Predictive Analysis of Type 2 Diabetes Mellitus Using Machine Learning

Indiana University Indianapolis, Indianapolis, IN 46202, USA,
shfnu@iu.edu, miccperr@iu.edu, saikvemu@iu.edu

Abstract: This project analyzes the different risk factors associated with diabetes. The goal is to identify which factors most directly correlate with diabetes to give healthcare providers and policymakers a tool to identify high-risk individuals. Data was cleaned using data capping and obvious outliers were removed followed by a multicollinearity check. Based on that, the weight and height column were removed as BMI already includes height and weight in its calculation. On data analysis, the dataset was found to be highly imbalanced with 5,043 patients as non-diabetic and 343 as diabetic. The highly imbalanced data was compensated for using SMOTE. Further Hyperparameter Tuning was completed using GridSearchCV and all the models were evaluated using classification reports, ROC curves, and AUCs.

Keywords: Diabetes, gender, pulse rate, systolic blood pressure, diastolic blood pressure, glucose, height, weight, body mass index, family history, data analysis, data visualization, SQL, Python

1 Project Scope

1.1 Introduction

Type 2 Diabetes Mellitus (T2DM) is a prevalent metabolic disorder characterized by high blood sugar levels, often resulting from insulin resistance and insufficient insulin production. Early prediction of T2DM is crucial in preventing its complications, which include cardiovascular disease, kidney damage, and neuropathy. This project aims to develop a machine-learning model to predict the risk of developing T2DM using patient demographics, clinical parameters, and medical history. The goal is to provide healthcare providers and policymakers with a tool to identify high-risk individuals and implement targeted interventions for diabetes prevention.

1.2 Aim

To create a predictive model that accurately identifies individuals at risk of developing Type 2 Diabetes Mellitus based on clinical and demographic factors.

1.2.1 Null Hypothesis

Null Hypothesis (H_0): The independent variables (features) used in the model do not significantly predict diabetes outcomes.

1.2.2 Alternative Hypothesis

Alternate Hypothesis (H_1): The independent variables (features) used in the model significantly predict diabetes outcomes.

1.3 Purpose

This study will empower healthcare providers with a decision-support tool that predicts T2DM risk, allowing for early interventions and personalized patient management strategies. Identifying key risk factors will enable targeted strategies to reduce the prevalence of T2DM and improve patient outcomes.

2 Methodology

2.1 Steps of the Project

1. **Identify the Type of Study**: Quantitative analysis to focus on predicting Type 2 Diabetes risk using machine learning techniques
2. **Data Collection**: The identified dataset contains 5,437 patient records, including demographics, clinical parameters, and medical history
3. **Data Analysis Tools**: Python libraries such as pandas, numpy, scikit-learn, matplotlib, and seaborn will be used for data analysis and visualization

2.2 Original Team Members and Responsibilities

Name	Background	Responsibilities
FNU Sheikh Azam Uddin	• Bachelor of Dental Surgery • MSc. Health Informatics • SQL, Python	1. Helped in data cleaning 2. Exploratory Data Analysis 3. Helped in Training of Models
Sai Kumar Vemula	• Bsc. Radiology and Imaging technology • Msc. Health Informatics • New to SQL and Python	1. Model training 2. Model refining 3. Weight checks and comparing models
Michael Perry	• BS in Security Informatics • SQL, Python	1. SQL 2. Python 3. Data import into myPHPAdmin 4. Cleaning data in myPHPAdmin

2.3 Actual Contribution from Individual Team Members

Name	Background	Responsibilities
FNU Sheikh Azam Uddin	• Bachelor of Dental Surgery • MSc. Health Informatics • SQL, Python	1. Helped in data cleaning 2. Exploratory Data Analysis 3. Helped in Training of Models
Sai Kumar Vemula	• Bsc. Radiology and Imaging technology • Msc. Health Informatics • New to SQL and Python	1. Model training 2. Model refining 3. Weight checks and comparing models 4. PowerPoint
Michael Perry	• BS in Security Informatics • SQL, Python	1. Data import into myPHPAdmin 2. Started cleaning of data 3. Report creation with data analysis from Python workbook

2.4 Project Challenges

The largest challenge we had throughout was dataset based. The initial draft included a COVID-19 dataset. It was recommended this dataset should be dropped and through the search for other datasets that fit the required standards the one for this project was found. Once analysis on the dataset started it became clear that most of the dataset did not have diabetes. This extreme imbalance would have caused altered results so the use of SMOTE was recommended to create synthetic data for diabetic patients. Another challenge faced was the underutilization of communication. While a group chat was created to keep each other up to date on progress, it was not utilized to its full potential, so meetings often felt like catching each other up rather than making progress. Although this was a challenge it is not felt that it caused any actual issues with the project, rather just more work to catch each other up on progress made since the last meeting.

3 Data Collection

The dataset is from DiaHealth: A Bangladeshi Dataset for Type 2 Diabetes Prediction (https://data.mendeley.com/datasets/7m7555vgrn/1). “This dataset provides comprehensive information on 5,437 patients, including 14 independent attributes such as demographics, clinical parameters, and medical history” (Prama et al., 2024). This dataset is a single csv file with all data included in the file.

4 Data Extraction and Storage

The entire dataset is going to be imported into phpMyAdmin with cleaning taking place after import. The entire dataset is broken down into 15 columns listed below.

Columns included in the dataset:

1. age
2. gender
3. pulse_rate
4. systolic_bp
5. diastolic_bp
6. glucose
7. height
8. weight
9. bmi
10. family_diabetes
11. hypertensive
12. family_hypertsion
13. cardiovascular_disease
14. stroke
15. diabetic

Descriptions of the specific columns listed above:

- Continuous Values
  - age
  - pulse_rate
  - systolic_bp
  - diastolic_bp
  - glucose
  - height
  - weight
  - BMI
- Binary Values
  - gender
  - family_diabetes
  - hypertensive
  - family_hypertension
  - cardiovascular_disease
  - stroke
  - diabetic

4.1 Data Import

Data from the single "Diabetes_Final_Data_V2.csv" file were imported into the groups database listed as "I501_Fall2024_Sec27791_group04_db" and then Import was used. Importing the "Diabetes_Final_Data_V2.csv" file into the current database was completed. The first import failed because the Body Mass Index (BMI) had two outliers in the triple digits (574.13, 155.74) when the column type defaulted to VARCHAR(4). The step to fix this is outlined below in the data cleaning section. After this issue was resolved the import was completed a second time. This time no failures occurred, and 5438 rows were successfully imported into the database.

4.2 Data Cleaning

All data cleansing sets are reviewed below. All data columns listed above we kept as all would add valuable information for the data analysis portion of the project.

The initial data import failed as the BMI values were too long for the default VARCHAR(4). The type was changed to VARCHAR(5) to successfully import the csv file. Once all data was imported there was a duplicate row of headers below the initial headers. This was removed with the below statement.

```
DELETE FROM diabetes_final_data_v2
WHERE age = 'age';
```

The above SQL statement was run with the simulate option in phpMyAdmin. Since DELETE is a powerful command, we wanted to make sure only the one row we wanted to delete would be affected. Once confirming only, the top row would be affected it was run, and the top row duplicate was removed from the database.

All data was imported as the VARCHAR data type and each of the following data columns were switched to the below types:

Row Number	Row Title	Updated Type
1	age	int(11)
2	gender	varchar(6)
3	pulse_rate	int(11)
4	systolic_bp	int(11)
5	diastolic_bp	int(11)
6	glucose	float
7	height	float
8	weight	float
9	bmi	float
10	family_diabetes	tinyint(1)
11	hypertensive	tinyint(1)
12	family_hypertension	tinyint(1)
13	cardiovascular_disease	tinyint(1)
14	stroke	tinyint(1)
15	diabetic	tinyint(1)

The ALTER statement used is below.

```
ALTER TABLE diabetes_final_data_v2
MODIFY age INTEGER,
MODIFY pulse_rate INTEGER,
MODIFY systolic_bp INTEGER,
MODIFY diastolic_bp INTEGER,
```

```
MODIFY glucose FLOAT,
MODIFY height FLOAT,
MODIFY weight FLOAT,
MODIFY bmi FLOAT;
```

The last column in the dataset “diabetic” initially had a value of “Yes” or “No” regarding the person in the dataset having diabetes. We wanted to switch the VARCHAR variable type to TINYINT. To accomplish this, we changed the “Yes” values to “1” and the “No” value to “0” with the two UPDATE SQL statements below.

```
UPDATE diabetes_final_data_v2
SET diabetic = 1
WHERE diabetic = ‘Yes’;

UPDATE diabetes_final_data_v2
SET diabetic = 0
WHERE diabetic = ‘No’;
```

After updating all variable types across columns, the describe() method from Pandas was used on the DataFrame. This lists the count, mean, std, min, 25%, 50%, 75% and max for each column. The two following figures, Figure 1 and Figure 2, display the data before most of the cleaning was completed. Only the aforementioned changes are present.

	age	pulse_rate	systolic_bp	diastolic_bp	glucose	height	weight	bmi
count	5437.000000	5437.000000	5437.000000	5437.000000	5437.000000	5437.000000	5437.000000	5437.000000
mean	45.533750	76.768990	133.859849	82.064742	7.540682	1.548571	53.626816	22.472301
std	14.321155	12.290076	22.293015	12.489593	2.923080	0.080955	10.091550	8.778764
min	8.000000	5.000000	62.000000	45.000000	0.000000	0.360000	3.000000	1.220000
25%	35.000000	69.000000	119.000000	73.000000	6.000000	1.520000	46.700000	19.630000
50%	45.000000	76.000000	130.000000	81.000000	6.920000	1.550000	53.000000	21.870000
75%	55.000000	84.000000	147.000000	90.000000	8.120000	1.600000	59.900000	24.490000
max	112.000000	133.000000	231.000000	119.000000	33.460000	1.960000	100.700000	574.130000

Figure 1. describe() results pt. 1

family_diabetes	hypertensive	family_hypertension	cardiovascular_disease	stroke	diabetic
5437.000000	5437.000000	5437.000000	5437.000000	5437.000000	5437.000000
0.037337	0.109803	0.039544	0.011587	0.003678	0.063270
0.189603	0.312673	0.194903	0.107029	0.060545	0.243471
0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

Figure 2. describe() results pt. 2

One of the most prominent outliers existed within the BMI column. With a maximum value of 574.13 this erratic value fell far outside of the realistic values for BMI. Visually shown in Figure 3 in a boxplot there are multiple values for BMI outside of realistic values.

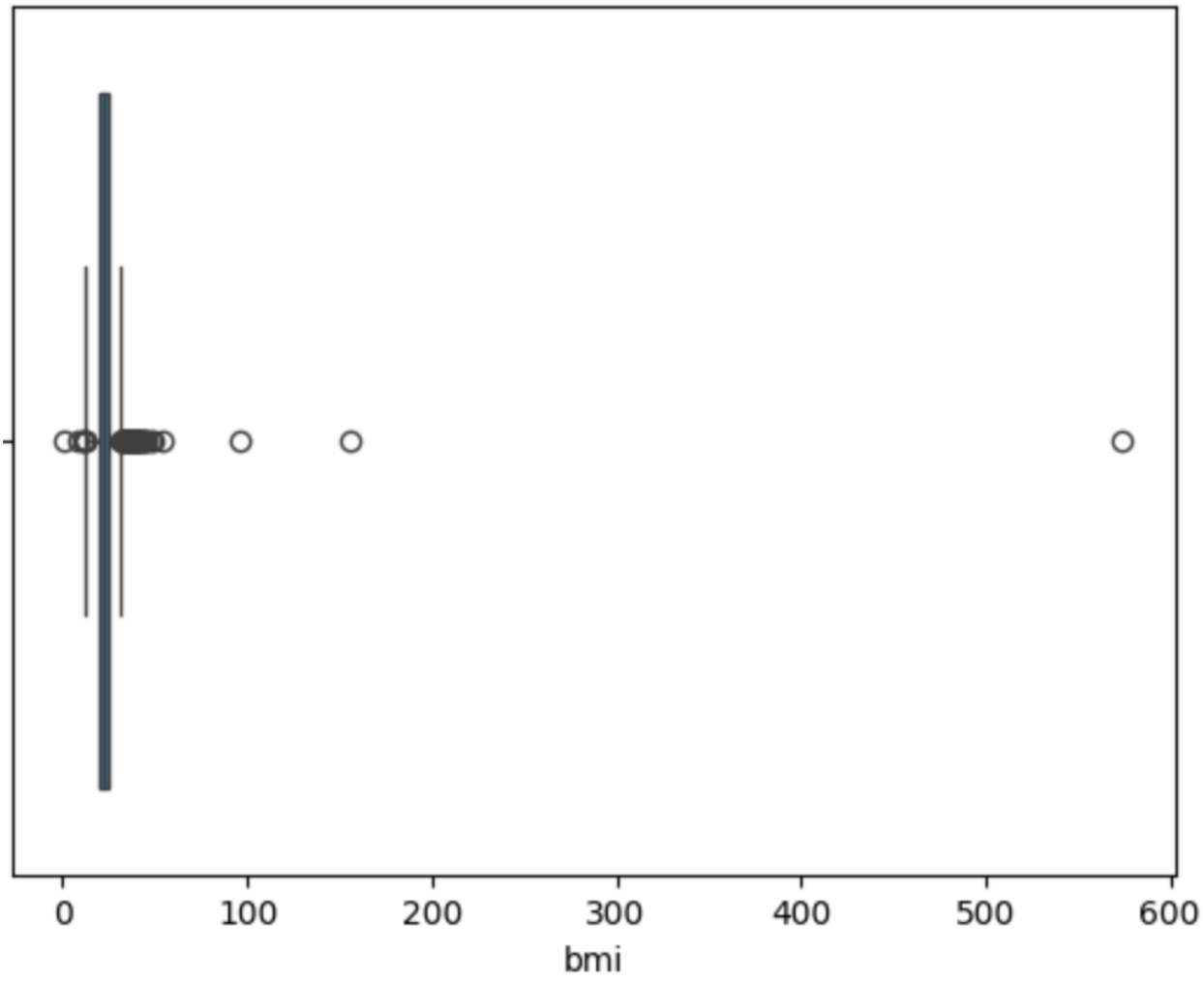

Figure 3. BMI boxplot

BMI is found using a calculation including height and weight. Given upper and lower bounds found to exclude noise within the dataset the decision to recalculate all BMI values was made. Before recalculating the BMI values a lower and upper bound was found for 4 of our parameters.

Parameter	Lower Bound	Upper Bound
Height (m)	1m	2.50 m
Weight (kg)	30 kg	300 kg
Pulse Rate (bpm)	40 bpm	180 bpm
Glucose (mmol/L)	2 mmol/L	30 mmol/L

Figure 4. Lower and upper bounds

Using these lower and upper bounds displayed in Figure 4 as limits all data below and above the listed bounds was deleted as it was considered noise. This process was completed using the lower and upper bounds as limits for the creation of a new DataFrame with erratic data removed. This was completed using the code in Figure 5.

```
limits = {
    'height': (1, 2.5),
    'weight': (30, 300),
    'pulse_rate': (40, 180),
    'glucose': (2, 30)
}

df_noiseless = df.copy()

for column, (lower, upper) in limits.items():
    if column in df_noiseless.columns:
        df_noiseless = df_noiseless[(df_noiseless[column] >= lower) & (df_noiseless[column] <= upper)]
    else:
        print(f"Column '{column}' not found in the DataFrame. Skipping.")

print(f"Cleaned DataFrame: {len(df_noiseless)}")
df_noiseless
```

Figure 5. New DataFrame creation with noise removed

This cleansing process dropped the original number of rows from 5,437 down to 5,386 during the erratic data portion. These 5,386 rows are therefore the final number of rows after our cleaning was complete, this process reduced the total number of rows by 51 during this process, along with the two rows mentioned above for a total of 53 rows removed from the original dataset. Using the same describe() method from above the data this time has erratic values removed and displayed in Figure 6 and Figure 7.

	age	pulse_rate	systolic_bp	diastolic_bp	glucose	height	weight	bmi
count	5386.000000	5386.000000	5386.000000	5386.000000	5386.000000	5386.000000	5386.000000	5386.000000
mean	45.493316	76.826216	133.800037	82.051244	7.563754	1.549720	53.703063	22.351444
std	14.257717	12.136131	22.255133	12.505997	2.843382	0.076742	10.011741	4.076193
min	8.000000	40.000000	62.000000	45.000000	2.000000	1.190000	30.000000	12.100000
25%	35.000000	69.000000	119.000000	73.000000	6.000000	1.520000	46.825000	19.662500
50%	45.000000	76.000000	130.000000	81.000000	6.925000	1.550000	53.000000	21.870000
75%	55.000000	84.000000	147.000000	90.000000	8.120000	1.600000	60.000000	24.490000
max	112.000000	133.000000	231.000000	119.000000	30.000000	1.960000	100.700000	54.080000

Figure 6. describe() results pt. 1 after noise cleaning

family_diabetes	hypertensive	family_hypertension	cardiovascular_disease	stroke	diabetic
5386.000000	5386.000000	5386.000000	5386.000000	5386.000000	5386.000000
0.037319	0.110100	0.039547	0.011697	0.003713	0.063684
0.189560	0.313044	0.194910	0.107528	0.060830	0.244211
0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

Figure 7. describe() results pt. 2 after noise cleaning

The newly cleansed data was checked again for any null data using .insnull(). None of the 15 columns had any null values and all the values were next checked for uniqueness with results displayed in Figure 8.

```
age                    84
gender                  2
pulse_rate             86
systolic_bp           136
diastolic_bp           73
glucose               912
height                 51
weight                502
bmi                  1373
family_diabetes         2
hypertensive            2
family_hypertension     2
cardiovascular_disease  2
stroke                  2
diabetic                2
```

Figure 8. Number of unique values per column

All the columns with 2 unique values were binary values of either 0 for false and 1 for positive or male and female under gender. All other columns had unique values much larger than 2. Finally duplicate values were checked, and no duplicate values were found. The first step was to copy the DataFrame that had noise removed (df_noiseless) into a new final DataFrame that had all data cleaned (df_cleaned) as shown in the code in Figure 9.

```
df_cleaned = df_noiseless.copy()
```

Figure 9. Creation of cleaned DataFrame from the noiseless DataFrame

After the noise was removed and data was checked for null values, unique values, and duplicate values the detection of outliers was completed using Interquartile Range (IQR) statistical method with visualization through boxplots and histograms.

An example of this process for glucose can be seen in the code below in Figure 10.

```
# glucose

Q1 = df_noiseless['glucose'].quantile(0.25)
Q3 =  df_noiseless['glucose'].quantile(0.75)
IQR = Q3 - Q1
lower_bound_g = Q1 - 1.5 * IQR
upper_bound_g = Q3 + 1.5 * IQR
```

Figure 10. Code for IQR on glucose

Once these new lower and upper bounds were created with IQR they were plotted into histograms comparing the noiseless DataFrame with the new cleaned DataFrame.

```
fig, axes = plt.subplots(1, 2, figsize=(16, 6))

sns.histplot(df_noiseless['glucose'], kde=True, color='blue', ax=axes[0])
axes[0].set_title('Glucose Distribution - Noiseless Data', fontsize=16, fontweight='bold')
axes[0].set_xlabel('Glucose Level (mmol/L)', fontsize=12)
axes[0].set_ylabel('Density', fontsize=12)
axes[0].grid(True, linestyle='--', linewidth=0.5, alpha=0.7)

sns.histplot(df_cleaned['glucose'], kde=True, color='green', ax=axes[1])
axes[1].set_title('Glucose Distribution - Cleaned Data', fontsize=16, fontweight='bold')
axes[1].set_xlabel('Glucose Level (mmol/L)', fontsize=12)
axes[1].set_ylabel('Density', fontsize=12)
axes[1].grid(True, linestyle='--', linewidth=0.5, alpha=0.7)

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

Figure 11. Code used to create subplot histograms for glucose

The code above in Figure 11 was used to create subplots for both histograms with an output of the following.

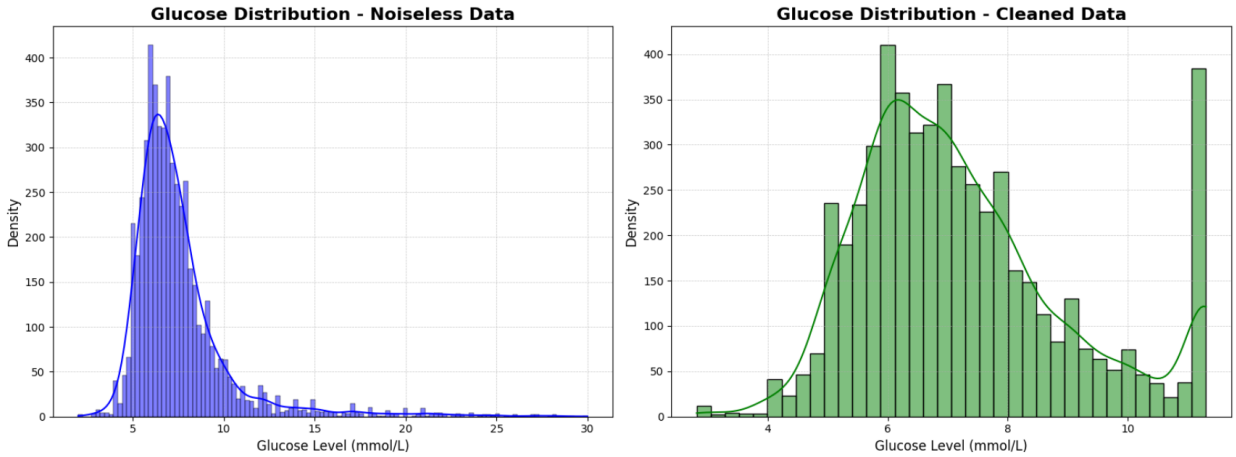

Figure 12. Histograms for noiseless and cleaned glucose datasets

From the histograms in Figure 12 the introduction of IQR capped the outlier data successfully as outlier data greater than 11.3 was removed. This process was repeated with all other columns to remove outliers in the data and to update the cleansed data copy to the new clean DataFrame.

With the cleaned data from the height and weight columns the BMI column was chosen to be updated with the following steps to more accurately describe the cleaned data.

The creation of a new measured BMI column was completed with a restriction to 2 decimal places. This was accomplished using the following code in Figure 13. To drop the existing column and recalculate the new column the code in Figure 14 was used.

```
df_cleaned = df_cleaned.drop('bmi', axis =1)
```

Figure 13. Restricting to two decimal places

```
df_cleaned['BMI'] = df_cleaned['weight'] / (df_cleaned['height'] ** 2)
df_cleaned['BMI'] = df_cleaned['BMI'].round(2)
df_cleaned.head()
```

Figure 14. Recalculating BMI data and adding it back into the dataset

Comparing the aforementioned boxplot before data was cleaned to the following that had all noise removed shows a massive increase in the realistic values and drop in the erratic values.

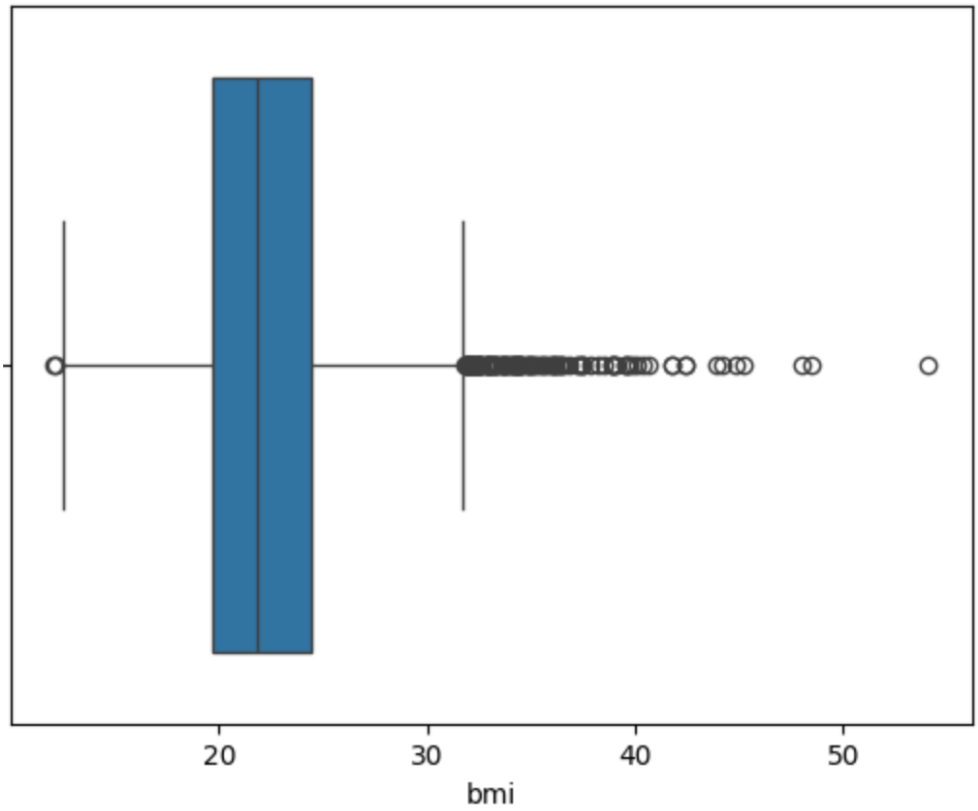

Figure 15. BMI noiseless boxplot

This boxplot from the noiseless DataFrame in Figure 15 can then be compared to the boxplot from the final cleaned DataFrame in Figure 16 where we can see the values larger than 50 were removed furthering the data cleaning process.

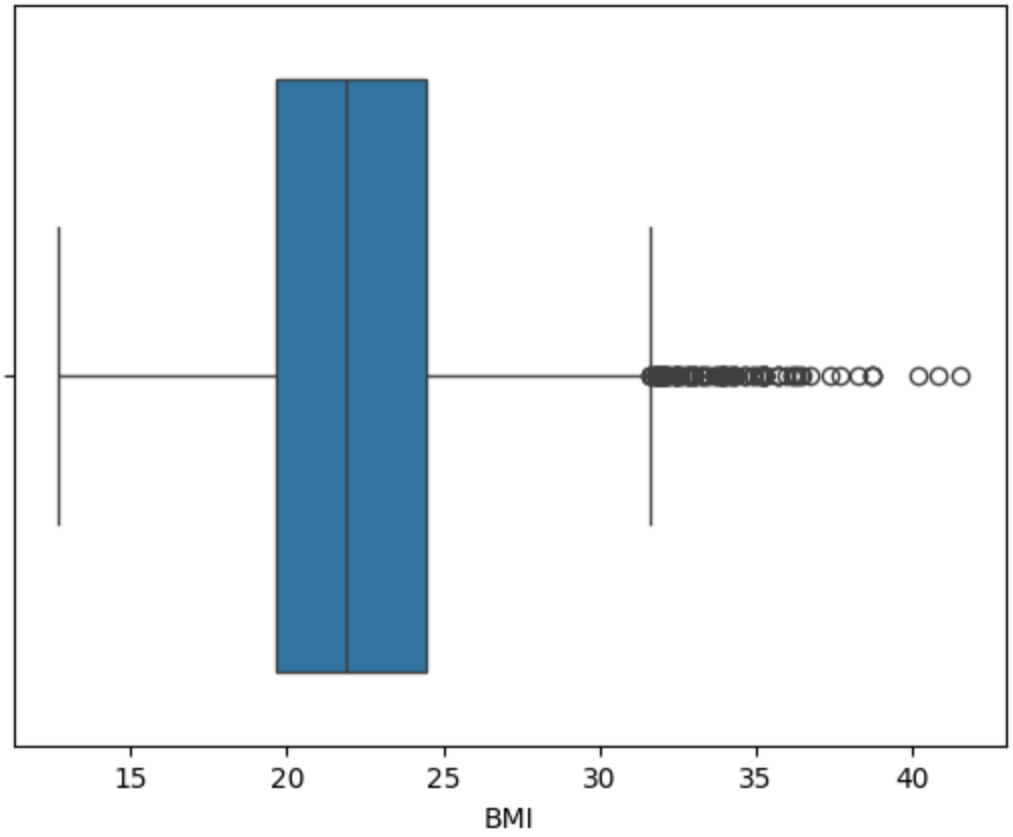

Figure 16. BMI cleaned boxplot

The final describe() method was used on the cleaned DataFrame to show how this multistep process of cleaning truly transformed the data in Figure 17 and Figure 18.

	age	pulse_rate	systolic_bp	diastolic_bp	glucose	height	weight	family_diabetes
count	5386.000000	5386.000000	5386.000000	5386.000000	5386.000000	5386.000000	5386.000000	5386.000000
mean	45.493316	76.728463	133.608429	82.033234	7.258554	1.550663	53.669163	0.037319
std	14.257717	11.818369	21.682615	12.451142	1.775239	0.069915	9.808054	0.189560
min	8.000000	46.500000	77.000000	47.500000	2.820000	1.400000	33.800000	0.000000
25%	35.000000	69.000000	119.000000	73.000000	6.000000	1.520000	46.825000	0.000000
50%	45.000000	76.000000	130.000000	81.000000	6.925000	1.550000	53.000000	0.000000
75%	55.000000	84.000000	147.000000	90.000000	8.120000	1.600000	60.000000	0.000000
max	112.000000	106.500000	189.000000	115.500000	11.300000	1.720000	81.415000	1.000000

Figure 17. describe() results pt. 1 after final cleaning

hypertensive	family_hypertension	cardiovascular_disease	stroke	diabetic	BMI
5386.000000	5386.000000	5386.000000	5386.000000	5386.000000	5386.000000
0.110100	0.039547	0.011697	0.003713	0.063684	22.318099
0.313044	0.194910	0.107528	0.060830	0.244211	3.864863
0.000000	0.000000	0.000000	0.000000	0.000000	12.720000
0.000000	0.000000	0.000000	0.000000	0.000000	19.650000
0.000000	0.000000	0.000000	0.000000	0.000000	21.885000
0.000000	0.000000	0.000000	0.000000	0.000000	24.440000
1.000000	1.000000	1.000000	1.000000	1.000000	41.540000

Figure 18. describe() results pt. 2 after final cleaning

Take note that BMI is now the final column inside of Figure 18 as the original values were replaced with the recalculated step mentioned above.

4.3 Binary Feature Analysis

Binary feature analysis was completed on all the columns with binary values. The code in Figure 19 was run to create count plot for each of the binary columns and visually display a breakdown of the data.

```
import matplotlib.pyplot as plt
import seaborn as sns

fig, axes = plt.subplots(3, 3, figsize=(15, 10))  # 2 rows, 3 columns
axes = axes.flatten()

sns.countplot(data=df_cleaned, x='family_diabetes', ax=axes[0])
axes[0].set_title('Family Diabetes')

sns.countplot(data=df_cleaned, x='hypertensive', ax=axes[1])
axes[1].set_title('Hypertensive')

sns.countplot(data=df_cleaned, x='family_hypertension', ax=axes[2])
axes[2].set_title('Family Hypertension')

sns.countplot(data=df_cleaned, x='cardiovascular_disease', ax=axes[3])
axes[3].set_title('Cardiovascular Disease')

sns.countplot(data=df_cleaned, x='stroke', ax=axes[4])
axes[4].set_title('Stroke')

sns.countplot(data=df_cleaned, x='diabetic', ax=axes[5])
axes[5].set_title('Diabetic')

sns.countplot(data=df_cleaned, x='gender', ax=axes[6])
axes[6].set_title('Gender')


axes[7].axis('off')
axes[8].axis('off')
plt.tight_layout()
plt.show()
```

Figure 19. Code to create count plots for binary columns

With the above code the subplots for the binary columns are in Figure 20.

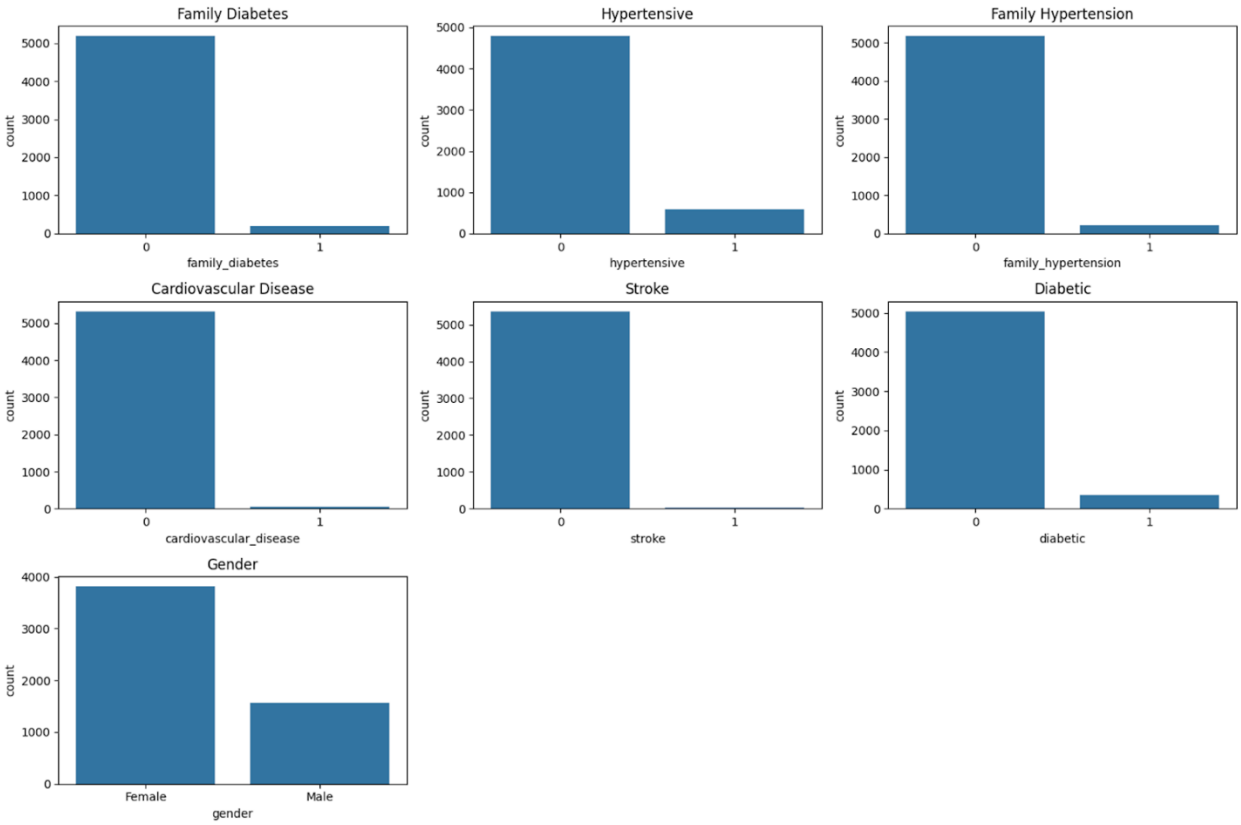

Figure 20. Count plots for binary columns

As evident in the count plots the columns of Family Diabetes, Hypertensive, Family Hypertension, Cardiovascular Disease, Stroke, and Diabetic all lean towards a strong negative weight. Meaning most of the patients did not have a history of any of these conditions. Under gender most of the patients were female and under 1/3 were male.

4.4 Continuous Versus Binary Variables

Box plots were run on the continuous variables with one consisting of non-diabetic patients and one of diabetic patients for each continuous variable. The code in Figure 21 was used to run the box plots.

```
import matplotlib.pyplot as plt
import seaborn as sns

fig, axes = plt.subplots(3, 3, figsize=(20, 15))
axes = axes.flatten()

sns.boxplot(data=df_cleaned, x='diabetic', y='systolic_bp', ax=axes[0])
axes[0].set_title('Systolic Blood Pressure by Diabetes Status')
axes[0].set_xlabel('Diabetes Status')
axes[0].set_ylabel('Systolic BP')

sns.boxplot(data=df_cleaned, x='diabetic', y='diastolic_bp', ax=axes[1])
axes[1].set_title('Diastolic Blood Pressure by Diabetes Status')
axes[1].set_xlabel('Diabetes Status')
axes[1].set_ylabel('Diastolic BP')

sns.boxplot(data=df_cleaned, x='diabetic', y='glucose', ax=axes[2])
axes[2].set_title('Glucose Levels by Diabetes Status')
axes[2].set_xlabel('Diabetes Status')
axes[2].set_ylabel('Glucose (mmol/L)')

sns.boxplot(data=df_cleaned, x='diabetic', y='weight', ax=axes[3])
axes[3].set_title('Weight by Diabetes Status')
axes[3].set_xlabel('Diabetes Status')
axes[3].set_ylabel('Weight (kg)')

sns.boxplot(data=df_cleaned, x='diabetic', y='BMI', ax=axes[4])
axes[4].set_title('BMI by Diabetes Status')
axes[4].set_xlabel('Diabetes Status')
axes[4].set_ylabel('BMI')

sns.boxplot(data=df_cleaned, x='diabetic', y='height', ax=axes[5])
axes[5].set_title('Height by Diabetes Status')
axes[5].set_xlabel('Diabetes Status')
axes[5].set_ylabel('Height (cm)')

sns.boxplot(data=df_cleaned, x='diabetic', y='pulse_rate', ax=axes[6])
axes[6].set_title('Pulse Rate by Diabetes Status')
axes[6].set_xlabel('Diabetes Status')
axes[6].set_ylabel('Pulse Rate (bpm)')

sns.boxplot(data=df_cleaned, x='diabetic', y='age', ax=axes[7])
axes[7].set_title('Age by Diabetes Status')
axes[7].set_xlabel('Diabetes Status')
axes[7].set_ylabel('Age(Year)')


axes[8].axis('off')
plt.tight_layout()
plt.show()
```

Figure 21. Code to run box plots on continuous values

The box plots created from the above code are in Figure 22.

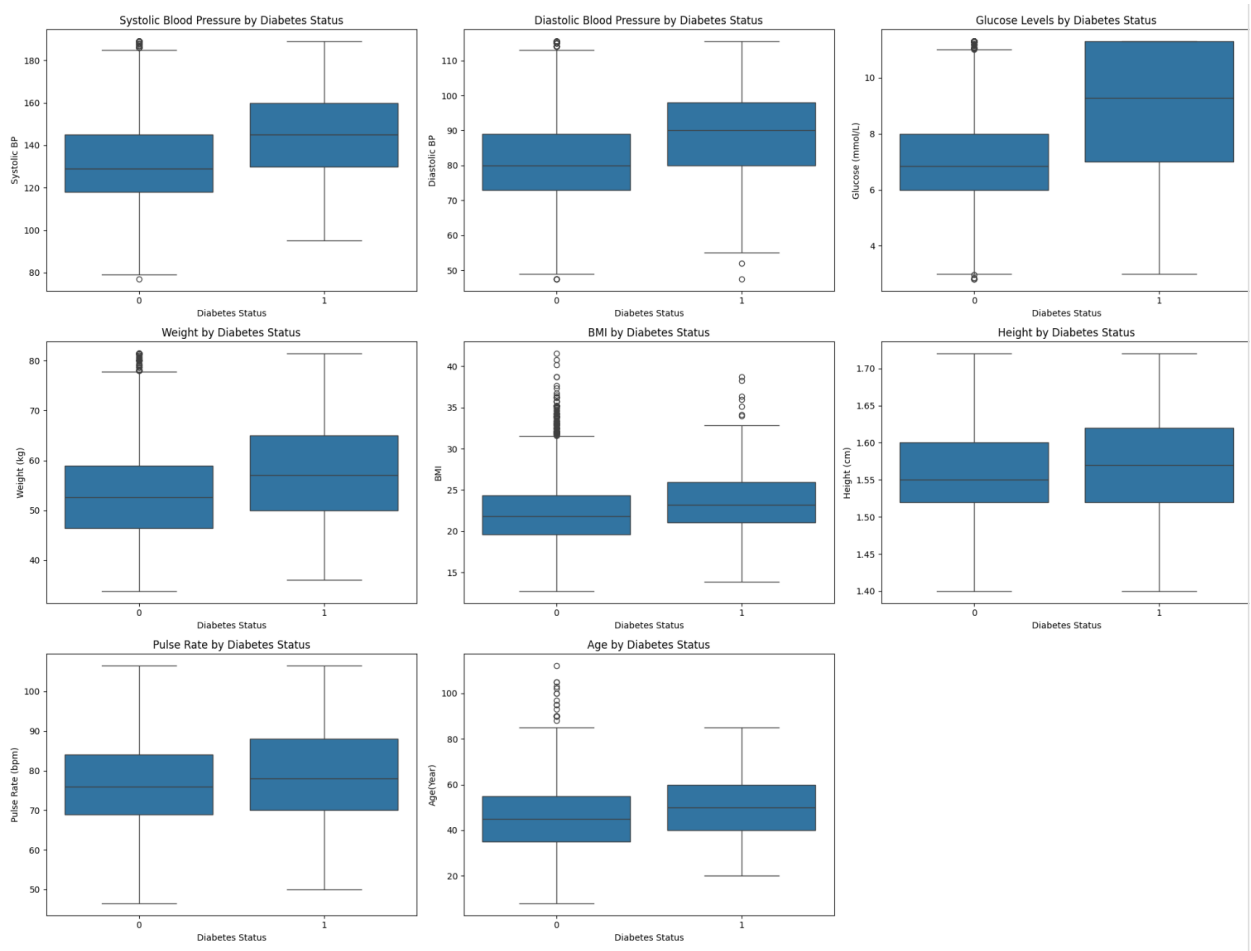

Figure 22. Box plots of all continuous values for diabetic and non-diabetic patients

One of the observations from the diabetic chart was most of the participants did not have diabetes. To further break down the diabetic chart, the non-diabetic patients were given the value of (0) and diabetic patients the value of (1) by gender was done with the code in Figure 23.

```
sns.countplot(data=df_cleaned, x='gender', hue='diabetic', palette='Set2')
plt.title('Diabetic Status by Gender')
plt.xlabel('Gender')
plt.ylabel('Count')
plt.legend(title='Diabetic', labels=['Non-Diabetic (0)', 'Diabetic (1)'])
plt.show()
```

Figure 23. Code to create count plots by gender

Resulting in the following count plot in Figure 24.

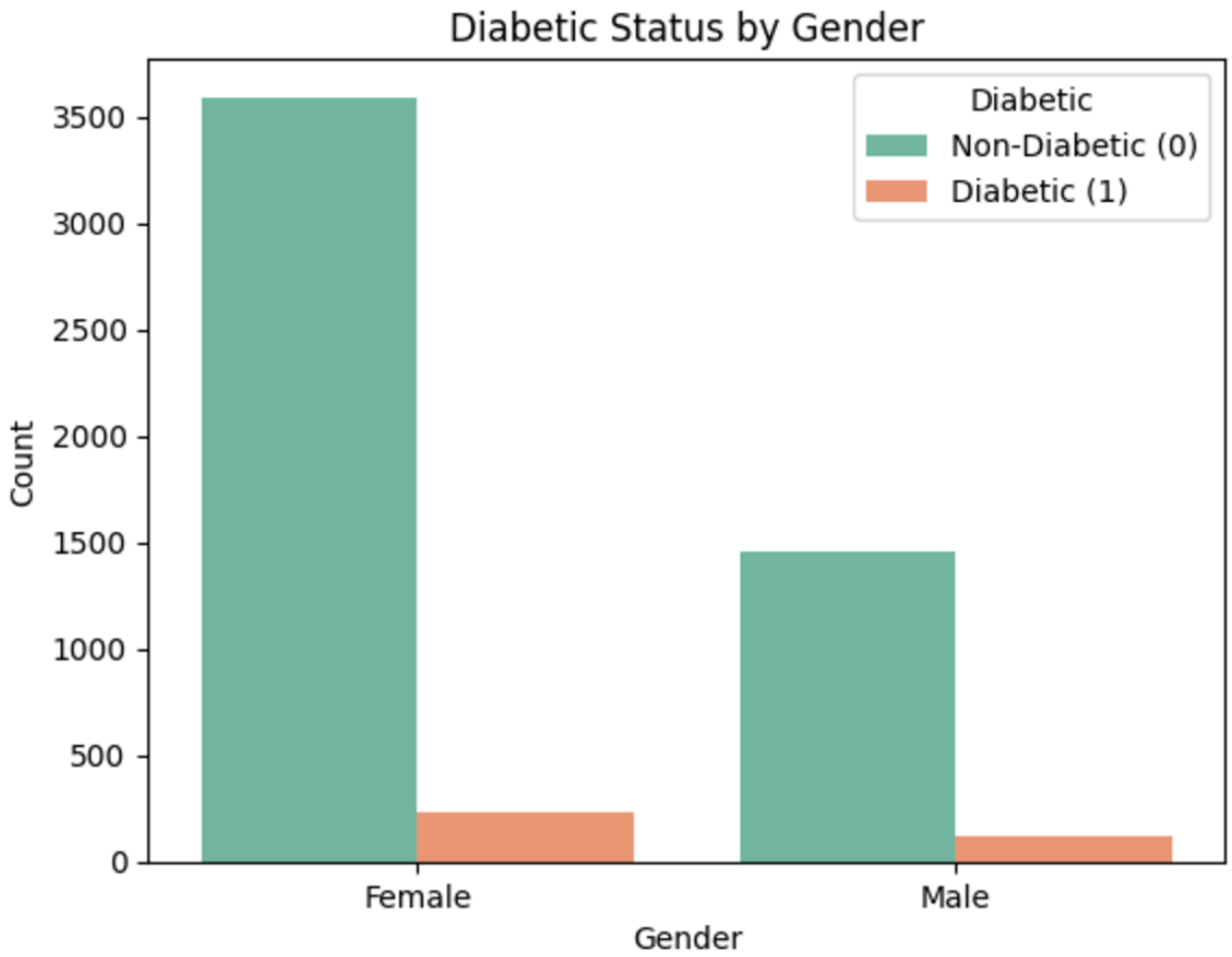

Figure 24. Count plot for non-diabetic and diabetic patients by gender

The count plot broken down by gender further displays how few participants with diabetes there are either female or male. The vast majority consisting of non-diabetic females with a little over 3,500 while non-diabetic males had a little over 1,500.

5 Data Analysis

Python in a Jupyter notebook was used for all analysis. For each of the sections the code used for the analysis will be followed by the results.

5.1 Connecting to database

- For all analysis on the dataset, we connected to phpMyAdmin SQL through a Jupyter notebook.
- Our dataset “diabetes_final_data_v2” was imported into the Jupyter notebook from our group database “I501_Fall2024_Sec27791_group04_db”

- When importing the initial data fetchall() was used and the data was converted into a DataFrame.
- The initial connection to phpMyAdmin, the SELECT statement for "diabetes_final_data_v2" and the formation of the DataFrame are all shown in the Python code in Figure 25.

```
import mysql.connector
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

conn = mysql.connector.connect(
    host="localhost",
    port=3306,
    user="[redacted]",
    passwd="[redacted]",
    database="I501_Fall2024_Sec27791_group04_db")

cursor = conn.cursor()
cursor.execute("SELECT * FROM diabetes_final_data_v2")
data = cursor.fetchall()
columns = [desc[0] for desc in cursor.description]  # Get column names

df = pd.DataFrame(data, columns=columns)
cursor.close()
conn.close()
```

Figure 25. Connecting Jupyter Notebook to myPHPAdmin

5.2 Testing for Normality

A Shapiro-Wilk test was run on each of the continuous data columns testing for normality. The code to run this analysis is below in Figure 26.

```python
from scipy.stats import shapiro

for column in df_cleaned.select_dtypes(include=['float64', 'int64']).columns:
    print(f"Testing normality for: {column}")
    stat, p = shapiro(df_cleaned[column].dropna())  # Drop missing values if any
    print(f"Shapiro-Wilk Test p-value for {column}: {p}")

    if p < 0.05:
        print(f"{column}: Data is not normally distributed.\n")
    else:
        print(f"{column}: Data is normally distributed.\n")
```

Figure 26. Python to run Shapiro-Wilk normality testing

The results from the code are as follows:

1) Age
   a) Shapiro-Wilk Test p-value: 1.6764780813122325e-28
   b) Age data is not normally distributed
2) Pulse Rate
   a) Shapiro-Wilk Test p-value: 1.7430172851066313e-17
   b) Pulse Rate data is not normally distributed
3) Systolic Blood Pressure
   a) Shapiro-Wilk Test p-value: 9.821555052179672e-33
   b) Systolic Blood Pressure data is not normally distributed
4) Diastolic Blood Pressure
   a) Shapiro-Wilk Test p-value: 6.720621343504398e-22
   b) Diastolic Blood Pressure data is not normally distributed
5) Glucose
   a) Shapiro-Wilk Test p-value: 3.60133705331478e-43
   b) Glucose data is not normally distributed
6) Height
   a) Shapiro-Wilk Test p-value: 1.0645847661103395e-30
   b) Height data is not normally distributed
7) Weight
   a) Shapiro-Wilk Test p-value: 4.793637299509779e-23
   b) Weight data is not normally distributed
8) Family Diabetes
   a) Shapiro-Wilk Test p-value:
   b) Family Diabetes data is not normally distributed
9) BMI
   a) Shapiro-Wilk Test p-value: 1.2814557061703813e-29
   b) Data is not normally distributed

Given the above results none of our data points are normally distributed.

5.3 Mann-Whitney U Test for Continuous Variables

Since all data was found to not be normally distributed using the Shapiro-Wilk tests the Mann-Whitney U Test was ran on the continuous variables to find if there is a significant different between groups. This test was run using the following code in Figure 27.

```
from scipy.stats import mannwhitneyu

# List of continuous variables to test
continuous_columns = ['age', 'pulse_rate', 'systolic_bp', 'diastolic_bp',
                      'glucose', 'height', 'weight', 'BMI']

# Perform Mann-Whitney U Test for each continuous variable based on diabetic status (0/1)
for column in continuous_columns:
    print(f"Performing Mann-Whitney U Test for '{column}' and 'diabetic':")

    # Filter the data by diabetic status (1 for diabetic, 0 for non-diabetic)
    group1 = df_cleaned[df_cleaned['diabetic'] == 1][column].dropna()  # Diabetic group
    group2 = df_cleaned[df_cleaned['diabetic'] == 0][column].dropna()  # Non-diabetic group

    # Check if either group is empty
    if group1.empty or group2.empty:
        print(f"One of the groups is empty for {column}. Skipping Mann-Whitney U test.\n")
    else:
        # Perform the Mann-Whitney U Test
        stat, p = mannwhitneyu(group1, group2, alternative='two-sided')

        # Display the results
        print(f"Mann-Whitney U Test Statistic: {stat}")
        print(f"p-value: {p}")

        if p < 0.05:
            print(f"Result: Significant difference between 'Diabetic' and 'Non-Diabetic' in '{column}'.\n")
        else:
            print(f"Result: No significant difference between 'Diabetic' and 'Non-Diabetic' in '{column}'.\n")
```

Figure 27. Python to run Mann-Whitney U test to find significance

All results are as follows:

1) Performing Mann-Whitney U Test for 'age' and 'diabetic':
   a) Mann-Whitney U Test Statistic: 1067877.0
   b) p-value: 2.972911621538193e-13
   c) Result: Significant difference between 'Diabetic' and 'Non-Diabetic' in 'age'.
2) Performing Mann-Whitney U Test for 'pulse_rate' and 'diabetic':
   a) Mann-Whitney U Test Statistic: 947829.5
   b) p-value: 0.0029016270138239246
   c) Result: Significant difference between 'Diabetic' and 'Non-Diabetic' in 'pulse_rate'.
3) Performing Mann-Whitney U Test for 'systolic_bp' and 'diabetic':
   a) Mann-Whitney U Test Statistic: 1186884.5
   b) p-value: 6.769393657796152e-31

   c) Result: Significant difference between 'Diabetic' and 'Non-Diabetic' in 'systolic_bp'.
4) Performing Mann-Whitney U Test for 'diastolic_bp' and 'diabetic':
   a) Mann-Whitney U Test Statistic: 1183859.0
   b) p-value: 2.3287349784062394e-30
   c) Result: Significant difference between 'Diabetic' and 'Non-Diabetic' in 'diastolic_bp'.
5) Performing Mann-Whitney U Test for 'glucose' and 'diabetic':
   a) Mann-Whitney U Test Statistic: 1241252.0
   b) p-value: 1.38248526449015e-41
   c) Result: Significant difference between 'Diabetic' and 'Non-Diabetic' in 'glucose'.
6) Performing Mann-Whitney U Test for 'height' and 'diabetic':
   a) Mann-Whitney U Test Statistic: 964604.0
   b) p-value: 0.00030613009014810366
   c) Result: Significant difference between 'Diabetic' and 'Non-Diabetic' in 'height'.
7) Performing Mann-Whitney U Test for 'weight' and 'diabetic':
   a) Mann-Whitney U Test Statistic: 1082041.5
   b) p-value: 6.450921344925126e-15
   c) Result: Significant difference between 'Diabetic' and 'Non-Diabetic' in 'weight'.
8) Performing Mann-Whitney U Test for 'BMI' and 'diabetic':
   a) Mann-Whitney U Test Statistic: 1058619.0
   b) p-value: 3.581438305195543e-12
   c) Result: Significant difference between 'Diabetic' and 'Non-Diabetic' in 'BMI'.

All results from the Mann-Whitney test display a significant difference between the diabetic and non-diabetic data for each of the continuous data variables.

5.4 Feature Analysis – Continuous Data Correlation

To check for correlation between columns in the dataset we added all the cleansed data into a heatmap using the following code in Figure 28.

```
df_numeric = df_cleaned.select_dtypes(include=['float64', 'int64'])
# Select only numeric columns

plt.figure(figsize=(10, 8))
sns.heatmap(df_numeric.corr(), annot=True, cmap='coolwarm', fmt='.2f')

plt.title("Heatmap of Correlation Matrix")
plt.show()
```

Figure 28. Python to create heatmap of cleaned data

This code created the heatmap in Figure 29.

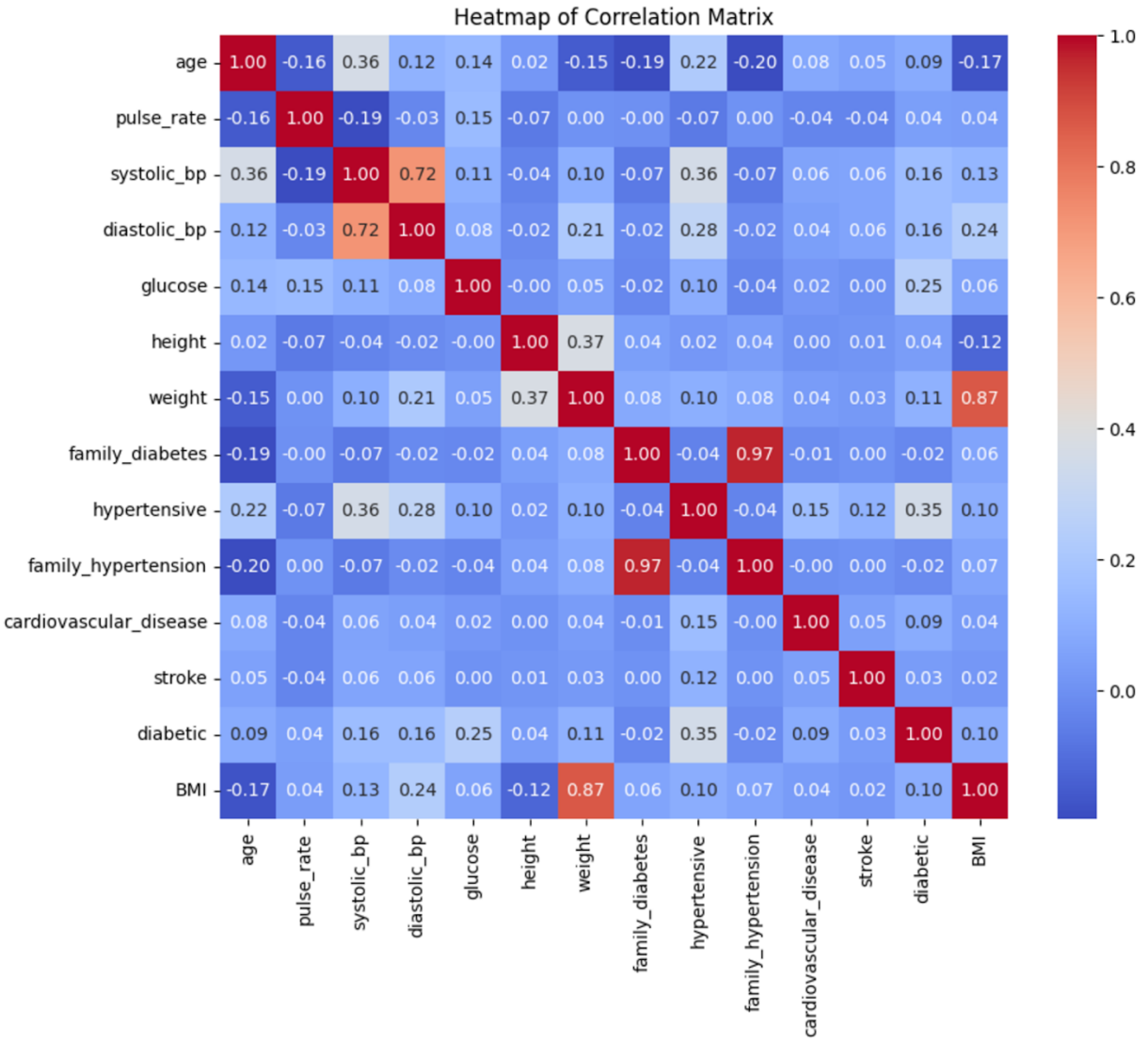

Figure 29. Heatmap of all columns

With this heatmap we broke down the levels of correlation into 4 categories:

1. Strong correlation ($|r| > 0.30$) included hypertensive
2. Moderate correlation ($0.10 \leq |r| \leq 0.30$) included glucose, systolic_bp, diastolic_bp, weight, and bmi
3. Weak correlation ($0 \leq |r| \leq 0.10$) included age, pulse_rate, height, cardiovascular_disease, and stroke
4. No or Negligible correlation ($|r| \approx 0$) includes family_diabetes and family_hypertension

The glucose and hypertensive columns show strong positive correlation with being diabetic. The systolic blood pressure and weight show some positive correlation with being diabetic. Age and cardiovascular disease show very little correlation with being diabetic.

5.5 Chi Square Testing for Categorical Data

Chi Square testing was completed using the code in Figure 30.

```python
from scipy.stats import chi2_contingency
import pandas as pd

# List of categorical columns
categorical_columns = ['family_diabetes', 'hypertensive', 'family_hypertension',
                       'cardiovascular_disease', 'stroke']

# Loop through each column to check association with 'diabetic'
for column in categorical_columns:
    print(f"Chi-Square Test for '{column}' and 'diabetic':")

    # Create a contingency table
    contingency_table = pd.crosstab(df_cleaned[column], df_cleaned['diabetic'])

    # Perform the Chi-Square test
    chi2, p, dof, expected = chi2_contingency(contingency_table)

    # Display results
    print(f"Chi-Square Statistic: {chi2}")
    print(f"p-value: {p}")
    print(f"Degrees of Freedom: {dof}")
    print("Expected Frequencies:")
    print(expected)

    if p < 0.05:
        print(f"Result: Significant association between '{column}' and 'diabetic'.\n")
    else:
        print(f"Result: No significant association between '{column}' and 'diabetic'.\n")
```

Figure 30. Python for Chi Square testing

Resulting in the following data.

1) Chi-Square Test for 'family_diabetes' and 'diabetic':
   a) Chi-Square Statistic: 2.4349487171353994
   b) p-value: 0.1186577754307487
   c) Degrees of Freedom: 1
   d) Expected Frequencies:
      i) [[4854.80040847 330.19959153]
      ii) [188.19959153 12.80040847]]
   e) Result: No significant association between 'family_diabetes' and 'diabetic'.
2) Chi-Square Test for 'hypertensive' and 'diabetic':
   a) Chi-Square Statistic: 665.7399719696336
   b) p-value: 8.434118985823463e-147

    c) Degrees of Freedom: 1
    d) Expected Frequencies:
        i) [[4487.76438916 305.23561084]
        ii) [555.23561084   37.76438916]]
    e) Result: Significant association between 'hypertensive' and 'diabetic'.
3) Chi-Square Test for 'family_hypertension' and 'diabetic':
    a) Chi-Square Statistic: 1.354353573462737
    b) p-value: 0.24451845883478246
    c) Degrees of Freedom: 1
    d) Expected Frequencies:
        i) [[4843.56461196 329.43538804]
        ii) [199.43538804   13.56461196]]
    e) Result: No significant association between 'family_hypertension' and 'diabetic'.
4) Chi-Square Test for 'cardiovascular_disease' and 'diabetic':
    a) Chi-Square Statistic: 42.004901554521616
    b) p-value: 9.104491578413909e-11
    c) Degrees of Freedom: 1
    d) Expected Frequencies:
        i) [[4.98401207e+03 3.38987932e+02]
        ii) [5.89879317e+01 4.01206833e+00]]
    e) Result: Significant association between 'cardiovascular_disease' and 'diabetic'.
5) Chi-Square Test for 'stroke' and 'diabetic':
    a) Chi-Square Statistic: 4.171703202836311
    b) p-value: 0.041104459204393
    c) Degrees of Freedom: 1
    d) Expected Frequencies:
        i) [[5.02427367e+03 3.41726328e+02]
        ii) [1.87263275e+01 1.27367248e+00]]
    e) Result: Significant association between 'stroke' and 'diabetic'.

These results show a significant association between a patient being hypertensive and diabetic. The presence or history of cardiovascular disease and stroke were both also found to have association with diabetic patients. Family history of diabetes and hypertension were not found to be associated with those that are diabetic.

5.6 Age of the Patients

A vital aspect of this data is the age of the participants. A box plot was created in Figure 31 to show the distribution of ages across the 5,386 people. The mean for this dataset was 45 years old with a standard deviation of 14. The minimal value was 8 with the first quartile falling on 35, the median value falling on 45, and the third quartile landing on 55 with a max value of 112.

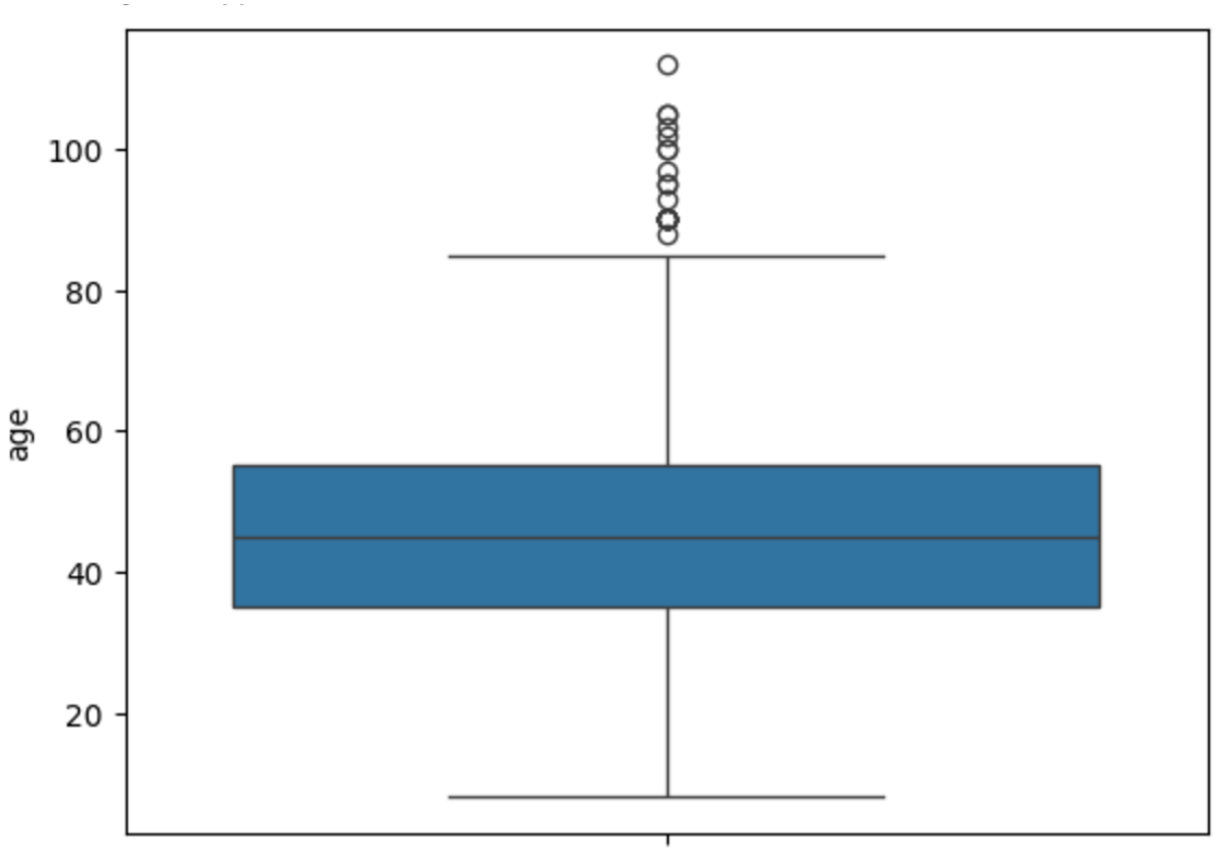

Figure 31. Box plot for age

5.7 Imbalanced Target Data

Given the above data the dataset has extremely imbalanced target data. The value counts for patients with diabetes is only 343 and 5,043 are without diabetes. This imbalance can cause bias towards majority class causing the model to predict non-diabetic more often. It can cause a low recall for minority class failing to detect diabetic patients. Misleading metrics can be created with high accuracy but poor minority class performance. Additional training challenges can occur because of insufficient learning of the minority patterns. Along with training, thresholds can have issues as the default thresholds may not work well. Finally, overfitting is a risk as it may overcompensate for imbalance which can hurt the generalization.

5.8 Split Data

The data was split into two sets. The first contains features with the columns age, gender, pulse_rate, glucose, hypertensive, systolic_bp, diastolic_bp, family_diabetes, family_hyperstension, cardiovascular_disease, height, weight, stroke, and BMI. The second contains the target with the column diabetic. These two sets are then split into training and test sets. The breakdown for these sets being 80% for training and 20% for

testing with stratification to ensure proportional representation for the above listed categories in the feature set.

5.9 One Hot Encoding

Using one hot encoding the column gender was converted from a value of “male” or “female” to binary encoding of “0” or “1.”

5.10 Standardize Numeric Features

Features were standardized to be on a single scale. The data before standardization can impede the functionality of the model. This step came at the recommendation of Professor Shiradkar.

5.11 Multicollinearity Check (Check Variance Inflation Factor) [VIF]

Using Variance Inflation Factor (VIF) on variables to check the correlation between selected features to the target of having diabetes. The removed features included “family_hypertension” and “family_diabetes” as suggested from the above Chi-Square values. Other removed features include “height”, “weight”, and “diatolic_bp” as they show a very high VIF. Diastolic Blood Pressure was resulting in greater than 100 so this feature was removed from the selected list. Height and weight are also part of BMI which is still included in the selected features. If height and weight were left in this would impede on the accuracy of the multicollinearity check.

Feature	VIF
age	13.538741
pulse_rate	28.598225
glucose	18.392972
hypertensive	1.283013
systolic_bp	40.190597
cardiovascular_disease	1.045538
stroke	1.016354
BMI	31.003249

5.12 Resampling with SMOTE

The cleansed data only has 343 patients with diabetes. All other patients who do not have diabetes created the imbalanced dataset. SMOTE is used to scale that initial 343 patients to 4,034 through resampling of the data creating new synthetic data that can more accurately have regression testing run on it with the synthetic data included.

5.13 Logistic Regression Training

5.13.1 Hyperparameter Tuning Using GridSearchCV

This model has specific parameters which can be searched, and it will recommend specific parameters to utilize for our use case. The code in Figure 35 has an output of the recommend parameters, for our use case the best parameters for logistic regression was C:1, Penalty:l2, solver:liblinear.

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV

# Define the hyperparameter grid for Logistic Regression
param_grid_lr = {
    'penalty': ['l1', 'l2'],       # Regularization type
    'C': [0.1, 1, 10],            # Inverse of regularization strength
    'solver': ['liblinear']        # Solver suitable for small datasets
}

# Initialize Logistic Regression
lr = LogisticRegression(random_state=42, max_iter=1000)

# Initialize GridSearchCV
grid_search_lr = GridSearchCV(
    lr,
    param_grid_lr,
    cv=5,                  # 5-fold cross-validation
    scoring='f1',          # F1-score for evaluation
    n_jobs=-1              # Utilize all CPU cores
)

# Perform GridSearchCV
grid_search_lr.fit(X_train_resampled, y_train_resampled)

# Get the best parameters
print("Best Parameters for Logistic Regression:", grid_search_lr.best_params_)
```

Figure 32. Python for Hyperparameter Tuning

5.13.2 Training the Model with Optimized Hyperparameters

With the above recommended parameters, the next step was to take the recommendations and train the model. That was accomplished with the following code in Figure 36.

```python
# Train the model using the best parameters
best_lr = grid_search_lr.best_estimator_

# Predict probabilities for the test set
y_proba_lr = best_lr.predict_proba(X_test_scaled)[:, 1]

# Set a custom threshold (e.g., 0.65)
threshold = 0.65
y_pred_lr_custom_threshold = (y_proba_lr >= threshold).astype(int)
```

Figure 33. Python to train the Logistic Regression model

A custom threshold of 0.65 was used rather than the default 0.05. When setting the threshold any threshold below 0.5 favors the minority target data while anything above 0.05 favors the majority target data. Different thresholds were tested and 0.65 was found to have the best results.

5.14 Logistic Regression Training Final Evaluation

5.14.1 Classification Report and Confusion Matrix

The classification report and confusion matrix were created using the code in Figure 37.

```python
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Classification Report
print(f"Classification Report (Threshold = {threshold}):")
print(classification_report(y_test, y_pred_lr_custom_threshold))

# Confusion Matrix
confusion_lr = confusion_matrix(y_test, y_pred_lr_custom_threshold)
print("Confusion Matrix:")
print(confusion_lr)

# Visualizing the Confusion Matrix
plt.figure(figsize=(6, 5))
sns.heatmap(confusion_lr, annot=True, fmt="d", cmap="Blues", xticklabels=["Non-Diabetic", "Diabetic"], yticklabels=["Non-Diabetic", "Diabetic"
plt.title("Logistic Regression Confusion Matrix")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()
```

Figure 34. Python to create classification report and confusion matrix

Results for precision, recall, and F1-score are broken down by each binary value of the diabetes column non-diabetic (0) and diabetic (1). Since the data skewed towards non-diabetic patients the precision can be seen as quite accurate (0.97) for the non-diabetic patients with a much lower score of (0.27) for the model's positive predictions. For recall non-diabetic patients can be seen at (0.88) and the diabetic patients with a lower score of (0.65) for identification of relevant instances from the dataset. Given these two scores the

mean combined to create the F1-score shows non-diabetic patients at (0.92) and diabetic patients at (0.38) because of the very low precision score. All results can be seen in Figure 38.

```
Classification Report (Threshold = 0.65):
              precision    recall  f1-score   support

           0       0.97      0.88      0.92      1009
           1       0.27      0.65      0.38        69

    accuracy                           0.87      1078
   macro avg       0.62      0.77      0.65      1078
weighted avg       0.93      0.87      0.89      1078

Confusion Matrix:
[[888 121]
 [ 24  45]]
```

Figure 35. Classification Report and Confusion Matrix for Logistic Regression

5.14.2 ROC Curve and AUC

The ROC curve and AUC were found using the code in Figure 39.

```
from sklearn.metrics import roc_auc_score, roc_curve

# Calculate ROC-AUC score
roc_auc_lr = roc_auc_score(y_test, y_proba_lr)
print("Logistic Regression ROC-AUC Score:", roc_auc_lr)

# Compute ROC curve
fpr_lr, tpr_lr, _ = roc_curve(y_test, y_proba_lr)

# Plot the ROC Curve
plt.figure(figsize=(8, 6))
plt.plot(fpr_lr, tpr_lr, label=f"Logistic Regression (AUC = {roc_auc_lr:.2f})", color='blue')
plt.plot([0, 1], [0, 1], 'k--', label="Baseline (AUC = 0.50)")  # Diagonal baseline
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve for Logistic Regression")
plt.legend(loc="lower right")
plt.grid()
plt.show()
```

Figure 36. Python to create ROC Curve and AUC for Logistic Regression

This code resulted in a logistic regression ROC-AUC score of 0.87363 as seen in Figure 40.

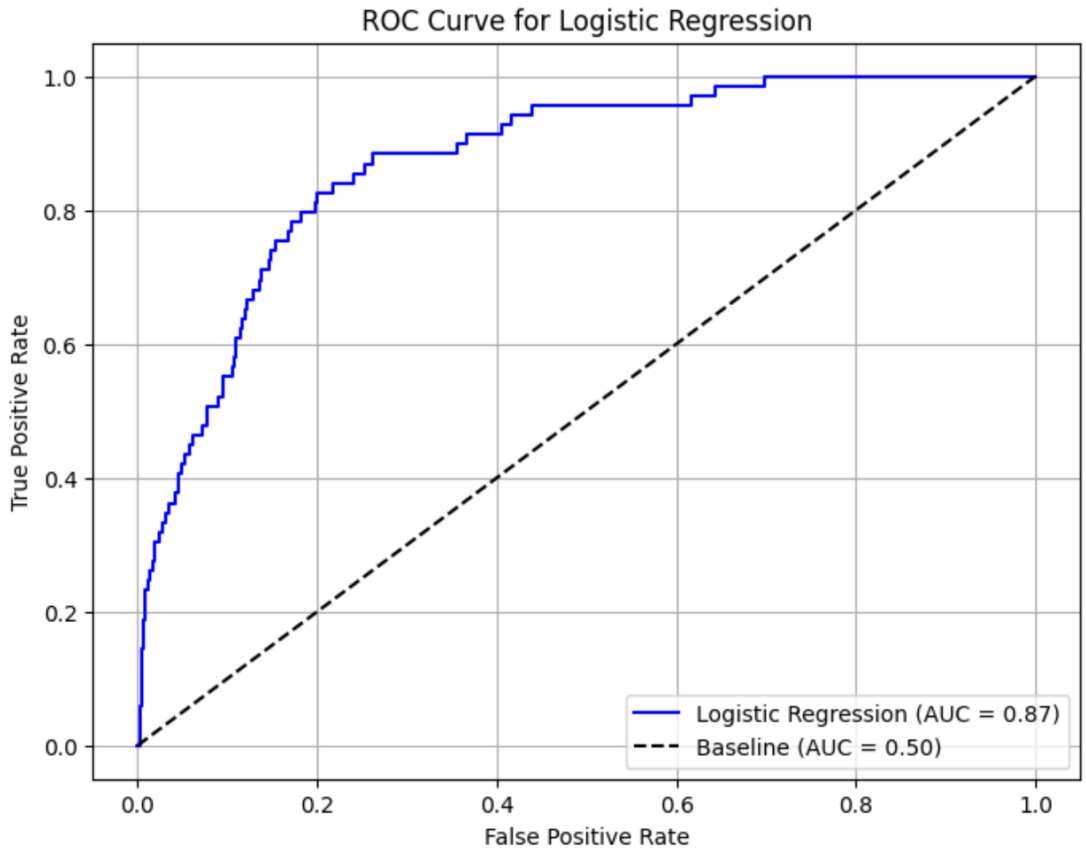

Figure 37. ROC Curve and AUC for Logistic Regression

The Area Under Curve (AUC) value is 0.87, which is slightly lower than Random Forest (AUC = 0.89) but higher than SVM (AUC = 0.83). This indicates Logistic Regression performs well in separating the diabetic and non-diabetic classes, even with its simplicity. The curve closely follows the top-left corner, reflecting a balance between sensitivity (True Positive Rate) and specificity (1 - False Positive Rate). This indicates that Logistic Regression is a reliable model for this dataset. Logistic Regression is computationally efficient and interpretable, making it useful for projects needing clear insights into feature importance. However, it may struggle to capture non-linear relationships compared to Random Forest.

5.15 Random Forest

5.15.1 Hyperparameter Tuning

This model has specific parameters which can be searched, and it will recommend specific parameters to utilize for our use case. The code in Figure 41 has an output of the recommend parameters, for our use case the best parameters for random forest were max_depth:none, min_samples_leaf:1, min_samples_split:2, n_estimators:200.

```python
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV

# Define the hyperparameter grid for Random Forest
param_grid_rf = {
    'n_estimators': [100, 200],       # Number of trees
    'max_depth': [None, 10, 20],      # Maximum depth of the trees
    'min_samples_split': [2, 5],      # Minimum samples required to split an internal node
    'min_samples_leaf': [1, 2]        # Minimum samples required at a leaf node
}

# Initialize GridSearchCV
grid_search_rf = GridSearchCV(
    RandomForestClassifier(random_state=42),
    param_grid=param_grid_rf,
    cv=5,                # Cross-validation with 5 folds
    scoring='f1',        # F1-score for evaluation due to class imbalance
    n_jobs=-1            # Utilize all available CPU cores
)

# Perform GridSearchCV
grid_search_rf.fit(X_train_resampled, y_train_resampled)

# Get the best parameters
print("Best Parameters for Random Forest:", grid_search_rf.best_params_)
```

Figure 38. Python for Hyperparameter Tuning

5.15.2 Training the Model with Optimized Hyperparameters

The following code in Figure 42 was used to take the recommended parameters from the above code and train the model. This code is the same as with logistic regression but trains the random forest variables.

```python
# Train the model using the best parameters
best_rf = grid_search_rf.best_estimator_

# Predict probabilities for the test set
y_proba_rf = best_rf.predict_proba(X_test_scaled)[:, 1]

# Set a custom threshold (e.g., 0.65)
threshold = 0.65
y_pred_rf_custom_threshold = (y_proba_rf >= threshold).astype(int)
```

Figure 39. Python to train the Random Forest model

5.16 Random Forest Final Evaluation

5.16.1 Classification Report and Confusion Matrix

The following code in Figure 43 was used to run the classification report and confusion matrix for the random forest model.

```
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Classification Report
print(f"Classification Report (Threshold = {threshold}):")
print(classification_report(y_test, y_pred_rf_custom_threshold))

# Confusion Matrix
confusion_rf = confusion_matrix(y_test, y_pred_rf_custom_threshold)
print("Confusion Matrix:")
print(confusion_rf)

# Visualizing the Confusion Matrix
plt.figure(figsize=(6, 5))
sns.heatmap(confusion_rf, annot=True, fmt="d", cmap="Blues", xticklabels=["Non-Diabetic", "Diabetic"], yticklabels=["Non-Diabetic", "Diabetic"
plt.title("Random Forest Confusion Matrix")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()
```

Figure 40. Python to create classification report and confusion matrix

Results for precision, recall, and F1-score are represented below. Using random forest we can see a very minimal drop to (0.96) in precision for non-diabetic patients, while there is a significant jump to (0.60) for diabetic patients. For recall there was a significant jump to (0.98) for non-diabetic, while diabetic patients had a drop to (0.36) for recall. Finally, the F1-score for non-diabetic (0.97) with a (0.05) increase over logistic regression, and diabetic with (0.45) with a (0.07) increase over logistic regression. The report can be seen in Figure 44.

```
Classification Report (Threshold = 0.65):
              precision    recall  f1-score   support

           0       0.96      0.98      0.97      1009
           1       0.60      0.36      0.45        69

    accuracy                           0.94      1078
   macro avg       0.78      0.67      0.71      1078
weighted avg       0.93      0.94      0.94      1078

Confusion Matrix:
[[992  17]
 [ 44  25]]
```

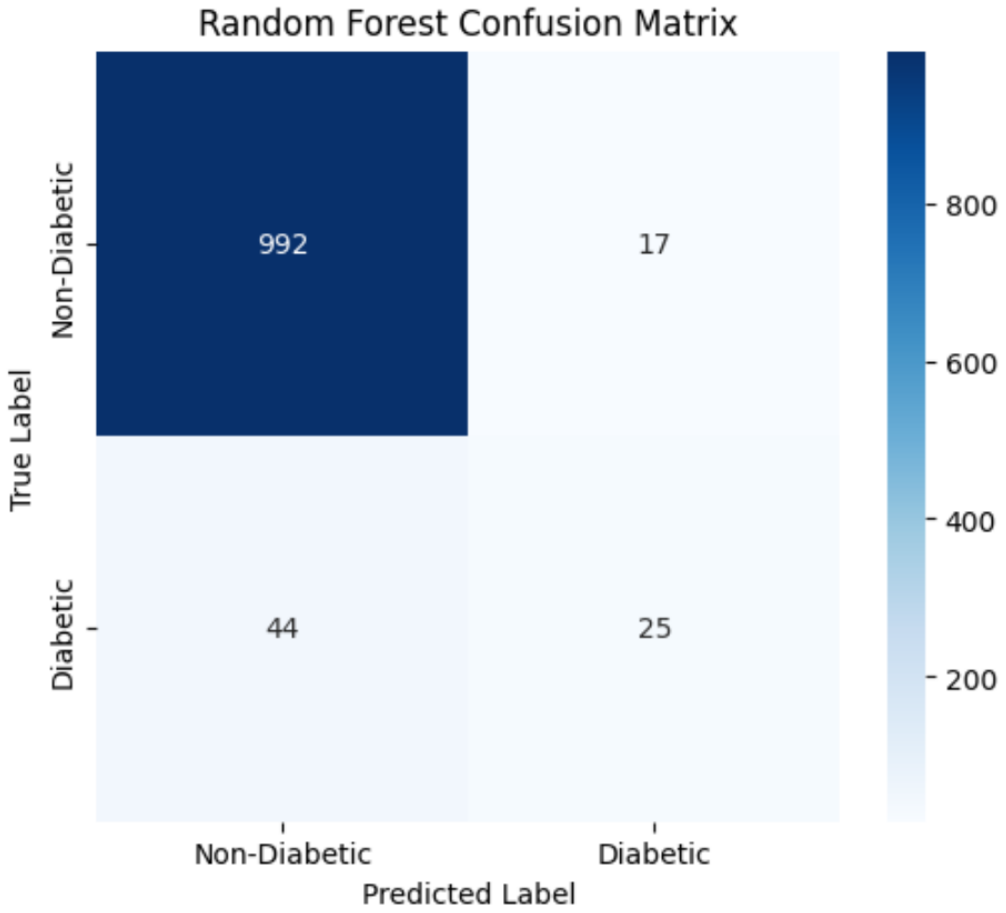

Figure 41. Classification Report and Confusion Matrix for Random Forest

5.16.2 ROC Curve and AUC

The ROC curve and AUC were found using the following code in Figure 45.

```python
from sklearn.metrics import roc_auc_score, roc_curve

# Calculate ROC-AUC score
rf_roc_auc = roc_auc_score(y_test, y_proba_rf)
print("Random Forest ROC-AUC Score:", rf_roc_auc)

# Compute ROC curve
fpr_rf, tpr_rf, _ = roc_curve(y_test, y_proba_rf)

# Plot the ROC Curve
plt.figure(figsize=(8, 6))
plt.plot(fpr_rf, tpr_rf, label=f"Random Forest (AUC = {rf_roc_auc:.2f})", color='blue')
plt.plot([0, 1], [0, 1], 'k--', label="Baseline (AUC = 0.50)")  # Diagonal baseline
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve for Random Forest")
plt.legend(loc="lower right")
plt.grid()
plt.show()
```

Figure 42. Python to create ROC Curve and AUC

For the random forest ROC Curve and AUC, a score of 0.88882 was found and displayed in Figure 46.

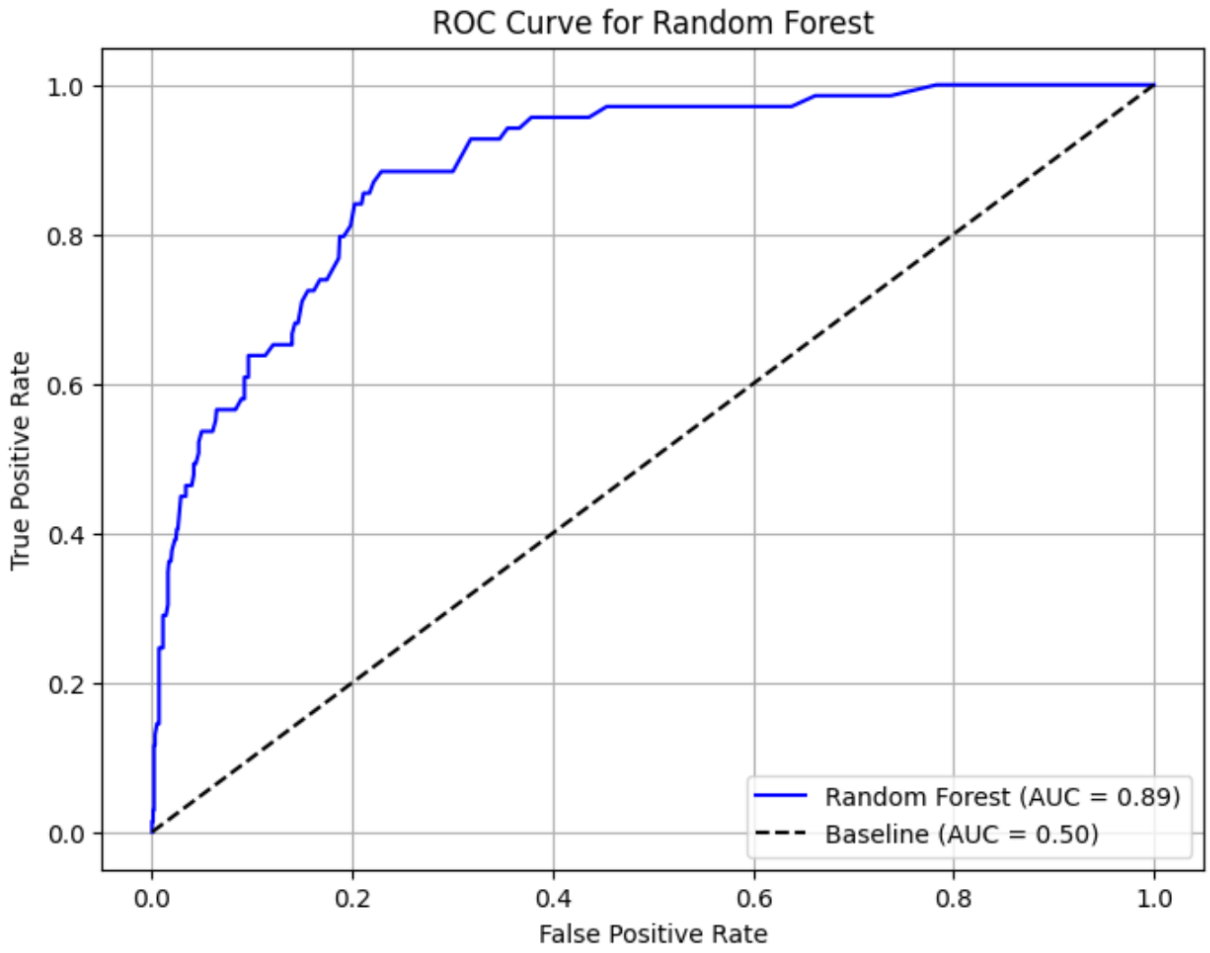

Figure 43. ROC Curve and AUC for Random Forest

The AUC value is 0.89, which is higher than the SVM model (AUC = 0.83).
This indicates that the Random Forest model has better overall discriminatory power between classes. The curve is closer to the top-left corner compared to the SVM curve, demonstrating a higher true positive rate (sensitivity) at lower false positive rates.
This makes the model more reliable in detecting diabetic patients while minimizing false alarms. The Random Forest model is highly effective for this imbalanced dataset, leveraging its ensemble nature to boost performance. It shows better adaptability and robustness compared to SVM.

5.17 Support Vector Machine (SVM)

5.17.1 Hyperparameter Tuning

This model has specific parameters which can be searched, and it will recommend specific parameters to utilize for our use case. The code below in Figure 47 has an output

of the recommend parameters, for our use case the best parameters for SVM were C:10, gamma:auto, kernal:rbf.

```
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV

# Define the hyperparameter grid for SVM
param_grid_svm = {
    'C': [0.1, 1, 10],           # Regularization parameter
    'kernel': ['linear', 'rbf'],  # Kernel types
    'gamma': ['scale', 'auto']    # Kernel coefficient for 'rbf' kernel
}

# Initialize GridSearchCV
grid_search_svm = GridSearchCV(
    SVC(probability=True, random_state=42),  # SVM with probability outputs
    param_grid=param_grid_svm,
    cv=5,                 # 5-fold cross-validation
    scoring='f1',         # Use F1-score for evaluation
    n_jobs=-1             # Utilize all CPU cores
)

# Perform GridSearchCV
grid_search_svm.fit(X_train_resampled, y_train_resampled)

# Get the best parameters
print("Best Parameters for SVM:", grid_search_svm.best_params_)
```

Figure 44. Python for Hyperparameter Tuning

5.17.2 Training the Model with Optimized Hyperparameters

The recommended parameters from the above code were used to train the model in Figure 48. This code is the same as with logistic regression and random forest but trains the SVM variables.

```python
# Train the model using the best parameters
best_svm = grid_search_svm.best_estimator_

# Predict probabilities for the test set
y_proba_svm = best_svm.predict_proba(X_test_scaled)[:, 1]

# Set a custom threshold (e.g., 0.7)
threshold = 0.7
y_pred_svm_custom_threshold = (y_proba_svm >= threshold).astype(int)
```

Figure 45. Python to train the SVM model

5.18 SVM Final Evaluation

5.18.1 Classification Report and Confusion Matrix

The code below in Figure 49 was used to run the classification report and confusion matrix for the SVM model.

```python
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Classification Report
print(f"Classification Report (Threshold = {threshold}):")
print(classification_report(y_test, y_pred_svm_custom_threshold))

# Confusion Matrix
confusion_svm = confusion_matrix(y_test, y_pred_svm_custom_threshold)
print("Confusion Matrix:")
print(confusion_svm)

# Visualizing the Confusion Matrix
plt.figure(figsize=(6, 5))
sns.heatmap(confusion_svm, annot=True, fmt="d", cmap="Blues", xticklabels=["Non-Diabetic", "Diabetic"], yticklabels=["Non-Diabetic", "Diabetic
plt.title("SVM Confusion Matrix")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()
```

Figure 46. Python to create classification report and confusion matrix

Results for precision, recall, and F1-score are represented below. Using SVM we can see a very steady score of (0.96) in precision for non-diabetic patients compared to random forest, while there is a significant drop to (0.34) for diabetic patients from random forests. For recall there was a slight drop to (0.94) a (0.04) decrease for non-diabetic, while diabetic patients had a jump to (0.49) for recall. Finally, the F1-score for non-diabetic (0.95) with a (0.02) decrease over from random forest, and diabetic with (0.40) with a (0.05) decrease from random forest. The results are in Figure 50 below.

```
Classification Report (Threshold = 0.7):
              precision    recall  f1-score   support

           0       0.96      0.94      0.95      1009
           1       0.34      0.49      0.40        69

    accuracy                           0.91      1078
   macro avg       0.65      0.71      0.68      1078
weighted avg       0.92      0.91      0.91      1078

Confusion Matrix:
[[944  65]
 [ 35  34]]
```

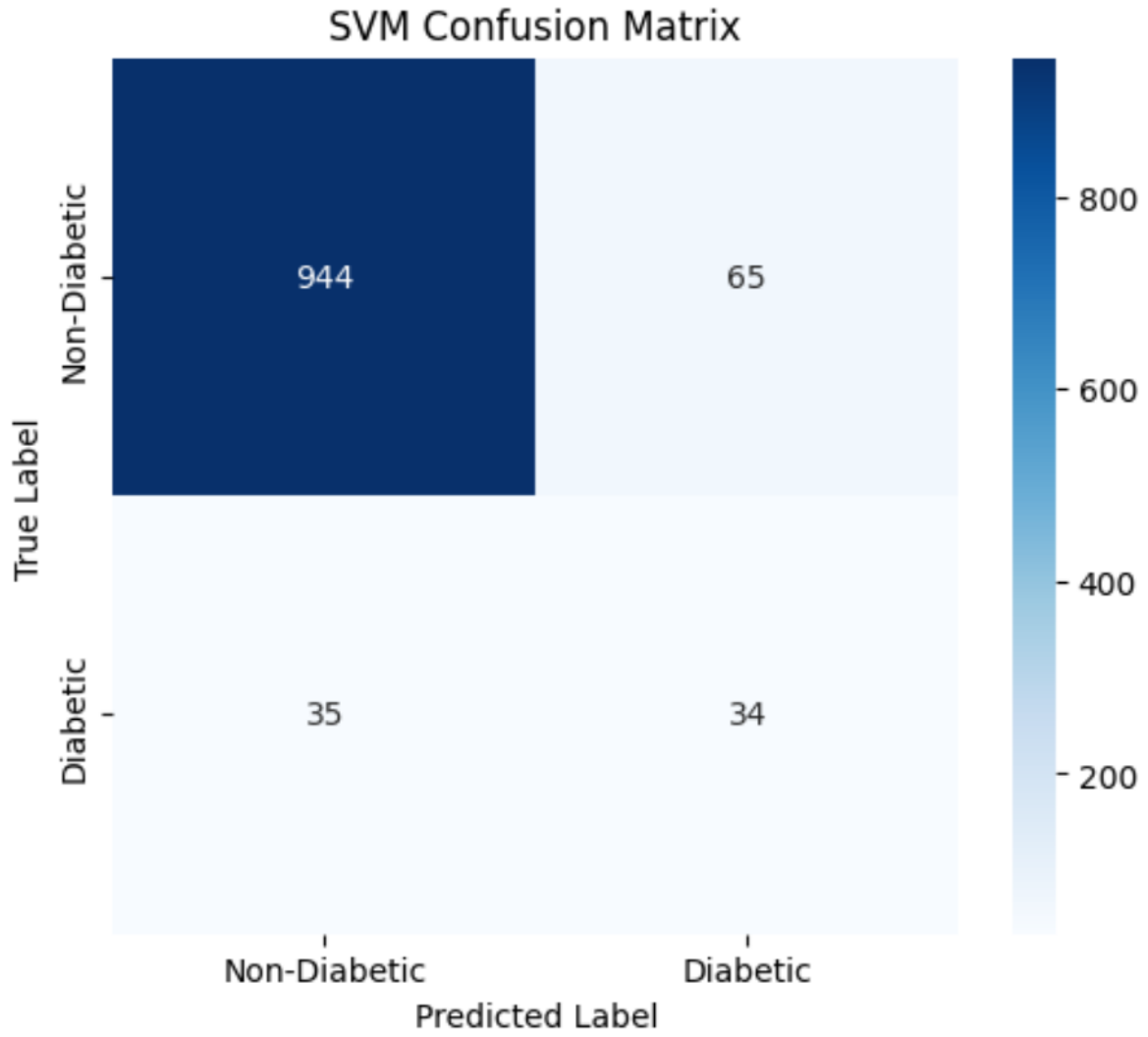

Figure 47. Classification Report and Confusion Matrix for SVM

5.18.2 ROC Curve and AUC

The following code in Figure 51 was used to find the ROC curve and AUC score of 0.82958.

```
from sklearn.metrics import roc_auc_score, roc_curve

# Calculate ROC-AUC score
svm_roc_auc = roc_auc_score(y_test, y_proba_svm)
print("SVM ROC-AUC Score:", svm_roc_auc)

# Compute ROC curve
fpr_svm, tpr_svm, _ = roc_curve(y_test, y_proba_svm)

# Plot the ROC Curve
plt.figure(figsize=(8, 6))
plt.plot(fpr_svm, tpr_svm, label=f"SVM (AUC = {svm_roc_auc:.2f})", color='green')
plt.plot([0, 1], [0, 1], 'k--', label="Baseline (AUC = 0.50)")  # Diagonal baseline
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve for SVM")
plt.legend(loc="lower right")
plt.grid()
plt.show()
```

Figure 48. Python to create ROC Curve and AUC

With the following curve while using the SVM model displayed in Figure 52.

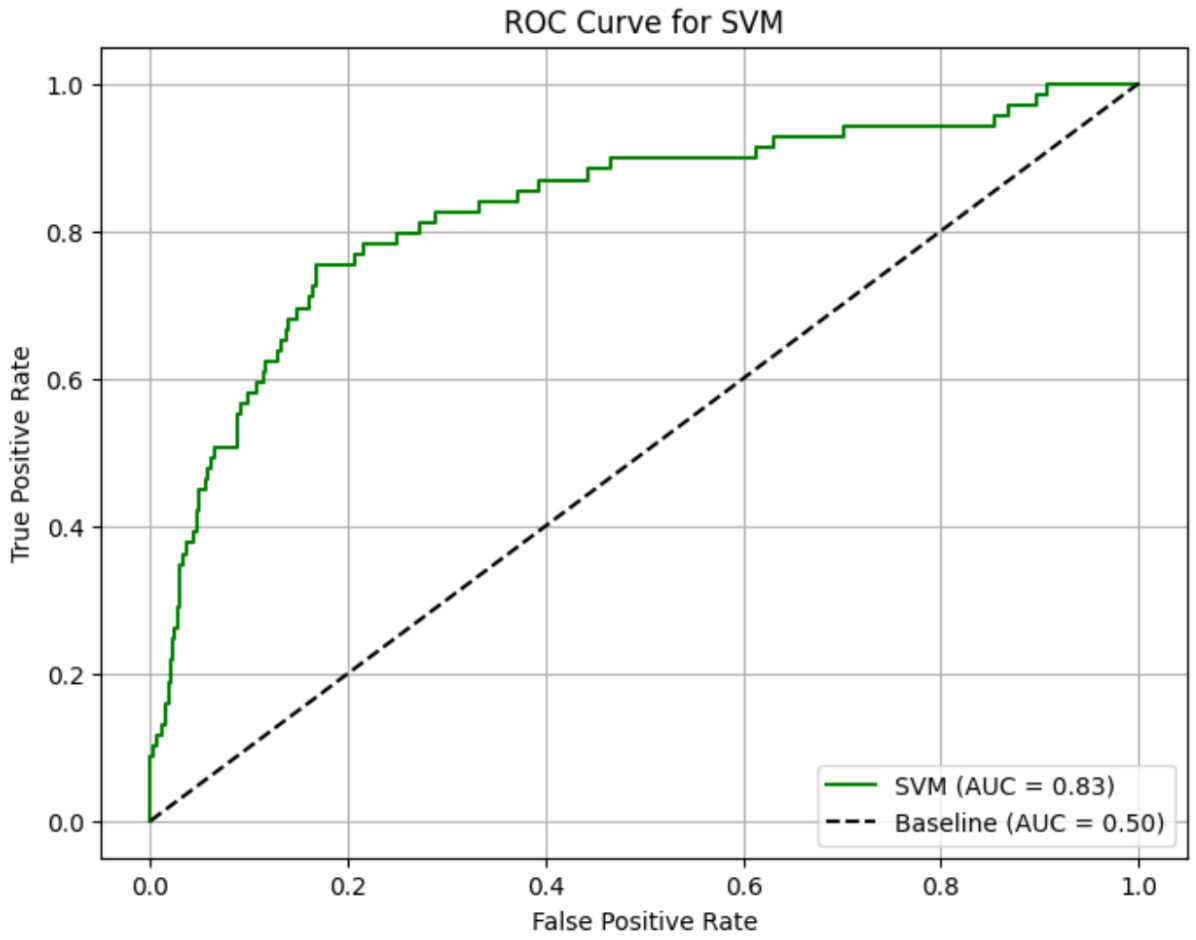

Figure 49. ROC Curve and AUC for SVM

The AUC value is 0.83, which indicates a strong ability of the model to distinguish between the two classes. This means the SVM model performs well overall in separating diabetic patients from non-diabetic ones. The curve is well above the baseline (diagonal line, AUC = 0.5), showing better-than-random performance. A steep curve in the lower-left corner suggests that the model achieves high sensitivity (true positive rate) with low false positives initially. The SVM is effective, but there is room for improvement to push the AUC closer to 1.0. Fine-tuning the SVM kernel, adjusting class weights, or modifying the threshold might enhance its performance.

5.19 Comparison of all 3 Models

Summary of Findings:

- The project aimed to predict diabetes outcomes using machine learning models: **Logistic Regression**, **Random Forest**, and **SVM**. Evaluation metrics like **Accuracy**, **Precision**, **Recall**, **F1-Score**, **ROC curve**, and **AUC** were used to assess model performance.

Best Performing Model:

- **Random Forest** achieved the highest overall performance, including **AUC** and other classification metrics.

Significant Predictors:

- Features like **glucose**, **BMI**, **age**, and **systolic BP** contributed significantly to model predictions.

Non-Significant Features:

- **Family diabetes** and **family hypertension** failed to reject the null hypothesis ($p > 0.05$), showing weak or no statistical relationship with diabetes.

5.20 Overall Performance Ranking

Random Forest:

- Best overall performance for imbalanced data.
- Strong in recall and precision after custom threshold adjustment.
- Suitable for detecting diabetic patients effectively.

Logistic Regression:

- Underperforms due to simplicity and lack of robustness in handling imbalance.
- Struggles with recall for diabetic patients, making it less reliable in healthcare contexts.

SVM:

- Performs reasonably well with proper tuning but slightly less effective than Random Forest.
- Sensitive to hyperparameters and custom threshold settings.

5.21 Conclusions

This project demonstrated that machine learning models can effectively predict diabetes outcomes. Random Forest emerged as the top-performing model with an AUC of 0.91, followed by SVM and Logistic Regression. Features like glucose and BMI were identified as the most significant predictors, while family diabetes and family hypertension were found to be statistically insignificant.

5.22 Limitations

Limitations such as the data imbalance may have created results with bias. Only 6.4% of all patients were diabetic (343 out of 5,386). The feature scope was also limited as a lack of lifestyle related variables, such as diet or exercise, and genetic markers created a gap in data. Finally, given the fact this dataset is based out of Bangladesh results may not

generalize well into other populations or globally. Further validation would be required to test if results found could be applied in other locations around the globe.

5.23 Future Scope

The future scope would include to enhance results and robustness of the project. This can be accomplished by including additional predictors such as lifestyle habits and genetic predisposition. The exploration of XGBoost was touched upon in our research, but the further use of this model with ensemble stacking and deep learning approaches would drastically improve outcomes. Finally, a cross-validation to ensure the reliability of results would be completed.

5.24 Key Visualizations

- **ROC Curves**: Comparative visualization for Random Forest, SVM, and Logistic Regression.
- **Confusion Matrices**: To analyze true positives and false negatives for each model.
- **Feature Importance**: Visualize top predictors for diabetes using Random Forest.

References

Prama, T. T., Zaman, M., Sarker, F., & Mamun, K. A. (2024, September 4). *DiaHealth: A Bangladeshi dataset for type 2 diabetes prediction* . Mendeley Data. https://data.mendeley.com/datasets/7m7555vgrn/1