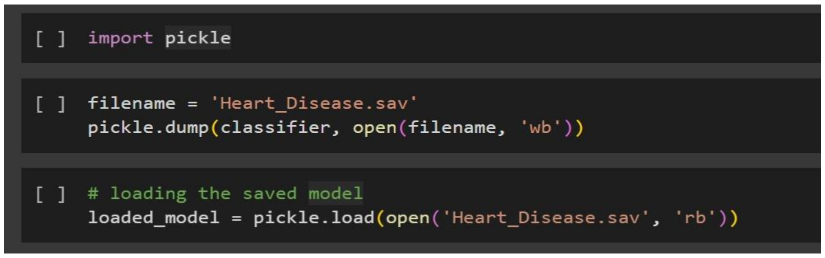
```
y_pred_prob_df=pd.DataFrame(data=y_pred_prob, columns=['Prob of no heart disease (0)','Prob of Heart Disease (1)'])
```

In this code snippet, pd refers to the **pandas** library, assuming it has been imported earlier. The **DataFrame()** function from pandas is used to create a DataFrame object named **y_pred_prob_df**.

The data parameter is set to **y_pred_prob**, which contains the predicted probabilities of the two classes obtained from the previous code snippet. The test data with a default classification threshold of 0.5

The columns parameter is set **to ['Prob of no heart disease (0)', 'Prob of Heart Disease (1)']**, specifying the column names for the DataFrame. This will label the two columns that represent the predicted probabilities of no heart disease and heart disease, respectively.

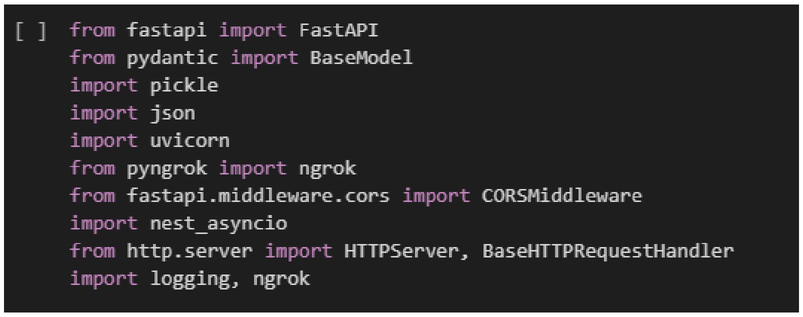
- API:



```
[ ] import pickle

[ ] filename = 'Heart_Disease.sav'
    pickle.dump(classifier, open(filename, 'wb'))

[ ] # loading the saved model
    loaded_model = pickle.load(open('Heart_Disease.sav', 'rb'))
```

"import pickle" is used to import the pickle module, which provides functionality for serializing and deserializing Python objects.

This variable **"filename"** will be used to specify the name of the file where the classifier will be saved.
"pickle.dump()" function to save the "classifier" object to the file specified by the "filename" variable. The "pickle.dump()" function takes two arguments: the object to be saved (the "classifier" variable) and a file object opened in binary write mode ('wb').
"open()" function used to open the file in binary write mode.
"pickle.load()" function takes a single argument, which is a file object opened in binary read mode ('rb'). Loads a saved model from the file 'Heart_Disease.sav' using the pickle module. The loaded model is assigned to the variable "loaded_model".



```
[ ]  from fastapi import FastAPI
     from pydantic import BaseModel
     import pickle
     import json
     import uvicorn
     from pyngrok import ngrok
     from fastapi.middleware.cors import CORSMiddleware
     import nest_asyncio
     from http.server import HTTPServer, BaseHTTPRequestHandler
     import logging, ngrok
```

1) **from fastapi import FastAPI:** FastAPI is a Python web framework used to build APIs quickly and efficiently.

2) **from pydantic import BaseModel:** BaseModel is used to create data models and perform data validation.

3) **import pickle:** Provides functionality for serializing and deserializing Python objects.

4) **import json:** Provides functions for working with JSON data.

5) **import uvicorn:** Is a lightning-fast ASGI server implementation used to run the FastAPI application.

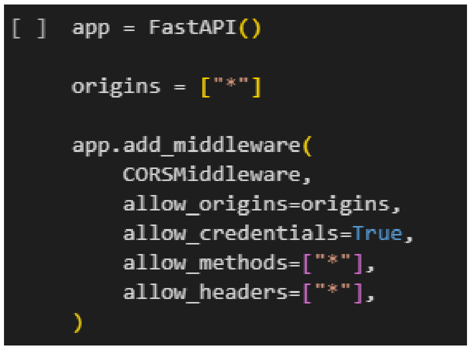
6) **from pyngrok import ngrok:** Provides a way to create secure tunnels to localhost for exposing local servers to the internet.

7) **from fastapi.middleware.cors import CORSMiddleware:** CORSMiddleware is used to handle Cross-Origin Resource Sharing (CORS) in FastAPI applications.

8) **import nest_asyncio:** Allows running asyncio event loops within Jupyter notebooks or other environments that already have an event loop running.

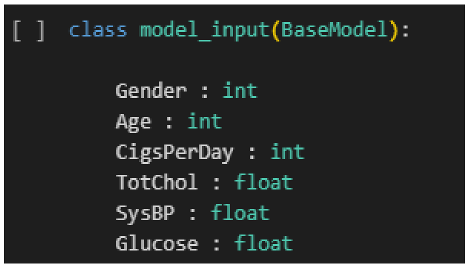
9) **from http.server import HTTPServer, BaseHTTPRequestHandler:** These classes are used for creating an HTTP server and handling HTTP requests.

10) **import logging, ngrok:** It seems that the logging module is used for logging purposes, and ngrok may be used for tunneling the server with additional configurations.



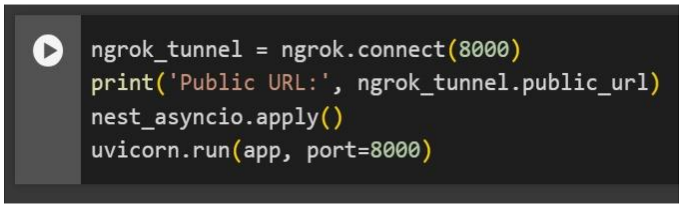
```
[ ]  app = FastAPI()

     origins = ["*"]

     app.add_middleware(
         CORSMiddleware,
         allow_origins=origins,
         allow_credentials=True,
         allow_methods=["*"],
         allow_headers=["*"],
     )
```

This code sets up the FastAPI application, creates a wildcard CORS configuration that allows requests from any origin, and adds the **CORSMiddleware** with the specified CORS settings to the application. This enables the API to handle requests from different domains and allows the use of credentials and various HTTP methods and headers in those requests.
The "*" wildcard allows any origins, methods, and headers.



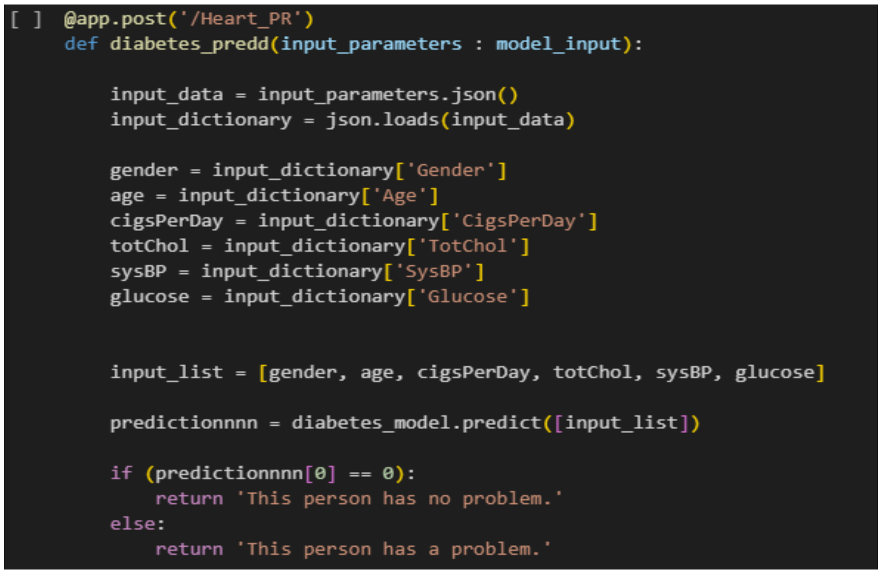
```
[ ]  class model_input(BaseModel):

         Gender : int
         Age : int
         CigsPerDay : int
         TotChol : float
         SysBP : float
         Glucose : float
```

The purpose of this class to be define the structure or schema of the input data for a model. The input data is expected to be in JSON format, and the class provides an example of the expected input fields.



```
ngrok_tunnel = ngrok.connect(8000)
print('Public URL:', ngrok_tunnel.public_url)
nest_asyncio.apply()
uvicorn.run(app, port=8000)
```

When executed, this code will establish a tunnel using ngrok to expose the local web server running on port 8000. The public URL generated by ngrok will be printed, allowing access to the local server from the internet. Finally, the uvicorn server will start running the specified application on port 8000.



```
[ ]  @app.post('/Heart_PR')
     def diabetes_predd(input_parameters : model_input):

         input_data = input_parameters.json()
         input_dictionary = json.loads(input_data)

         gender = input_dictionary['Gender']
         age = input_dictionary['Age']
         cigsPerDay = input_dictionary['CigsPerDay']
         totChol = input_dictionary['TotChol']
         sysBP = input_dictionary['SysBP']
         glucose = input_dictionary['Glucose']


         input_list = [gender, age, cigsPerDay, totChol, sysBP, glucose]

         predictionnnn = diabetes_model.predict([input_list])

         if (predictionnnn[0] == 0):
             return 'This person has no problem.'
         else:
             return 'This person has a problem.'
```

1) **@app.post('/Heart_Disease_prediction'):** This is a decorator that defines the route and HTTP method for the endpoint. It specifies that the endpoint should be accessible via a POST request to the '/ Heart_Disease_prediction' URL.

2) **def Heart_Disease_predd(input_parameters: model_input):** This is the function that will be executed when the POST request is made to the file endpoint. which is likely the class defined earlier to represent the input data structure.

3) **input_data = input_parameters.json():** This line retrieves the JSON data from the input_parameters object and assigns it to the input_data variable. It converts the input_parameters object to a JSON string.

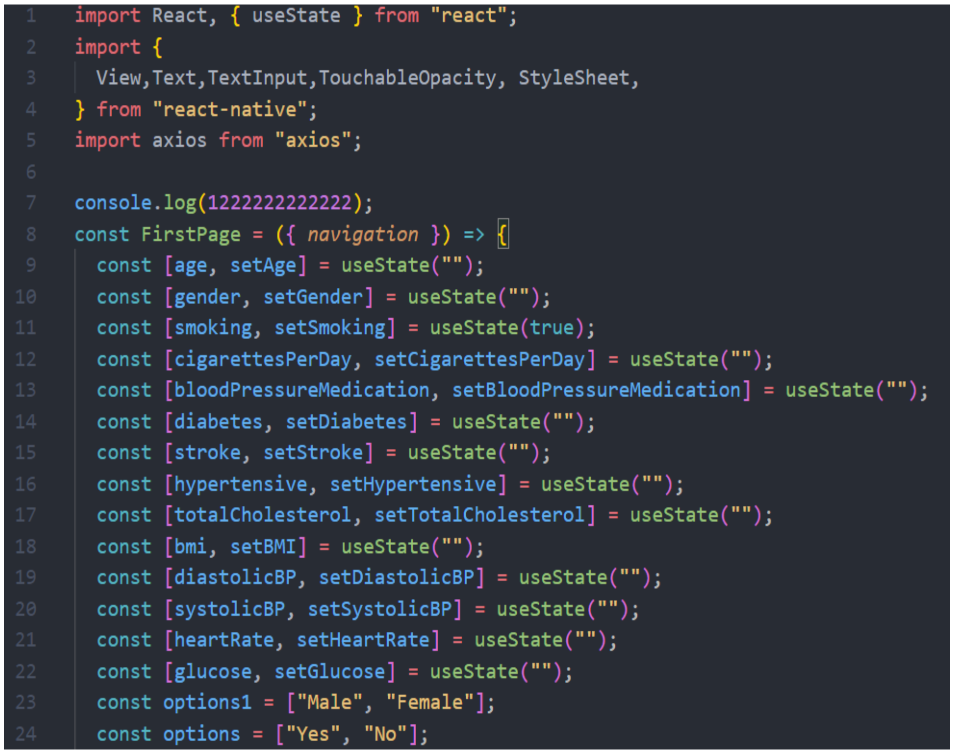
4) **input_dictionary = json.loads(input_data):** This line parses the JSON string stored in input_data and converts it into a Python dictionary using the json.loads() function. This allows you to access the individual values of the input data.

5) **Gender = input_dictionary['Gender'], Age = input_dictionary['Age'] …**: These lines extract the respective values from the input_dictionary. It assumes that the dictionary keys match the attribute names defined in the model_input class.

6) **input_list = […]:** This line creates a list called input_list containing the extracted values from the previous step. This list represents the input data that will be passed to the diabetes prediction model.

7) **prediction = Heart_Disease _model.predict([input_list]):** This line calls the predict() method on the object to make a prediction based on the input data. It passes the input_list as an argument to the predict() method. The result of the prediction is stored in the prediction variable.

- Screens react native:



```
import React, { useState } from "react";
import {
  View,Text,TextInput,TouchableOpacity, StyleSheet,
} from "react-native";
import axios from "axios";

console.log(1222222222222);
const FirstPage = ({ navigation }) => {
  const [age, setAge] = useState("");
  const [gender, setGender] = useState("");
  const [smoking, setSmoking] = useState(true);
  const [cigarettesPerDay, setCigarettesPerDay] = useState("");
  const [bloodPressureMedication, setBloodPressureMedication] = useState("");
  const [diabetes, setDiabetes] = useState("");
  const [stroke, setStroke] = useState("");
  const [hypertensive, setHypertensive] = useState("");
  const [totalCholesterol, setTotalCholesterol] = useState("");
  const [bmi, setBMI] = useState("");
  const [diastolicBP, setDiastolicBP] = useState("");
  const [systolicBP, setSystolicBP] = useState("");
  const [heartRate, setHeartRate] = useState("");
  const [glucose, setGlucose] = useState("");
  const options1 = ["Male", "Female"];
  const options = ["Yes", "No"];
```

1) **import React, {useState} from 'react'; :** This line imports the React library and the "useState" hook from the React package. The "useState" hook allows functional components to have state and enables them to update their state dynamically.

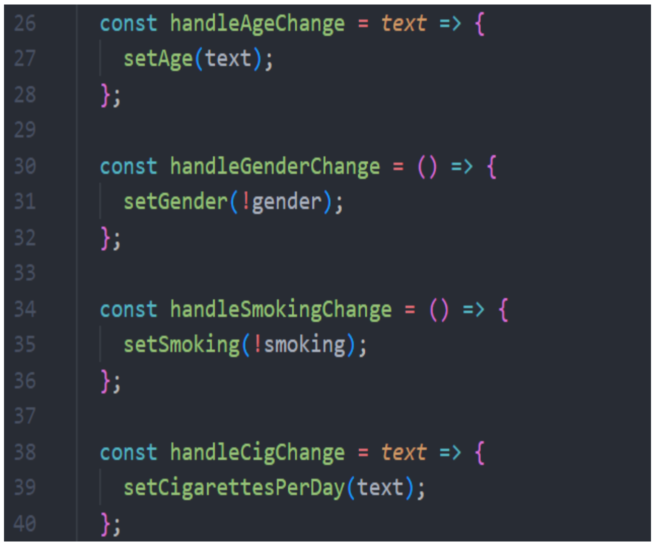
2) **import {View, Text, TextInput, … } from 'react-native'; :** This line imports specific components and utilities from the "react-native" library. These components are used to create user interfaces in React Native applications.

3) **const FirstPage = ({ navigation }) => { ... }:** The component receives the navigation prop as an argument, which is likely passed from a parent component and used for navigation purposes.

4) **const options1 = ["Male", "Female"]; :** This line initializes an array named options1 with the values "Male" and "Female". This array likely represents a set of options related to gender.

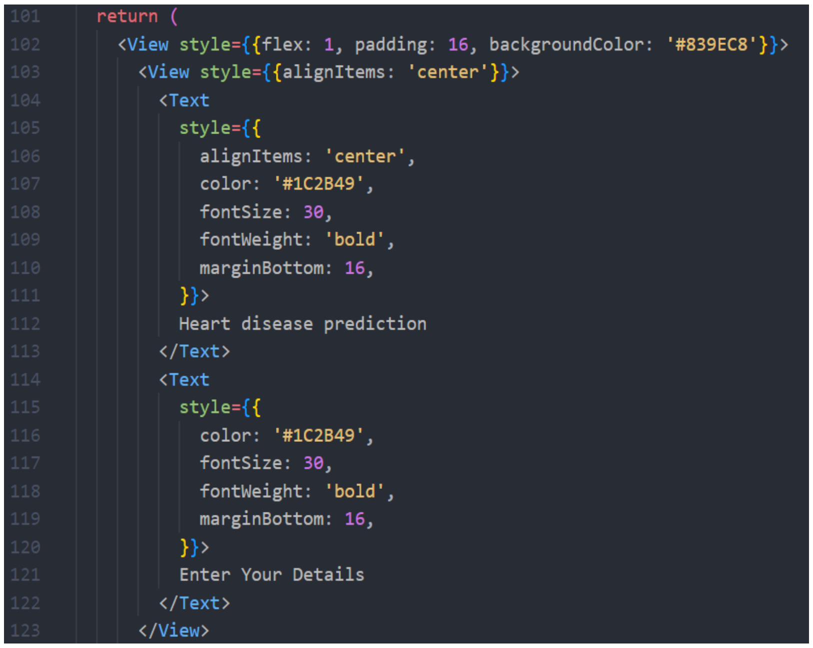
5) **const options = ["Yes", "No"]; :** This line initializes an array named options with the values "Yes" and "No". This array likely represents a set of yes/no options.

6) **const [age, setAge] = useState(''); :** This line declares a state variable named "age" using the "useState" hook. The "age" variable is initially set to an empty string (''). The "setAge" function is used to update the value of the "age" variable.



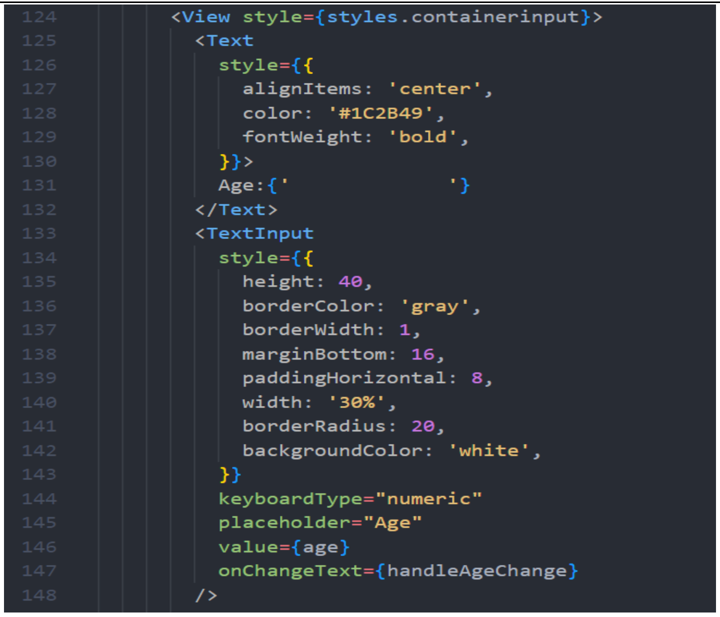
```
const handleAgeChange = text => {
  setAge(text);
};

const handleGenderChange = () => {
  setGender(!gender);
};

const handleSmokingChange = () => {
  setSmoking(!smoking);
};

const handleCigChange = text => {
  setCigarettesPerDay(text);
};
```

These functions are typically used as event handlers in response to user interactions. For example, **handleAgeChange** could be used as an event handler for a text input field where the user enters their age.



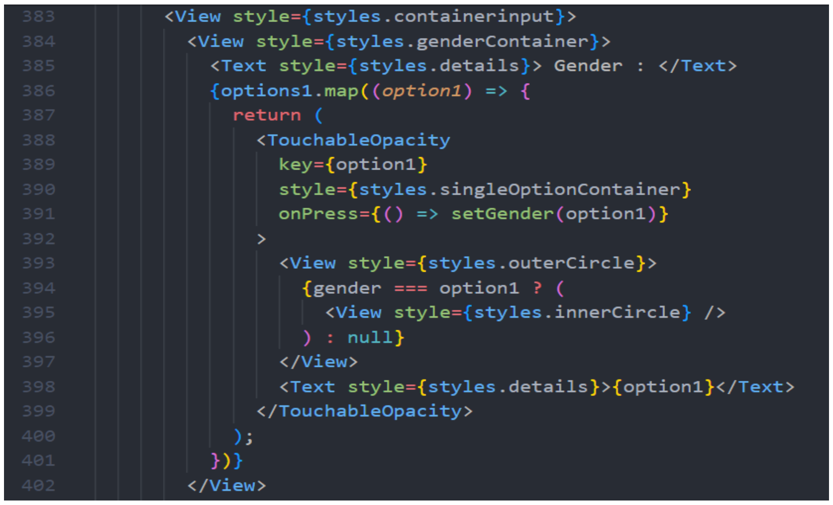
```
return (
  <View style={{flex: 1, padding: 16, backgroundColor: '#839EC8'}}>
    <View style={{alignItems: 'center'}}>
      <Text
        style={{
          alignItems: 'center',
          color: '#1C2B49',
          fontSize: 30,
          fontWeight: 'bold',
          marginBottom: 16,
        }}>
        Heart disease prediction
      </Text>
      <Text
        style={{
          color: '#1C2B49',
          fontSize: 30,
          fontWeight: 'bold',
          marginBottom: 16,
        }}>
        Enter Your Details
      </Text>
    </View>
```

This code renders a view and two text elements, one displaying "Heart disease prediction" and the other displaying "Enter Your Details". The text elements are centered within their parent views.



```
<View style={styles.containerinput}>
  <Text
    style={{
      alignItems: 'center',
      color: '#1C2B49',
      fontWeight: 'bold',
    }}>
    Age:{'              '}
  </Text>
  <TextInput
    style={{
      height: 40,
      borderColor: 'gray',
      borderWidth: 1,
      marginBottom: 16,
      paddingHorizontal: 8,
      width: '30%',
      borderRadius: 20,
      backgroundColor: 'white',
    }}
    keyboardType="numeric"
    placeholder="Age"
    value={age}
    onChangeText={handleAgeChange}
  />
```

This code renders a view with a label "Age:" followed by an input field where the user can enter their age. The input field has some styling applied to it. **keyboardType="numeric"** specifies that the keyboard type for this input field should be numeric, allowing the user to enter only numeric values. The value entered by the user is controlled by the age state variable, and any changes in the input field trigger the **handleAgeChange** function.

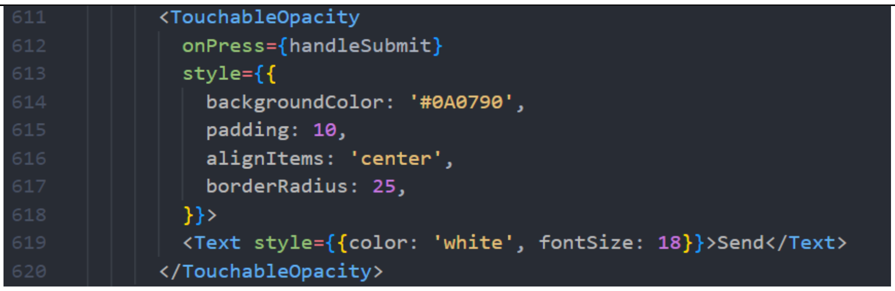


```
<View style={styles.containerinput}>
  <View style={styles.genderContainer}>
    <Text style={styles.details}> Gender : </Text>
    {options1.map((option1) => {
      return (
        <TouchableOpacity
          key={option1}
          style={styles.singleOptionContainer}
          onPress={() => setGender(option1)}
        >
          <View style={styles.outerCircle}>
            {gender === option1 ? (
              <View style={styles.innerCircle} />
            ) : null}
          </View>
          <Text style={styles.details}>{option1}</Text>
        </TouchableOpacity>
      );
    })}
  </View>
```

{options1.map((option1) => { ... })}: This code uses the map() function to iterate over an array of options1. It generates a set of UI elements based on each option in the array.

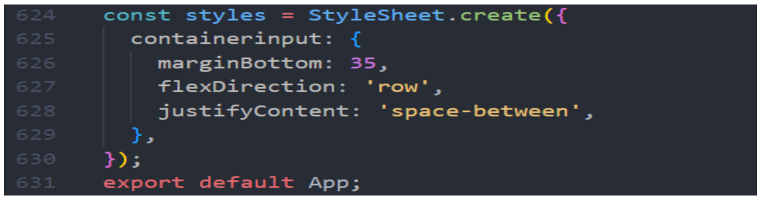
<TouchableOpacity ...>: This component represents a touchable area that can be pressed by the user. It wraps the UI elements related to each option.

{gender === option1 ? ... }: This conditional statement checks if the gender state matches the current option1. If they match, it renders the inner circle component; otherwise, it renders nothing (null).



```
<TouchableOpacity
  onPress={handleSubmit}
  style={{
    backgroundColor: '#0A0790',
    padding: 10,
    alignItems: 'center',
    borderRadius: 25,
  }}>
  <Text style={{color: 'white', fontSize: 18}}>Send</Text>
</TouchableOpacity>
```

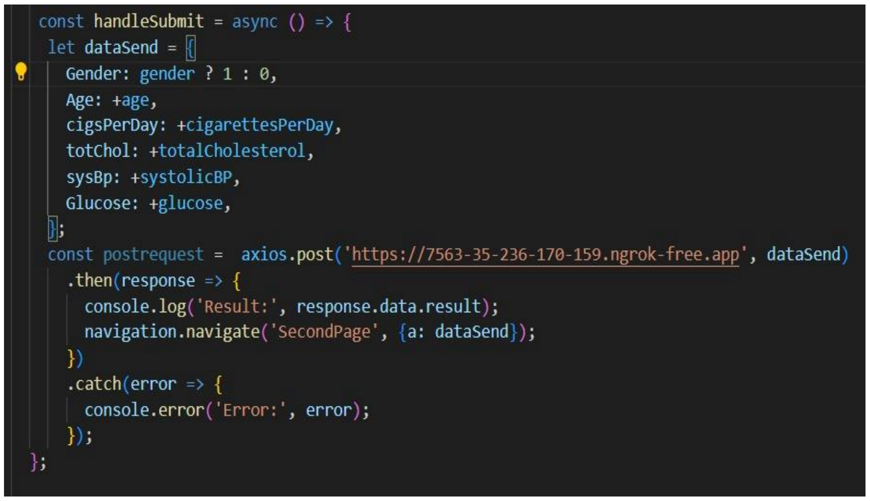
<TouchableOpacity>: This is a component that provides touchable behavior to its child components. It allows the user to interact with the button by pressing it. Result is a touchable button and text that says "Send". When the button is pressed, the handleSubmit function is called, allowing you to perform some action or handle form submission.



```
const styles = StyleSheet.create({
  containerinput: {
    marginBottom: 35,
    flexDirection: 'row',
    justifyContent: 'space-between',
  },
});
export default App;
```

This code declares a constant variable named styles that utilizes the StyleSheet.create function from React Native. It is commonly used to define and organize styles for components in a more structured and efficient manner. Then exports the App component as the default export.

flexDirection: 'row': Sets the direction of the flex container to horizontal, which means its child components will be laid out in a row.
justifyContent: 'space-between': Distributes the child components along the flex container with equal space between them, pushing the first and last components to the edges.



```
const handleSubmit = async () => {
 let dataSend = {
   Gender: gender ? 1 : 0,
   Age: +age,
   cigsPerDay: +cigarettesPerDay,
   totChol: +totalCholesterol,
   sysBp: +systolicBP,
   Glucose: +glucose,
 };
 const postrequest =  axios.post('https://7563-35-236-170-159.ngrok-free.app', dataSend)
   .then(response => {
     console.log('Result:', response.data.result);
     navigation.navigate('SecondPage', {a: dataSend});
   })
   .catch(error => {
     console.error('Error:', error);
   });
};
```

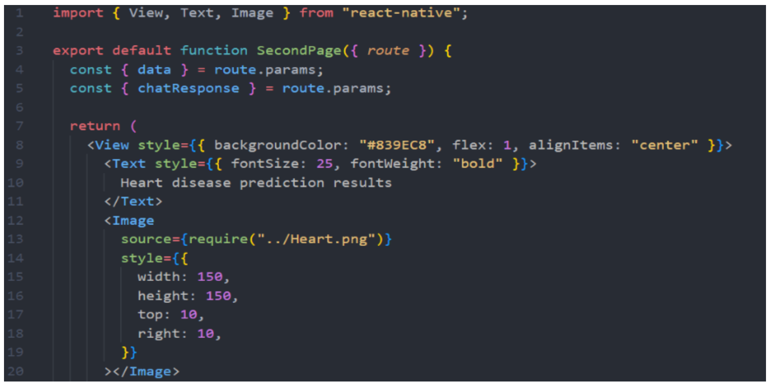
1) **handlesubmit = async () => { ... } :** It is an asynchronous function (async) which suggests that it may contain asynchronous operations such as API calls.

2) **let datasend = { ... }; :** This line declares a variable named datasend and assigns an object to it. The object appears to contain various data properties, The + operator is used before each variable to convert its value to a number.

3) **const postrequest = axios.post('…', datasend) :** This code uses axios, a popular JavaScript library for making HTTP requests, to send a POST request to the specified URL with the datasend object as the request payload.

4) **.then(response => { ... }) :** This is a promise chain that executes when the POST request is successful. The response from the server is passed to the callback function as the response parameter.
navigation.navigate('SecondPage', { a: datasend }); : It navigates to the 'SecondPage' using the navigation object. It passes the datasend object as route parameters with the key 'a'.

5) **.catch(error => { ... }) :** This code executes if there is an error in the POST request. The error object is passed to the callback function as the error parameter.

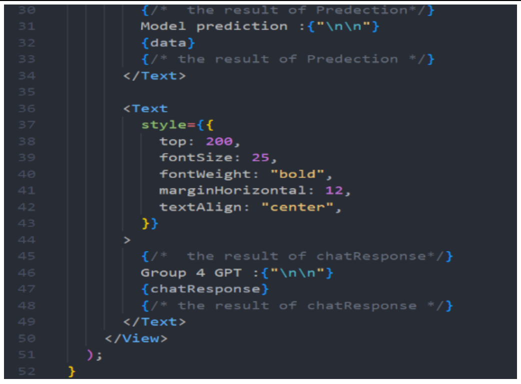


```
import { View, Text, Image } from "react-native";

export default function SecondPage({ route }) {
  const { data } = route.params;
  const { chatResponse } = route.params;

  return (
    <View style={{ backgroundColor: "#839EC8", flex: 1, alignItems: "center" }}>
      <Text style={{ fontSize: 25, fontWeight: "bold" }}>
        Heart disease prediction results
      </Text>
      <Image
        source={require("../Heart.png")}
        style={{
          width: 150,
          height: 150,
          top: 10,
          right: 10,
        }}
      ></Image>
```

1) **export default function SecondPage({ route }) { ... }:** This code exports a default functional component named SecondPage. It receives the route prop as an argument, which is likely passed from a navigation component.

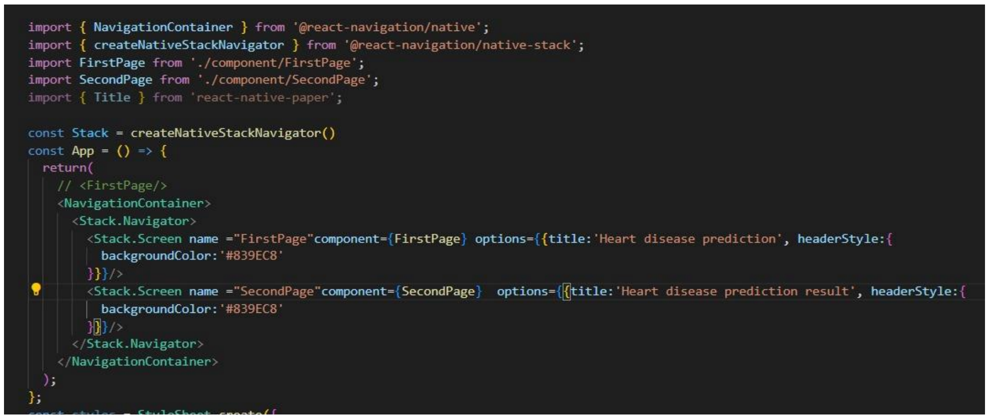
2) **const { data } = route.params; :** This line extracts the data parameter from the route.params object. It uses destructuring assignment to assign the value of route.params.data to the data variable. This allows accessing the data parameter within the component.

3) **const { chatResponse } = route.params;** : This line extracts the chatResponse parameter from the route.params object. It uses destructuring assignment to assign the value of route.params.chatResponse to the chatResponse variable. This allows accessing the chatResponse parameter within the component.



```
          {/*  the result of Predection*/}
          Model prediction :{"\n\n"}
          {data}
          {/* the result of Predection */}
        </Text>

        <Text
          style={{
            top: 200,
            fontSize: 25,
            fontWeight: "bold",
            marginHorizontal: 12,
            textAlign: "center",
          }}
        >
          {/*  the result of chatResponse*/}
          Group 4 GPT :{"\n\n"}
          {chatResponse}
          {/* the result of chatResponse */}
        </Text>
      </View>
    );
}
```

{data}: This expression renders the value of the data variable within the <Text> component. It could be a dynamically generated text or a variable containing a response from the model. **{chatResponse}:** This expression renders the value of the chatResponse variable within the <Text> component. It could be a dynamically generated text or a variable containing a response from the chatGPT.



```
import { NavigationContainer } from '@react-navigation/native';
import { createNativeStackNavigator } from '@react-navigation/native-stack';
import FirstPage from './component/FirstPage';
import SecondPage from './component/SecondPage';
import { Title } from 'react-native-paper';

const Stack = createNativeStackNavigator()
const App = () => {
  return(
    // <FirstPage/>
    <NavigationContainer>
      <Stack.Navigator>
        <Stack.Screen name ="FirstPage"component={FirstPage} options={{title:'Heart disease prediction', headerStyle:{
          backgroundColor:'#839EC8'
        }}}/>
        <Stack.Screen name ="SecondPage"component={SecondPage}  options={{title:'Heart disease prediction result', headerStyle:{
          backgroundColor:'#839EC8'
        }}}/>
      </Stack.Navigator>
    </NavigationContainer>
  );
};
```

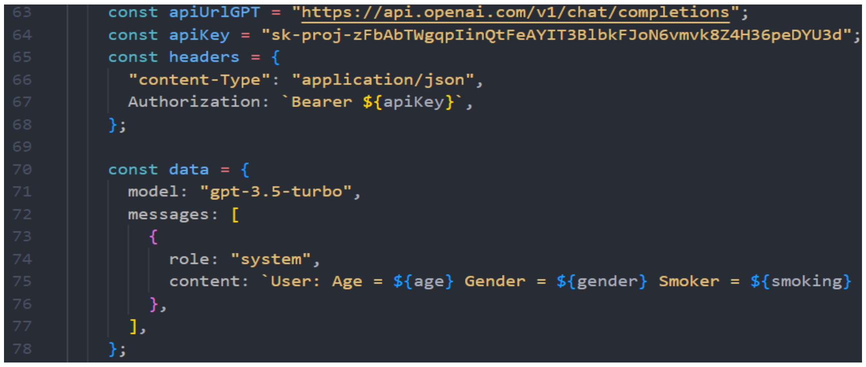
1) **import { NavigationContainer } from '@react-navigation/native'; :** The NavigationContainer is a container component that manages the navigation tree for the app.

2) **import { createNativeStackNavigator } from '@react-navigation/native-stack'; :** It is used to create a native stack navigator, which provides a way to navigate between screens using a stack-based navigation model.

3) **const Stack = createNativeStackNavigator(); :** This line creates a stack navigator and assigns it to the Stack constant. The Stack constant represents the navigator that will manage the navigation stack.

4) **<Stack.Screen name="FirstPage" component={FirstPage} options={...} />:** This Stack.Screen component represents the first screen in the navigation stack. It specifies the name of the screen, the FirstPage component to render, and additional options for the screen. **component={FirstPage}:** Specifies the FirstPage component to render when this screen is navigated to. **options={...}:** Provides options for configuring the screen's appearance and behavior.

- Group 4 GPT:



```
const apiUrlGPT = "https://api.openai.com/v1/chat/completions";
const apiKey = "sk-proj-zFbAbTWgqpIinQtFeAYIT3BlbkFJoN6vmvk8Z4H36peDYU3d";
const headers = {
  "content-Type": "application/json",
  Authorization: `Bearer ${apiKey}`,
};

const data = {
  model: "gpt-3.5-turbo",
  messages: [
    {
      role: "system",
      content: `User: Age = ${age} Gender = ${gender} Smoker = ${smoking}
    },
  ],
};
```

1) **const apiUrlGPT = "…"; :** This URL represents the endpoint for making chat completions using the OpenAI API.

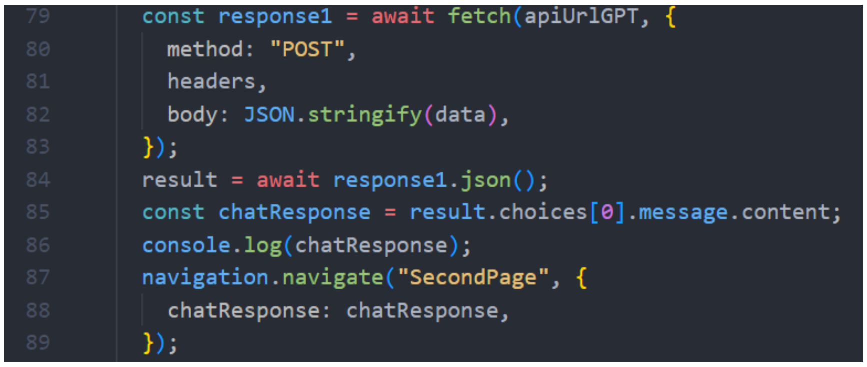
2) **const apiKey = "…";** : The API key is used for authentication and authorization when making requests to the OpenAI API.

3) **const headers = {…}; :** The headers object specifies the content type as JSON ("content-Type": "application/json") and includes the Authorization header with the API key using the Bearer authentication scheme.

4) **const data = { ... }:** The data object contains the following properties:
   - model: "gpt-3.5-turbo": This property specifies the model to be used for the chat completion.

   - messages: [{ ... }]: This property is an array of message objects. There is a single message object with the following properties:
   - role: "system": This property specifies the role of the message, which is set as "system".

   - content: `User: Age = ${age} Gender = ${gender}… Can you predict if this person may have heart disease or not(Direct answer)?`: This property contains the content of the message. It includes placeholder variables (${age}, ${gender}, etc.) that are likely intended to be replaced with actual values when making the API request. The message represents a user query providing various health-related parameters and asking if the person may have heart disease or not.



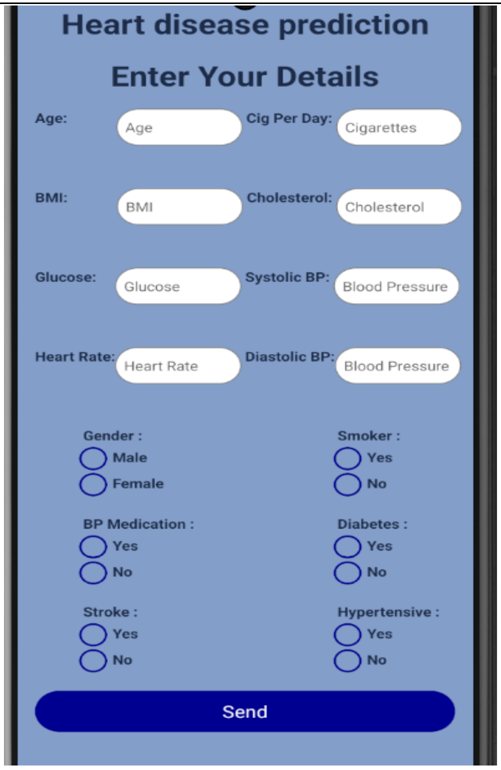
```
const response1 = await fetch(apiUrlGPT, {
  method: "POST",
  headers,
  body: JSON.stringify(data),
});
result = await response1.json();
const chatResponse = result.choices[0].message.content;
console.log(chatResponse);
navigation.navigate("SecondPage", {
  chatResponse: chatResponse,
});
```

1) **const response1 = await fetch(apiUrlGPT, { ... }):** This line makes an HTTP POST request to the apiUrlGPT endpoint, which represents the OpenAI chat completions API. The request is made using the fetch function, which is a built-in JavaScript function for making HTTP requests. The await keyword is used to wait for the response before proceeding to the next line.

   - apiUrlGPT: This is the endpoint URL for the OpenAI chat completions API, as defined earlier in the code.
   - { method: "POST", headers, body: JSON.stringify(data) }: This is the configuration object passed as the second argument to the fetch function. It specifies that the request should be a POST method and includes the headers and the request body. The headers and body variables are defined earlier in the code.

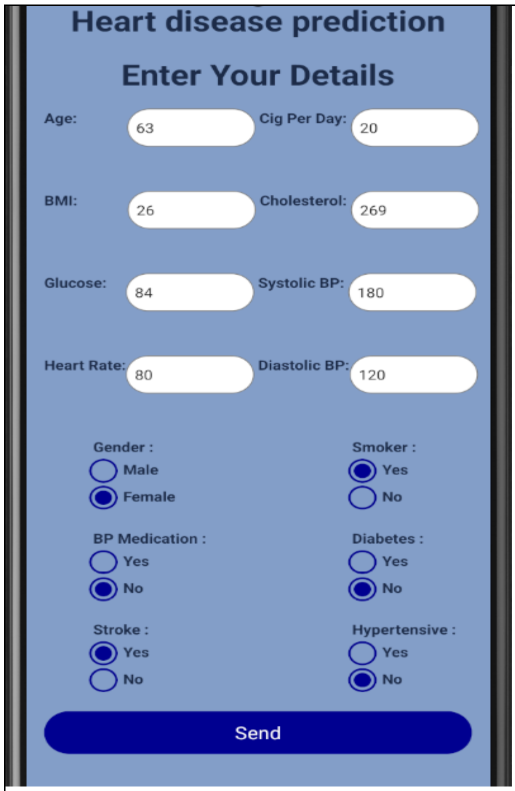
2) **result = await response1.json(); :** This line extracts the JSON data from the response1 object. The json() method is called on the response1 object, which returns a promise that resolves to the parsed JSON data.

3) **const chatResponse = result.choices[0].message.content; :** This line extracts the content of the message from the result object. The result object contains the response data received from the OpenAI chat model. The choices property is an array of possible completions, and in this case, we are accessing the first completion ([0]). The message property within the completion object contains the generated message from the chat model, and the content property within the message object contains the actual text of the generated response. The extracted response content is assigned to the chatResponse variable.



Here's a brief explanation of each user inputs:

- **Demographic:**
  - Gender: male or female
  - Age: age of the patient
- **Behavioral:**
  - Current Smoker: whether or not the patient is a current smoker
  - Cigs Per Day: the number of cigarettes that the person smoked in one day
- **Medical(history):**
  - BP Meds: whether or not the patient was on blood pressure medication
  - Prevalent Stroke: whether or not the patient had previously had a stroke
  - Prevalent Hyp: whether or not the patient was hypertensive
  - Diabetes: whether or not the patient had diabetes
- **Medical(current):**
  - Tot Chol: total cholesterol level
  - Sys BP: systolic blood pressure
  - Dia BP: diastolic blood pressure
  - BMI: Body Mass Index
  - Heart Rate: heart rate
  - Glucose: glucose level

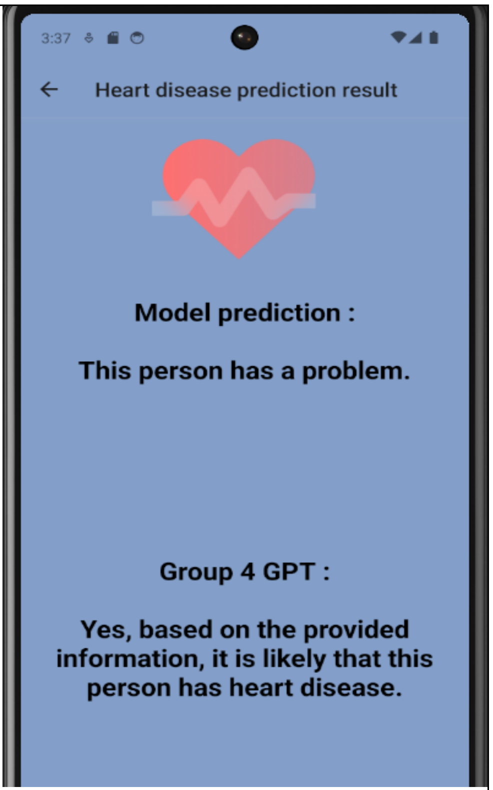


Heart disease prediction

Enter Your Details

Age:	63	Cig Per Day:	20
BMI:	26	Cholesterol:	269
Glucose:	84	Systolic BP:	180
Heart Rate:	80	Diastolic BP:	120

Gender : Male [✓ Female]
Smoker : [✓ Yes] No
BP Medication : Yes [✓ No]
Diabetes : Yes [✓ No]
Stroke : [✓ Yes] No
Hypertensive : Yes [✓ No]



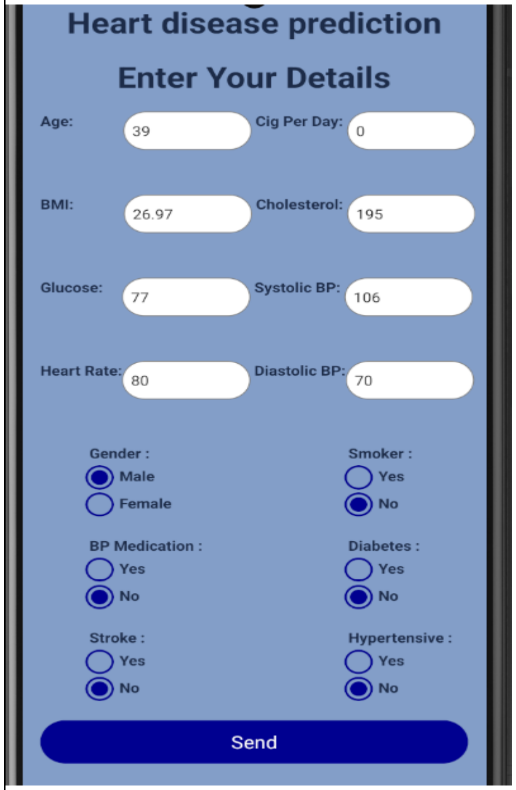
Heart disease prediction result

Model prediction :

This person has a problem.

Group 4 GPT :

Yes, based on the provided information, it is likely that this person has heart disease.

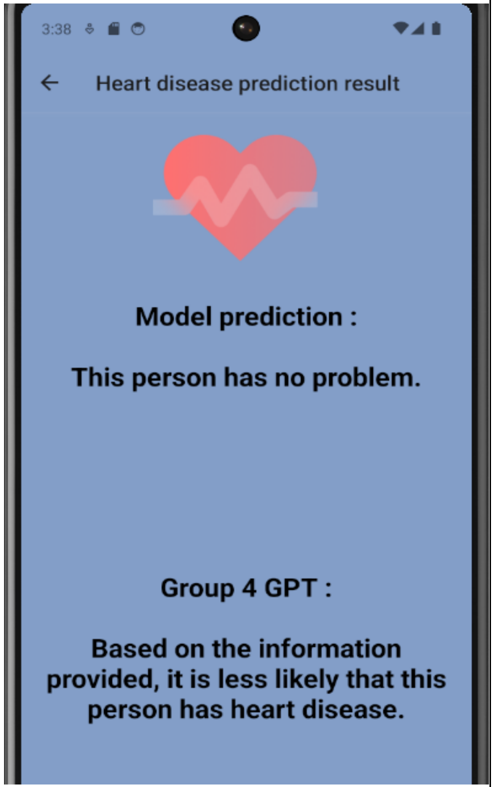


Heart disease prediction

Enter Your Details

Age:	39	Cig Per Day:	0
BMI:	26.97	Cholesterol:	195
Glucose:	77	Systolic BP:	106
Heart Rate:	80	Diastolic BP:	70

Gender : [✓ Male] Female
Smoker : Yes [✓ No]
BP Medication : Yes [✓ No]
Diabetes : Yes [✓ No]
Stroke : Yes [✓ No]
Hypertensive : Yes [✓ No]



Heart disease prediction result

Model prediction :

This person has no problem.

Group 4 GPT :

Based on the information provided, it is less likely that this person has heart disease.

Based on the patient data the model and GPT predicts whether a person may have heart disease or not.

Project Link:

Project model and API :
https://colab.research.google.com/drive/1mS9fAVdNAfAOuUns3O_u-tgTZ-0dnhWs#scrollTo=b3RJEyF7NUCL

Project codes using React Native:
https://drive.google.com/drive/folders/15XR1MkUGMBo3izPzP4mLi-vZosDxq-VL?usp=sharing