Final Project Report

Topic: Predicting 10-Year Risk of Coronary Heart Disease using K-Nearest Neighbors

INT 7623 Data Science for Business

Prof. Dr. Areej Salaymeh

Lawrence Technological University

Tej Ketan Acharya

Vikalp Vinod Chaubey

May 02, 2025

Table of Contents

Dataset Characteristics

The dataset used is the Framingham Heart Study dataset, which contains 4,240 records of patient data with 16 variables (15 features and 1 target). Each record represents a participant, and the variables capture various health metrics and risk factors. The target variable *TenYearCHD* indicates whether the patient experienced CHD within 10 years. Below is the data dictionary listing all columns and their descriptions exactly as provided:

1. **Gender:** Numeric (binary), Gender of the patient (1 = Male, 0 = Female)
2. **age:** Numeric, Description: Age of the patient in years
3. **education:** Numeric (continuous), Description: Education level (1 = some high school, 2 = high school graduate, 3 = some college, 4 = college graduate, 5 = postgraduate)
4. **currentSmoker:** Numeric (binary), Description: Smoking status (1 = current smoker, 0 = non-smoker)
5. **cigsPerDay:** Numeric, Description: Number of cigarettes smoked per day
6. **BPMeds:** Numeric (binary), Description: Use of blood pressure medications (1 = yes, 0 = no)
7. **prevalentStroke:** Numeric (binary), Description: History of stroke (1 = yes, 0 = no)
8. **prevalentHyp:** Numeric (binary), Description: History of hypertension (1 = yes, 0 = no)
9. **diabetes:** Numeric (binary), Description: Diabetes status (1 = yes, 0 = no)
10. **totChol:** Numeric, Description: Total serum cholesterol level (mg/dl)
11. **sysBP:** Numeric, Description: Systolic blood pressure (mm Hg)
12. **diaBP:** Numeric, Description: Diastolic blood pressure (mm Hg)
13. **BMI:** Numeric, Description: Body Mass Index (weight in kg / height in m^2)
14. **heartRate:** Numeric, Description: Heart rate (beats per minute)
15. **glucose:** Numeric, Description: Glucose level (mg/dl)
16. **TenYearCHD:** Numeric (binary), Description: Presence of heart disease within the next 10 years (1 = heart disease, 0 = no heart disease)

Dataset summary: The first 15 attributes above will serve as features for prediction, and TenYearCHD is the target outcome. The dataset is suitable for a binary classification task using KNN. Prior to modeling, we will need to explore the data distribution, handle any missing values or outliers, and consider the class imbalance (since the occurrence of CHD is less common than non-occurrence in this dataset).

Introduction

Coronary Heart Disease (CHD) is a leading cause of mortality, and early risk prediction is crucial for preventative care. In this project, we tackle the problem of predicting whether a patient will develop

CHD within the next 10 years. We use data from the Framingham Heart Study, a longitudinal dataset of cardiovascular risk factors, to build a predictive model. The task is framed as a supervised binary classification (0 = no CHD, 1 = CHD) and we employ the K-Nearest Neighbors (KNN) algorithm to perform the prediction. KNN is a straightforward yet powerful classification technique that assigns a class based on the majority vote of the closest training examples in feature space. The goal is to construct a complete machine learning pipeline – from exploratory data analysis through model evaluation – and identify the best KNN model for predicting 10-year CHD risk.

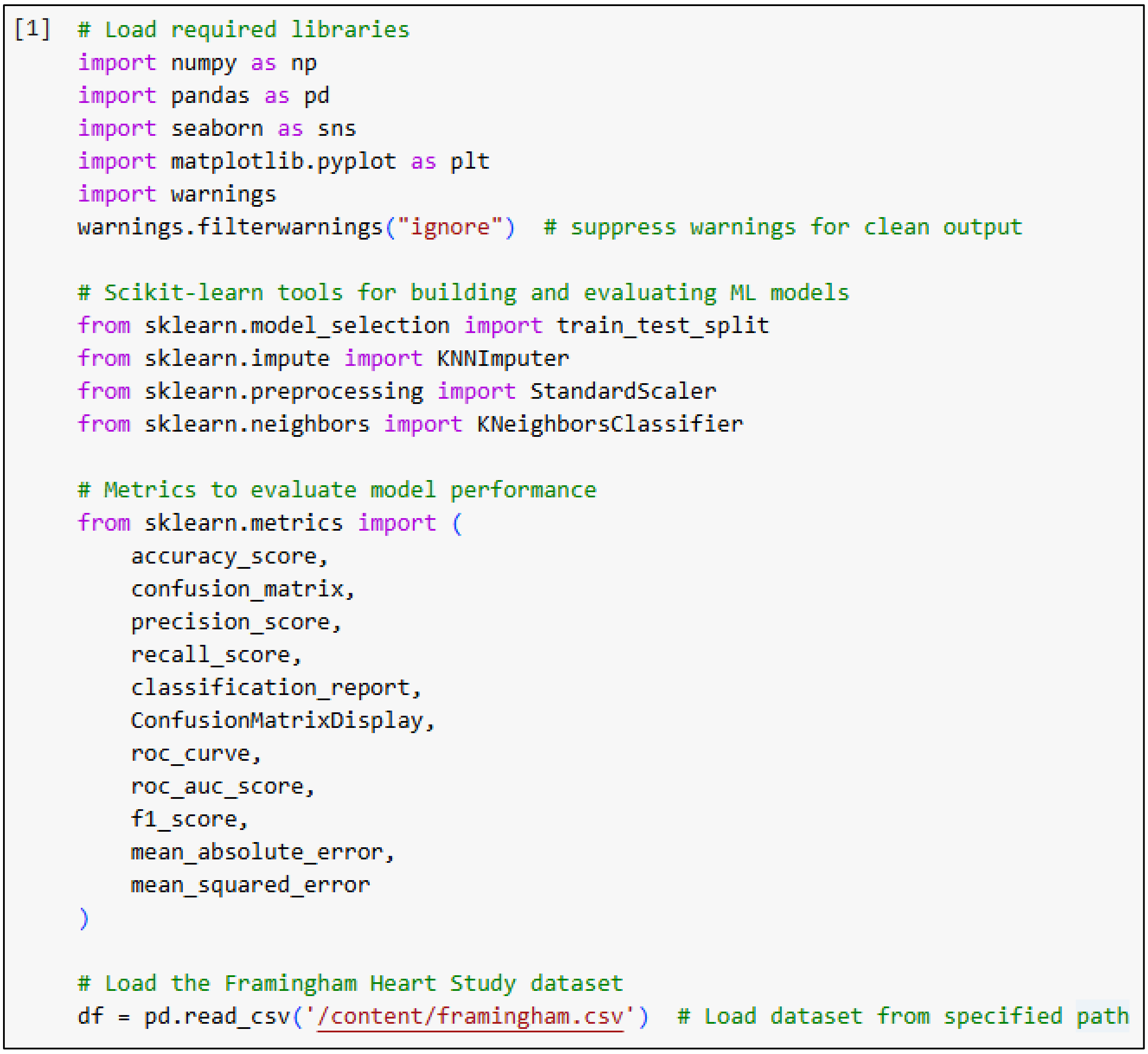
Problem Statement: Predict a patient's 10-year risk of CHD using their medical and lifestyle features from the Framingham Heart Study.

Methodology: Supervised learning with a binary classification task. We implement a KNN classifier, exploring different values of *K* (number of neighbors) to balance model complexity and performance.

In the following sections, we present a step-by-step report of the analysis. We start with loading all the necessary libraries and the farmingham dataset itself, followed by exploratory data analysis (EDA) to understand the dataset characteristics and key risk factor trends. Next, we describe data preprocessing steps such as handling missing values and class imbalance. We then detail the partitioning of data into training and testing sets and the selection of candidate *K* values. After training multiple KNN models, we evaluate their performance on the test set using metrics including accuracy, confusion matrices, ROC curves, and recall. Finally, we discuss the best-performing model and suggest potential improvements. Throughout the report, Code Cell screenshots show the implementation, accompanied by explanations of the code and interpretations of the outputs and figures.

Part 1: Data Loading and Library Import

Step 1: Importing libraries and loading the dataset: The analysis was conducted in Python using common data science libraries. We first imported necessary libraries: pandas for data manipulation, NumPy for numerical operations, matplotlib/seaborn for visualization, and scikit-learn for machine learning algorithms. After importing libraries, the dataset was loaded from the CSV file using pd.read_csv('framingham.csv'). This command reads the CSV file into a pandas DataFrame.

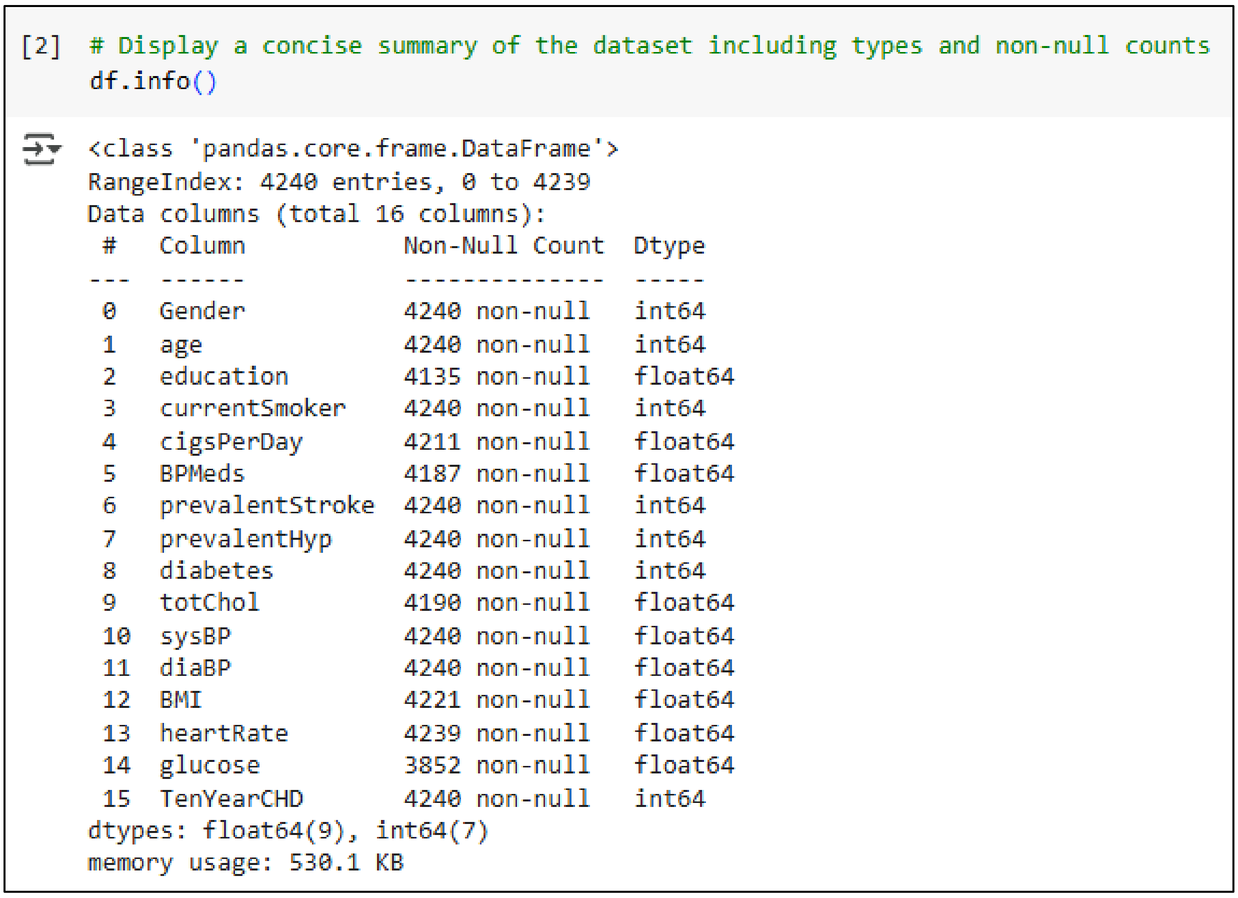


```
[1] # Load required libraries
    import numpy as np
    import pandas as pd
    import seaborn as sns
    import matplotlib.pyplot as plt
    import warnings
    warnings.filterwarnings("ignore")  # suppress warnings for clean output

    # Scikit-learn tools for building and evaluating ML models
    from sklearn.model_selection import train_test_split
    from sklearn.impute import KNNImputer
    from sklearn.preprocessing import StandardScaler
    from sklearn.neighbors import KNeighborsClassifier

    # Metrics to evaluate model performance
    from sklearn.metrics import (
        accuracy_score,
        confusion_matrix,
        precision_score,
        recall_score,
        classification_report,
        ConfusionMatrixDisplay,
        roc_curve,
        roc_auc_score,
        f1_score,
        mean_absolute_error,
        mean_squared_error
    )

    # Load the Framingham Heart Study dataset
    df = pd.read_csv('/content/framingham.csv')  # Load dataset from specified path
```

Observations: No issues were encountered during loading, and the dataset is now ready for inspection in the DataFrame format.

Part 2: Exploratory Data Analysis (EDA) - Textual Exploration

Step 1: Examining dataset info and dimensions: We used df.info() to get an overview of the DataFrame's structure.

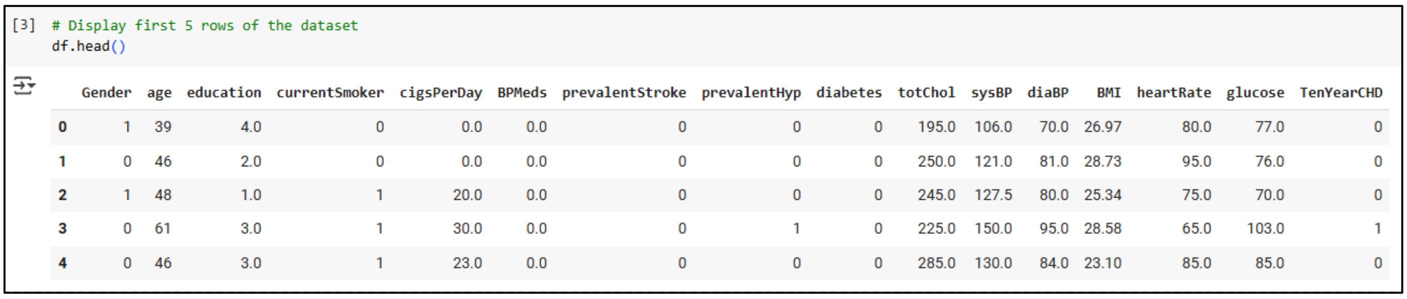


```
[2] # Display a concise summary of the dataset including types and non-null counts
    df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4240 entries, 0 to 4239
Data columns (total 16 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   Gender           4240 non-null   int64  
 1   age              4240 non-null   int64  
 2   education        4135 non-null   float64
 3   currentSmoker    4240 non-null   int64  
 4   cigsPerDay       4211 non-null   float64
 5   BPMeds           4187 non-null   float64
 6   prevalentStroke  4240 non-null   int64  
 7   prevalentHyp     4240 non-null   int64  
 8   diabetes         4240 non-null   int64  
 9   totChol          4190 non-null   float64
 10  sysBP            4240 non-null   float64
 11  diaBP            4240 non-null   float64
 12  BMI              4221 non-null   float64
 13  heartRate        4239 non-null   float64
 14  glucose          3852 non-null   float64
 15  TenYearCHD       4240 non-null   int64  
dtypes: float64(9), int64(7)
memory usage: 530.1 KB
```

Observations: The info output indicated that there are 4240 entries (rows) and 16 columns. It also listed each column's data type and the count of non-null values. From this, we observed that most columns are of type int64 or float64, appropriate for numerical analysis. Importantly, the info summary revealed that some columns have fewer non-null entries than 4240, indicating the presence of missing values in certain columns. For instance, education has 4135 non-null entries (out of 4240), suggesting some missing values. Similarly, slight deficits in counts for cigsPerDay, BPMeds, totChol, BMI, heartRate, and a more substantial number of missing entries in glucose (only 3852 non-null) were noted. This step establishes the size of the dataset and alerts us to the need for missing data handling at a later stage. The shape of the data (4240, 16) confirms the dataset dimensions.

Step 2: Previewing the dataset structure: We then used df.head() to display the first five rows of the DataFrame. This initial preview allows us to verify that the data has been read correctly and to observe the format of each column.

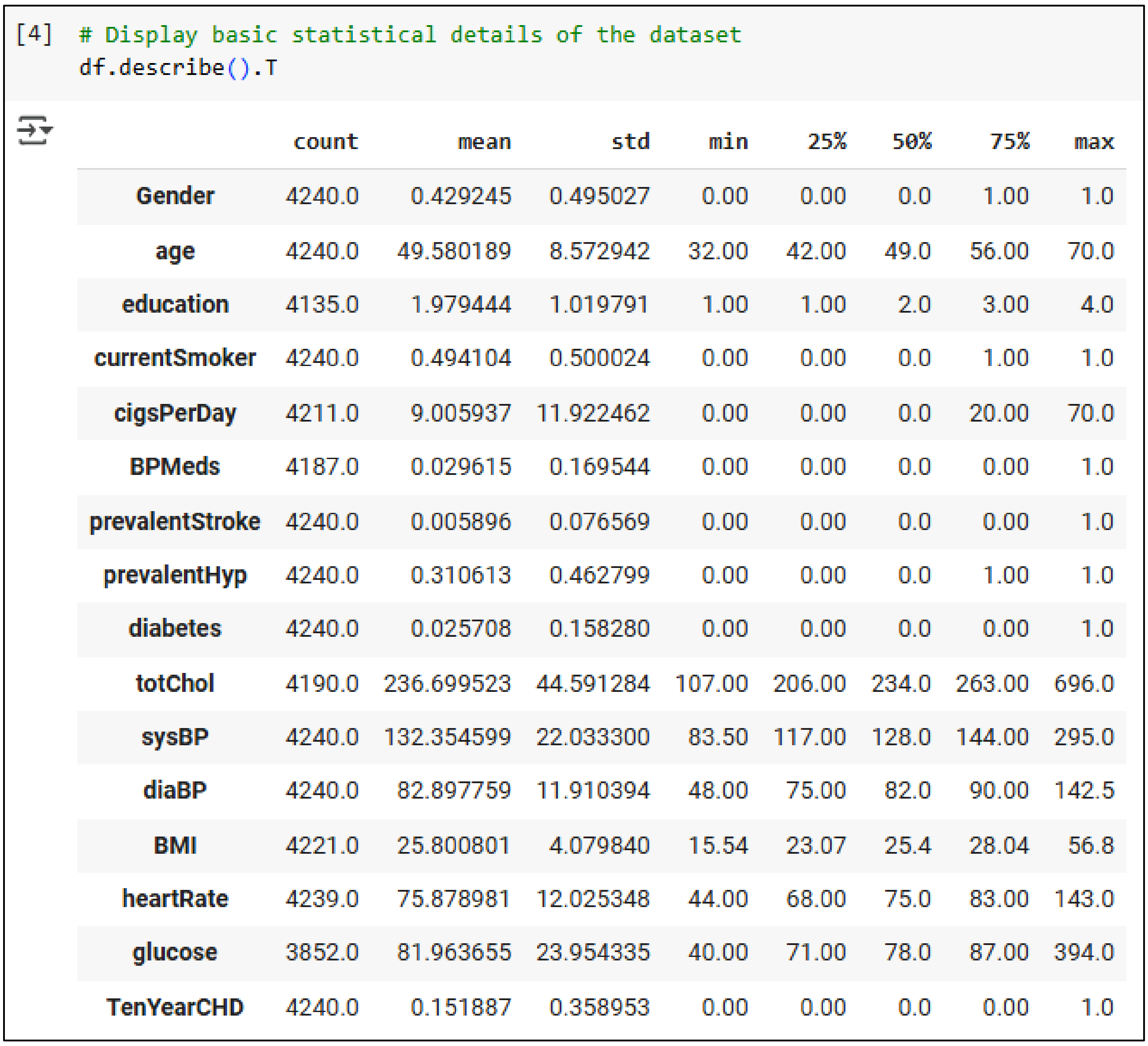


```
[3] # Display first 5 rows of the dataset
    df.head()
```

	Gender	age	education	currentSmoker	cigsPerDay	BPMeds	prevalentStroke	prevalentHyp	diabetes	totChol	sysBP	diaBP	BMI	heartRate	glucose	TenYearCHD
0	1	39	4.0	0	0.0	0.0	0	0	0	195.0	106.0	70.0	26.97	80.0	77.0	0
1	0	46	2.0	0	0.0	0.0	0	0	0	250.0	121.0	81.0	28.73	95.0	76.0	0
2	1	48	1.0	1	20.0	0.0	0	0	0	245.0	127.5	80.0	25.34	75.0	70.0	0
3	0	61	3.0	1	30.0	0.0	0	1	0	225.0	150.0	95.0	28.58	65.0	103.0	1
4	0	46	3.0	1	23.0	0.0	0	0	0	285.0	130.0	84.0	23.10	85.0	85.0	0

Observations: The output showed columns such as Gender, age, education, currentSmoker, etc., along with sample values for the first five participants. For example, we could see a row representing a 39-year-old female (Gender = 0, age = 39) with certain values for cholesterol, blood pressure, etc. The column names and values match the expected data dictionary. The data appears to be loaded properly, with no obvious formatting issues.

Step 3: Descriptive statistics for numerical features: Using df.describe(), we generated summary statistics for all numeric columns. This provided the count, mean, standard deviation, minimum, and quartile values for each feature.



```
[4] # Display basic statistical details of the dataset
    df.describe().T
```

	count	mean	std	min	25%	50%	75%	max
Gender	4240.0	0.429245	0.495027	0.00	0.00	0.0	1.00	1.0
age	4240.0	49.580189	8.572942	32.00	42.00	49.0	56.00	70.0
education	4135.0	1.979444	1.019791	1.00	1.00	2.0	3.00	4.0
currentSmoker	4240.0	0.494104	0.500024	0.00	0.00	0.0	1.00	1.0
cigsPerDay	4211.0	9.005937	11.922462	0.00	0.00	0.0	20.00	70.0
BPMeds	4187.0	0.029615	0.169544	0.00	0.00	0.0	0.00	1.0
prevalentStroke	4240.0	0.005896	0.076569	0.00	0.00	0.0	0.00	1.0
prevalentHyp	4240.0	0.310613	0.462799	0.00	0.00	0.0	1.00	1.0
diabetes	4240.0	0.025708	0.158280	0.00	0.00	0.0	0.00	1.0
totChol	4190.0	236.699523	44.591284	107.00	206.00	234.0	263.00	696.0
sysBP	4240.0	132.354599	22.033300	83.50	117.00	128.0	144.00	295.0
diaBP	4240.0	82.897759	11.910394	48.00	75.00	82.0	90.00	142.5
BMI	4221.0	25.800801	4.079840	15.54	23.07	25.4	28.04	56.8
heartRate	4239.0	75.878981	12.025348	44.00	68.00	75.0	83.00	143.0
glucose	3852.0	81.963655	23.954335	40.00	71.00	78.0	87.00	394.0
TenYearCHD	4240.0	0.151887	0.358953	0.00	0.00	0.0	0.00	1.0

Observations:

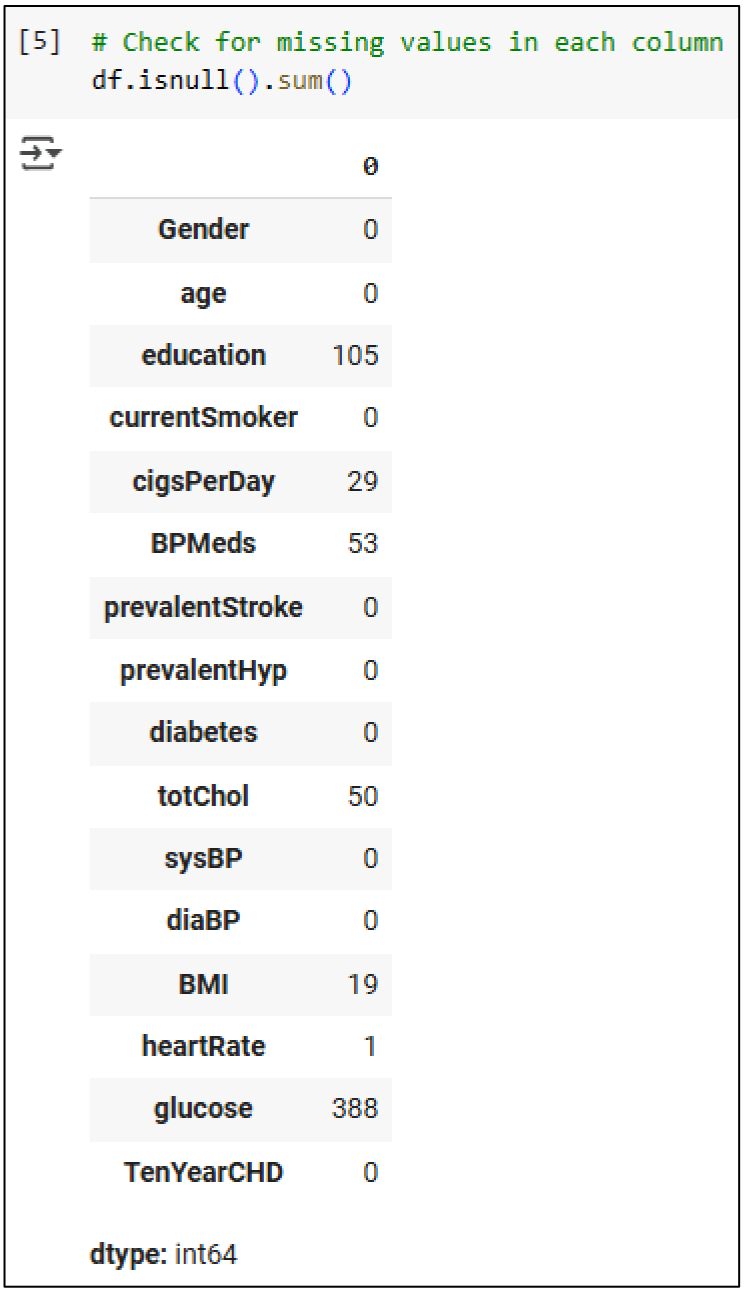
From the descriptive statistics, we observed several key characteristics of the dataset:

- The average age of participants is approximately 49.6 years, with a standard deviation of about 8.6 years. The age range spans from 32 to 70 years, indicating the cohort is adult individuals, including middle-aged to older adults.

- The mean systolic blood pressure (sysBP) is about 132.35 mm Hg (std ~22), and diastolic (diaBP) about 82.9 mm Hg (std ~11.9). The blood pressure values range from as low as 83/48 to as high as 295/142, suggesting there are individuals with very high blood pressure (potential outliers at the high end).
- The average total cholesterol (totChol) is ~236.7 mg/dL (std ~44.3). Notably, the maximum cholesterol value is 696 mg/dL, which is extremely high and may indicate a potential outlier or data entry error (as typical cholesterol levels rarely reach that value).
- Participants have an average Body Mass Index (BMI) of ~25.8, with a range from 15.5 up to 56.8. The upper range indicates some participants are in the morbidly obese category (BMI > 50), again highlighting possible outliers in weight-related metrics.
- The glucose level ranges from 40 to 394 mg/dL, with a mean of ~81.6. The maximum of 394 mg/dL is very high, indicating at least one participant had severe hyperglycemia (likely diabetic).
- For binary indicators: about 42.9% of the records are male (Gender mean ~0.429, where Male=1) implying ~57.1% female; ~49.4% are current smokers; ~31% have prevalent hypertension; ~2.6% have diabetes, etc. The target variable TenYearCHD has a mean of 0.1519, indicating 15.19% of the participants had a CHD event within 10 years, whereas 84.81% did not. This confirms the class imbalance (approximately 1 positive case to 5.6 negative cases).

These descriptive stats give a broad understanding of the dataset's distribution. They also alert us to the presence of some extreme values (potential outliers) and the imbalance in the target class, which will be important considerations for modeling.

Step 4: Identify missing data in each column: Using isnull() we searched for any null values that are present in the dataset in each column and used sum() to aggregate which provided use with the total null values that are present in each column.



```
[5] # Check for missing values in each column
    df.isnull().sum()
```

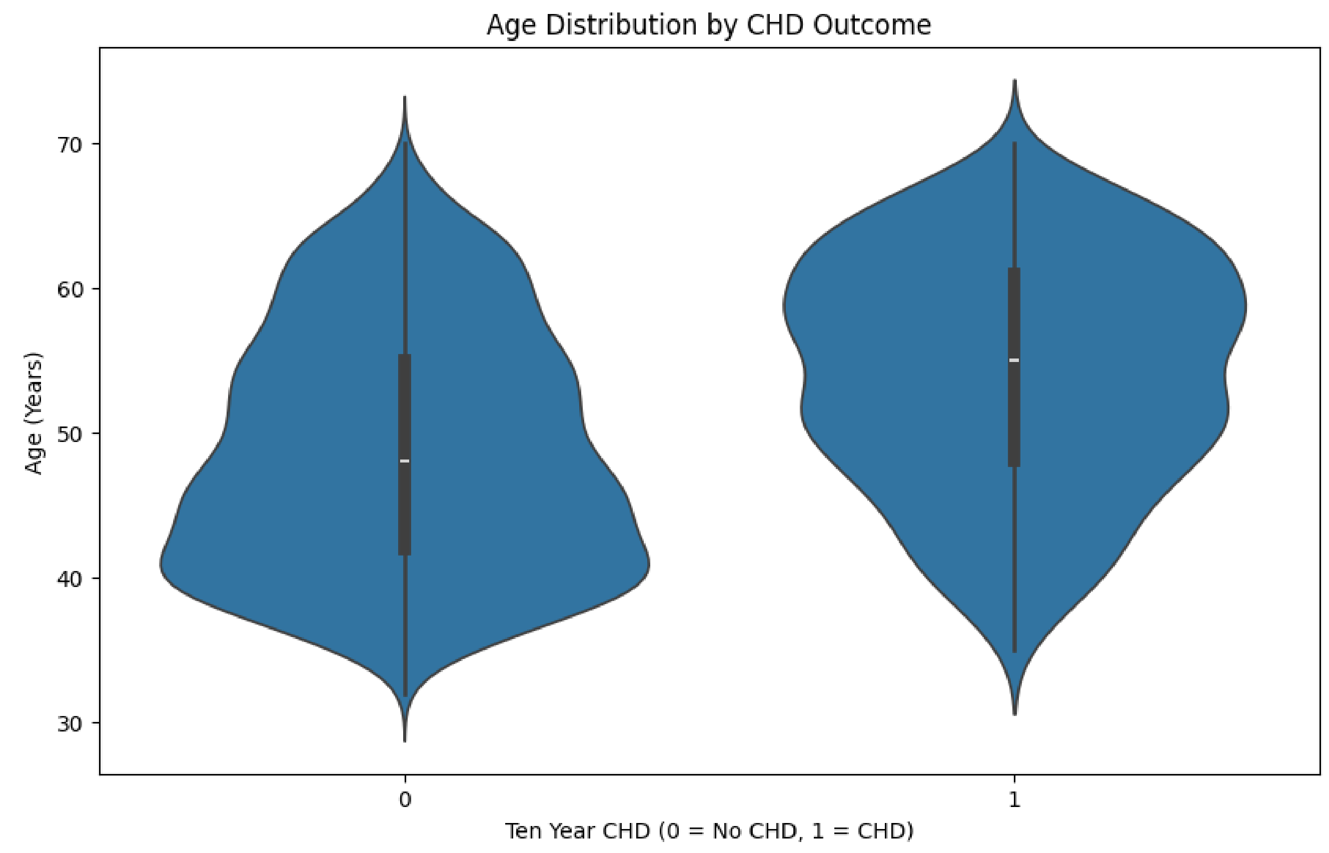
```
                    0
        Gender      0
           age      0
     education    105
  currentSmoker     0
     cigsPerDay    29
        BPMeds     53
prevalentStroke     0
   prevalentHyp     0
       diabetes     0
        totChol    50
          sysBP     0
          diaBP     0
            BMI    19
      heartRate     1
        glucose   388
     TenYearCHD     0

dtype: int64
```

Observations: The output indicated several features with missing values: for example, education had 105 missing entries, cigsPerDay had 29 missing, BPMeds 53 missing, totChol 50 missing, BMI 19 missing, heartRate 1 missing, and glucose had the most with 388 missing entries. These missing values needed to be addressed during preprocessing (discussed in the next section). Aside from missingness, the data types were mostly numeric (float64 or int64), which facilitates direct analysis after imputation.

Part 3: Exploratory Data Analysis (EDA) - Visual Exploration

Step 1: Violin plot of age by CHD outcome: Age is a well-known risk factor for heart disease, so we explored how age is distributed among those who did and did not develop CHD. To visualize the age distribution in relation to CHD diagnosis, we used a violin plot, which combines a boxplot with a KDE distribution on each side, showing both summary statistics and data density.

```
[6]  # Age distribution grouped by CHD outcome
     plt.figure(figsize=(10, 6))
     sns.violinplot(x='TenYearCHD', y='age', data=df)
     plt.title('Age Distribution by CHD Outcome')
     plt.xlabel('Ten Year CHD (0 = No CHD, 1 = CHD)')
     plt.ylabel('Age (Years)')
     plt.show()
```



Observations:

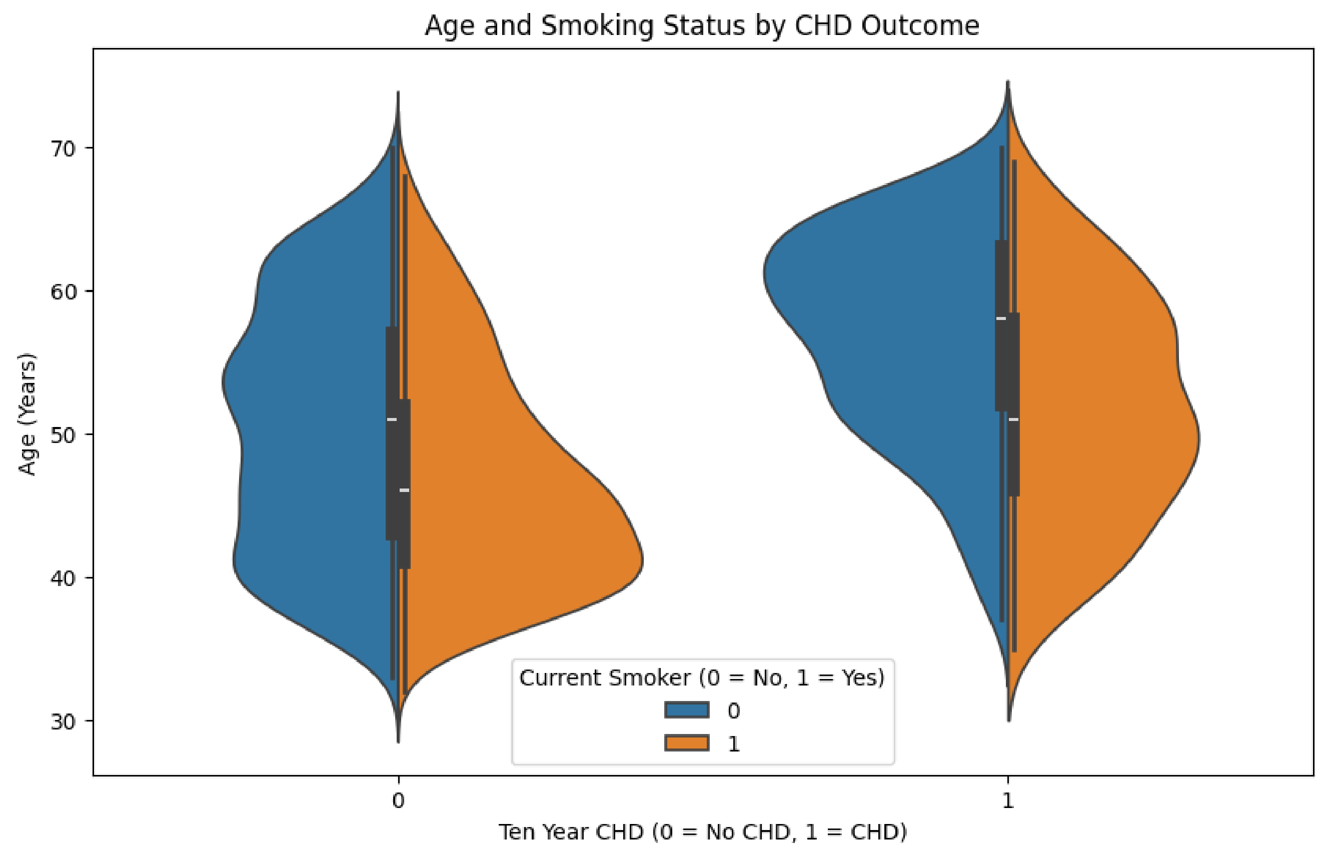
In the above figure, the left violin (labeled 0) represents individuals who did not develop CHD, and the right violin (labeled 1) corresponds to those who did:

- The median age of CHD-positive individuals is clearly higher than that of CHD-negative ones, reinforcing the established link between age and CHD.
- The spread of the CHD-positive group is also slightly narrower, indicating that most CHD cases are concentrated in the 50 to 65 age range.
- In contrast, the CHD-negative group includes a wider spread of ages, with a notable concentration in the 40–50 range.

This visualization reinforces earlier findings: age is one of the strongest individual predictors of CHD. The violin plot also reveals the density of age values, giving a deeper visual cue about concentration ranges.

<u>Step 2: Violin plot of age and smoking status by CHD outcome:</u> We extended the violin plot analysis by splitting the age distribution by both smoking status and CHD outcome. This gives a more nuanced view of how smoking and age together interact with CHD risk.

```
[7]  # Age distribution grouped by CHD and colored by smoking status
     plt.figure(figsize=(10, 6))
     sns.violinplot(x='TenYearCHD', y='age', data=df, hue='currentSmoker', split=True)
     plt.title('Age and Smoking Status by CHD Outcome')
     plt.xlabel('Ten Year CHD (0 = No CHD, 1 = CHD)')
     plt.ylabel('Age (Years)')
     plt.legend(title='Current Smoker (0 = No, 1 = Yes)')
     plt.show()
```



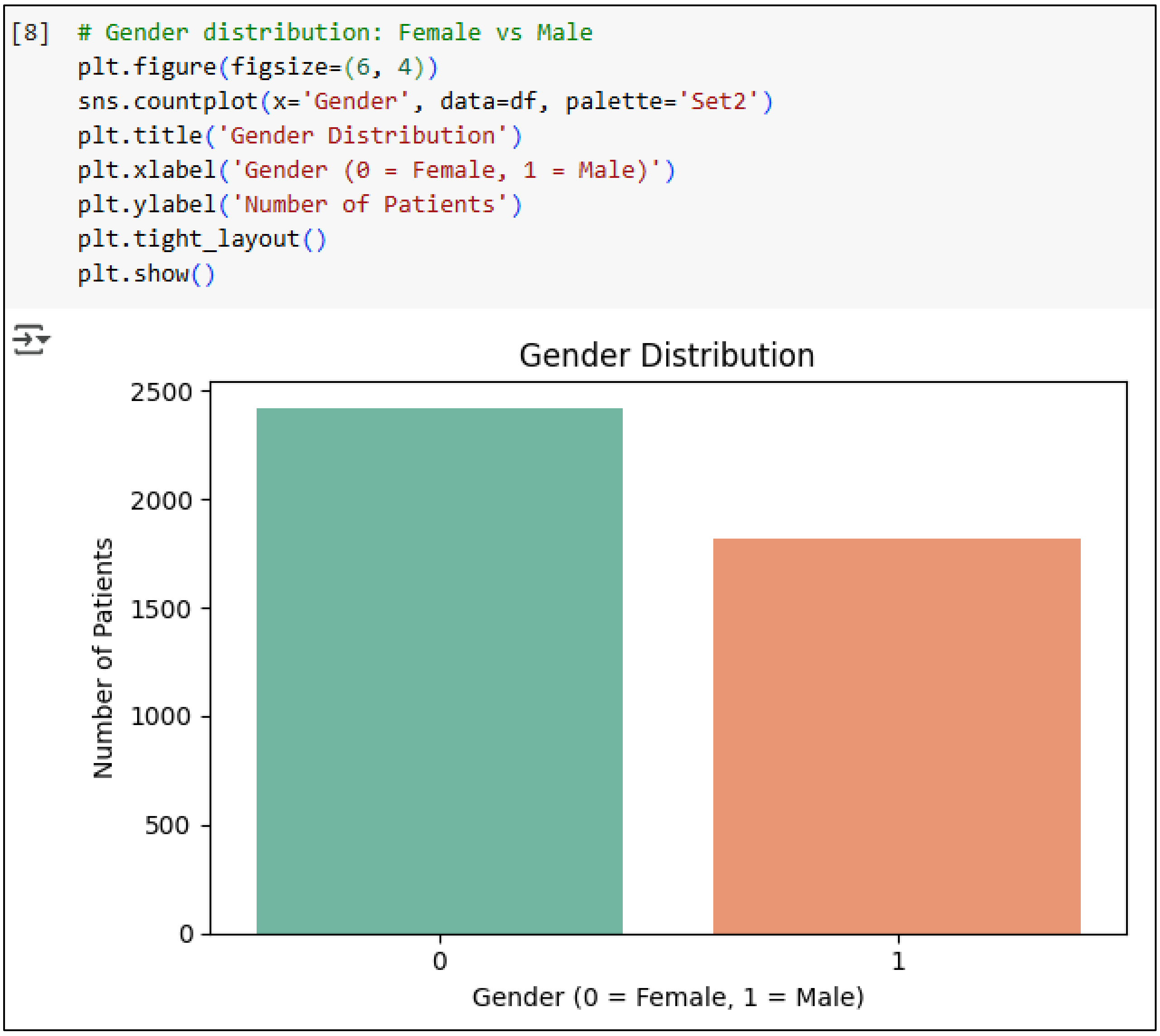
<u>Observations:</u>

In the Figure above, each side of the violins (orange = smoker, blue = non-smoker) shows how age varies for smokers and non-smokers within both CHD outcome categories:

- Among those who did not develop CHD, the age distribution appears similar across smoking status, though non-smokers have slightly older median ages.
- Among those who did develop CHD, a more visible concentration of smokers appears in the lower-to-mid age range (40s to early 50s), while non-smokers skew older.

This indicates that while age remains a dominant factor, smoking status modifies the age-related risk: CHD-positive smokers tend to be slightly younger than their non-smoking counterparts.

<u>Step 3: Gender distribution in the dataset:</u> We analyzed the **gender composition** of the dataset using a simple bar chart to understand sample balance.



```
[8]  # Gender distribution: Female vs Male
     plt.figure(figsize=(6, 4))
     sns.countplot(x='Gender', data=df, palette='Set2')
     plt.title('Gender Distribution')
     plt.xlabel('Gender (0 = Female, 1 = Male)')
     plt.ylabel('Number of Patients')
     plt.tight_layout()
     plt.show()
```

<u>Observations:</u>

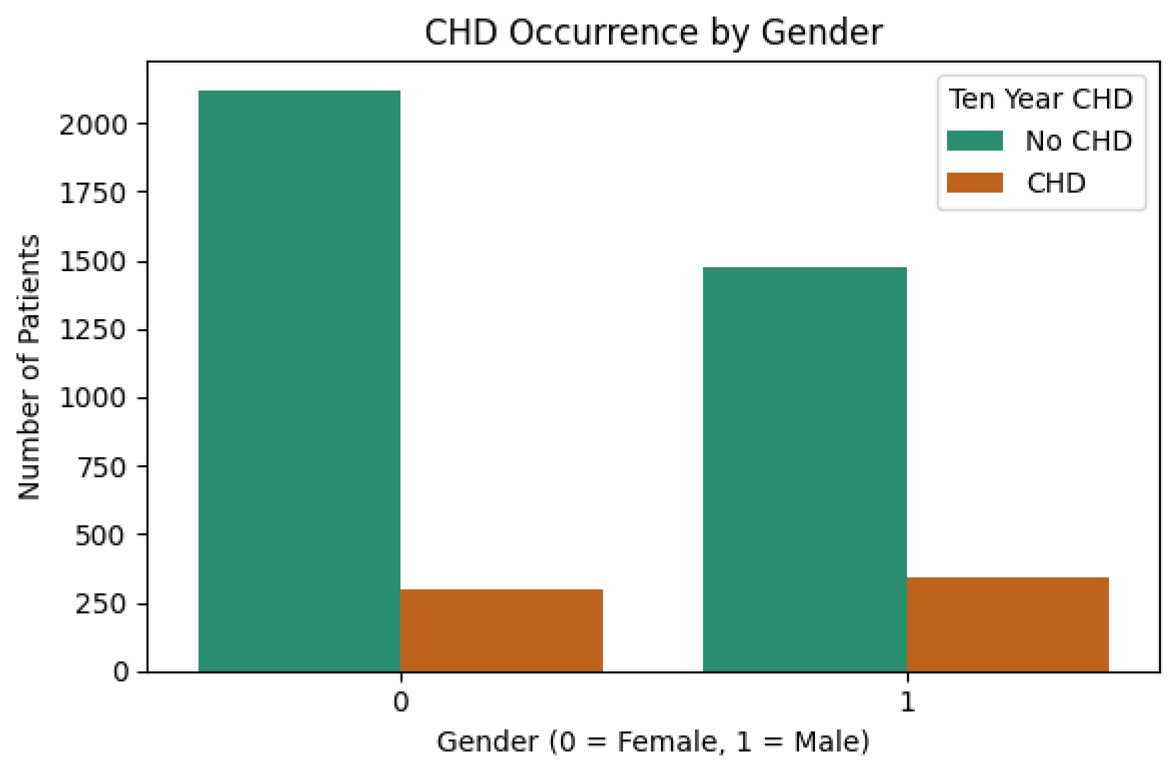
In the above figure, the bars represent counts of patients grouped by gender:

- Gender 0 (Female) includes over 2,400 patients.
- Gender 1 (Male) includes fewer than 2,000 patients.

This highlights a slight imbalance, with females representing a larger portion of the sample. This matters when evaluating gender-based outcomes, since raw CHD counts need to be interpreted proportionally.

<u>Step 4: CHD occurrence by gender:</u> To explore how CHD prevalence differs by gender, we plotted a stacked bar chart comparing CHD-positive and CHD-negative cases within each gender.

```
[9] # CHD distribution across gender categories
    plt.figure(figsize=(6, 4))
    sns.countplot(x='Gender', hue='TenYearCHD', data=df, palette='Dark2')
    plt.title('CHD Occurrence by Gender')
    plt.xlabel('Gender (0 = Female, 1 = Male)')
    plt.ylabel('Number of Patients')
    plt.legend(title='Ten Year CHD', labels=['No CHD', 'CHD'])
    plt.tight_layout()
    plt.show()
```



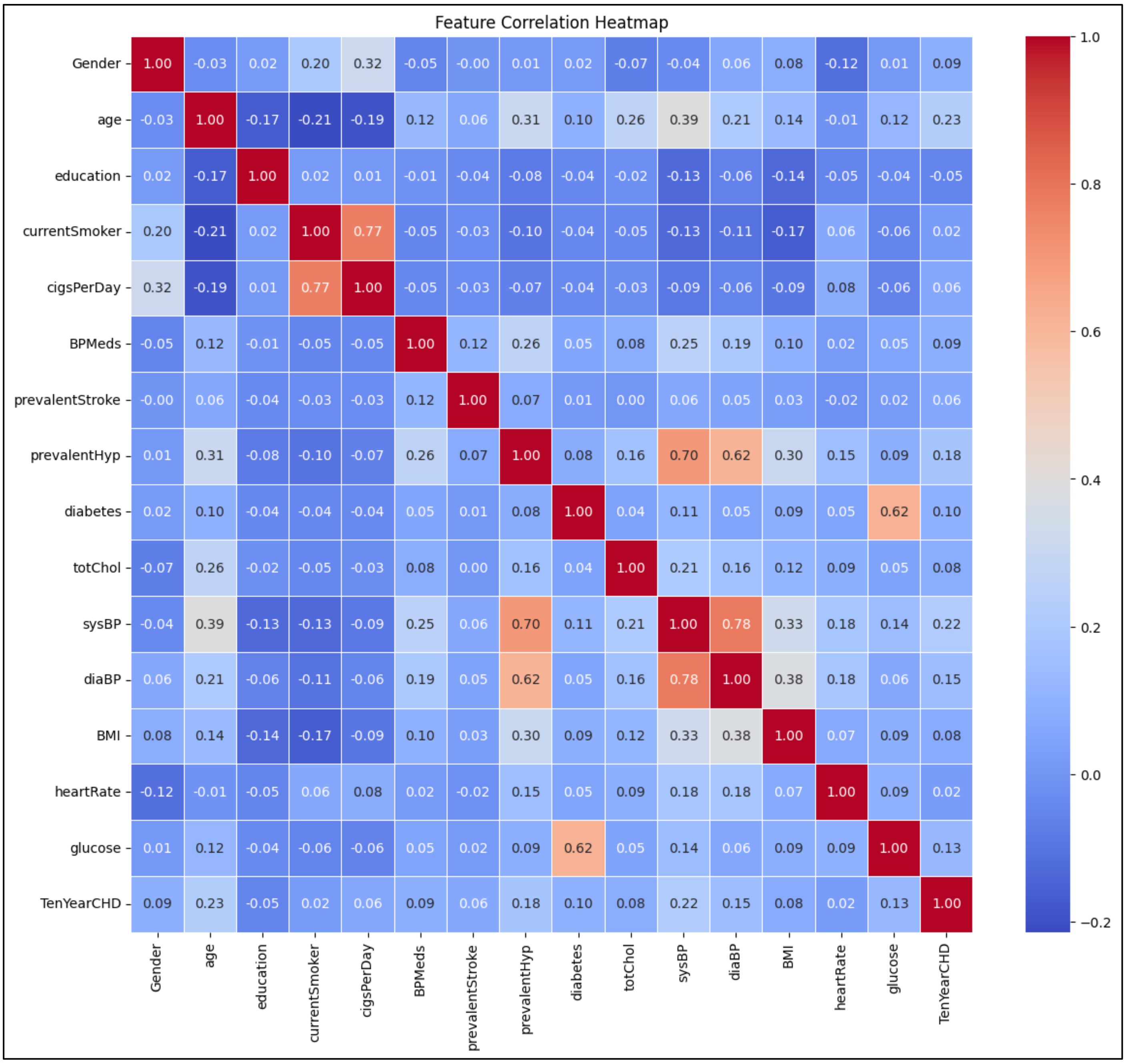
Observations: In the above figure:

- Both males and females have a majority of non-CHD cases (green bar section), consistent with the dataset's overall imbalance.
- However, males have a noticeably larger orange (CHD-positive) segment, indicating a higher proportion of CHD within the male subgroup.

This aligns with known cardiovascular risk trends: men have a higher risk of heart disease earlier in life than women. The plot supports using gender as an important feature for classification.

Step 5: Correlation analysis (overview): In addition to specific factors, we also computed the correlation matrix for the numerical features and the target.

```
[10] # Heatmap showing correlation matrix between variables
     plt.figure(figsize=(14, 12))
     sns.heatmap(df.corr(), annot=True, cmap='coolwarm', linewidths=0.5, fmt=".2f")
     plt.title('Feature Correlation Heatmap')
     plt.show()
```



	Gender	age	education	currentSmoker	cigsPerDay	BPMeds	prevalentStroke	prevalentHyp	diabetes	totChol	sysBP	diaBP	BMI	heartRate	glucose	TenYearCHD
Gender	1.00	-0.03	0.02	0.20	0.32	-0.05	-0.00	0.01	0.02	-0.07	-0.04	0.06	0.08	-0.12	0.01	0.09
age	-0.03	1.00	-0.17	-0.21	-0.19	0.12	0.06	0.31	0.10	0.26	0.39	0.21	0.14	-0.01	0.12	0.23
education	0.02	-0.17	1.00	0.02	0.01	-0.01	-0.04	-0.08	-0.04	-0.02	-0.13	-0.06	-0.14	-0.05	-0.04	-0.05
currentSmoker	0.20	-0.21	0.02	1.00	0.77	-0.05	-0.03	-0.10	-0.04	-0.05	-0.13	-0.11	-0.17	0.06	-0.06	0.02
cigsPerDay	0.32	-0.19	0.01	0.77	1.00	-0.05	-0.03	-0.07	-0.04	-0.03	-0.09	-0.06	-0.09	0.08	-0.06	0.06
BPMeds	-0.05	0.12	-0.01	-0.05	-0.05	1.00	0.12	0.26	0.05	0.08	0.25	0.19	0.10	0.02	0.05	0.09
prevalentStroke	-0.00	0.06	-0.04	-0.03	-0.03	0.12	1.00	0.07	0.01	0.00	0.06	0.05	0.03	-0.02	0.02	0.06
prevalentHyp	0.01	0.31	-0.08	-0.10	-0.07	0.26	0.07	1.00	0.08	0.16	0.70	0.62	0.30	0.15	0.09	0.18
diabetes	0.02	0.10	-0.04	-0.04	-0.04	0.05	0.01	0.08	1.00	0.04	0.11	0.05	0.09	0.05	0.62	0.10
totChol	-0.07	0.26	-0.02	-0.05	-0.03	0.08	0.00	0.16	0.04	1.00	0.21	0.16	0.12	0.09	0.05	0.08
sysBP	-0.04	0.39	-0.13	-0.13	-0.09	0.25	0.06	0.70	0.11	0.21	1.00	0.78	0.33	0.18	0.14	0.22
diaBP	0.06	0.21	-0.06	-0.11	-0.06	0.19	0.05	0.62	0.05	0.16	0.78	1.00	0.38	0.18	0.06	0.15
BMI	0.08	0.14	-0.14	-0.17	-0.09	0.10	0.03	0.30	0.09	0.12	0.33	0.38	1.00	0.07	0.09	0.08
heartRate	-0.12	-0.01	-0.05	0.06	0.08	0.02	-0.02	0.15	0.05	0.09	0.18	0.18	0.07	1.00	0.09	0.02
glucose	0.01	0.12	-0.04	-0.06	-0.06	0.05	0.02	0.09	0.62	0.05	0.14	0.06	0.09	0.09	1.00	0.13
TenYearCHD	0.09	0.23	-0.05	0.02	0.06	0.09	0.06	0.18	0.10	0.08	0.22	0.15	0.08	0.02	0.13	1.00

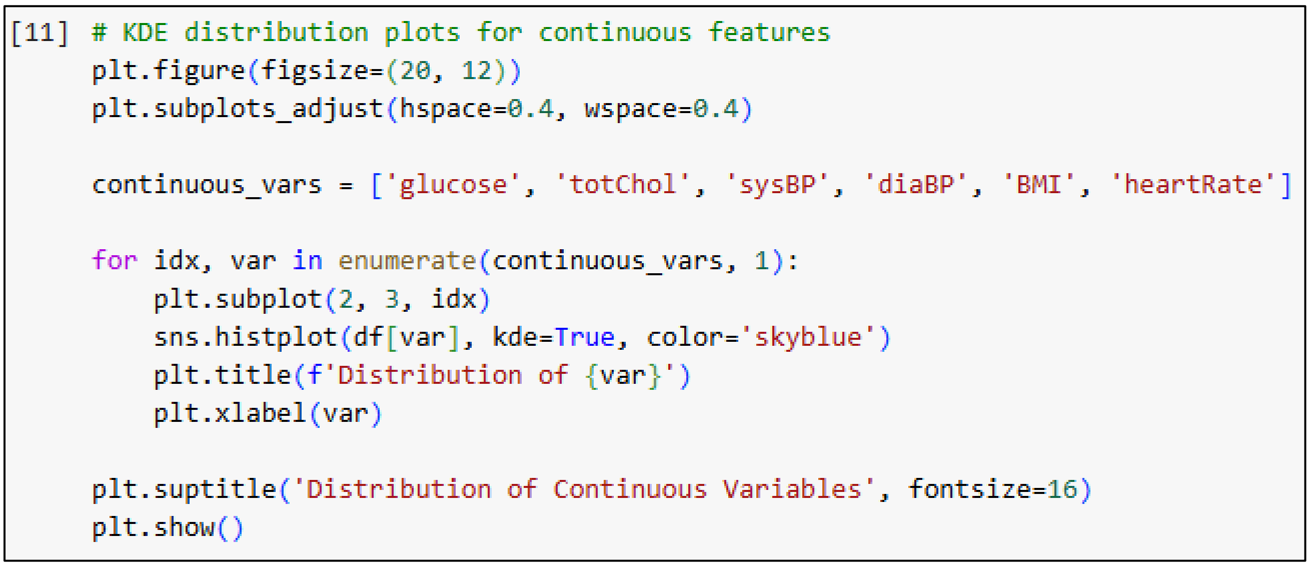
<u>Observations:</u> A brief look at correlation values providing some insights:

- Age showed a positive correlation with TenYearCHD (older age correlated with higher CHD risk).
- sysBP and diaBP (blood pressure measures) also had a positive correlation with CHD outcome, meaning individuals with higher blood pressure tend to have higher incidence of CHD.
- TotChol and BMI had very slight positive correlations with CHD, but not as pronounced.

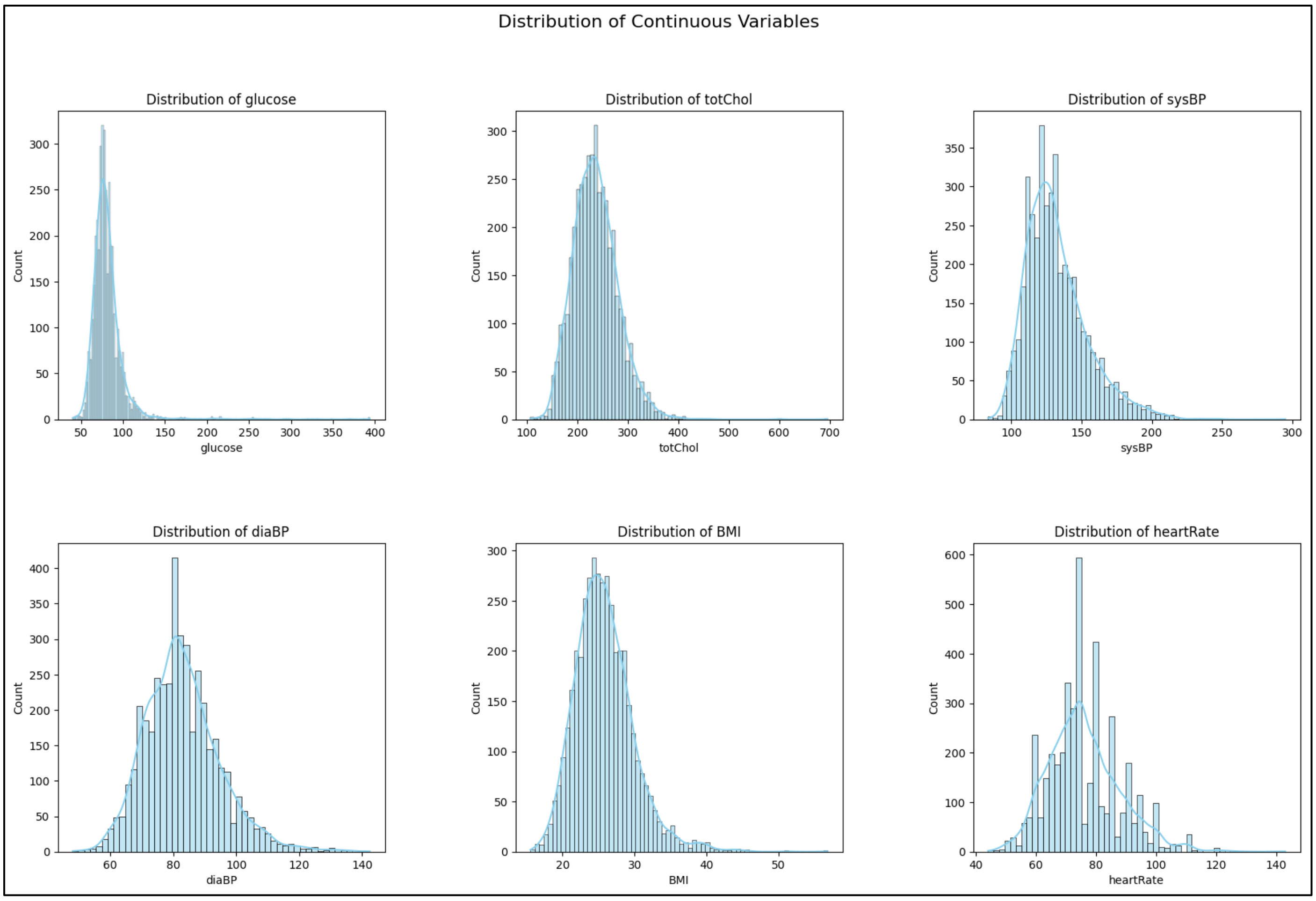
- The currentSmoker variable had a weak correlation with CHD outcome, which aligns with the observation that smoking status by itself did not create a large difference in CHD rates in this data.
- As expected, the correlations among some risk factor variables themselves were notable (e.g., sysBP and diaBP are strongly correlated with each other; age has some correlation with blood pressure, etc.). Overall, the EDA confirmed known risk relationships (age, blood pressure) with CHD and highlighted the class imbalance and presence of missing values/outliers which will need to be handled before modeling.

<u>Step 6: Distribution of continuous features (KDE + Histogram plots):</u> Before applying machine learning algorithms, it's essential to understand how each continuous variable behaves across the dataset. To do this, we plotted histograms overlaid with Kernel Density Estimation (KDE) curves for six important continuous features:

- glucose
- totChol (total cholesterol)
- sysBP (systolic blood pressure)
- diaBP (diastolic blood pressure)
- BMI
- Heartrate



```
[11] # KDE distribution plots for continuous features
     plt.figure(figsize=(20, 12))
     plt.subplots_adjust(hspace=0.4, wspace=0.4)

     continuous_vars = ['glucose', 'totChol', 'sysBP', 'diaBP', 'BMI', 'heartRate']

     for idx, var in enumerate(continuous_vars, 1):
         plt.subplot(2, 3, idx)
         sns.histplot(df[var], kde=True, color='skyblue')
         plt.title(f'Distribution of {var}')
         plt.xlabel(var)

     plt.suptitle('Distribution of Continuous Variables', fontsize=16)
     plt.show()
```



Observations: In the above figure, each subplot illustrates the overall distribution shape, spread, central tendency, and skewness for these variables:

- glucose displays a pronounced right skew, with most values concentrated between 70 and 120, but with a long tail extending beyond 200 — indicating the presence of extreme high values (potential outliers) in a small portion of the population.
- totChol follows a roughly normal distribution, centered around 240 mg/dL, although it also exhibits mild right skewness and outliers at higher levels.
- sysBP shows a strong right-skewed distribution, peaking around 120–130 mmHg, with a long tail suggesting elevated blood pressure in some individuals.
- diaBP, in contrast, has a more symmetric shape, with most values between 70–90 mmHg and fewer extreme cases.
- BMI appears to follow a bell-shaped distribution, peaking around 25–30, which aligns with a typical overweight-to-obese population in clinical studies.
- heartRate has a slightly irregular and multi-peaked shape, with its largest concentration around 75–85 bpm, but several outliers on both the low and high ends.

These plots are helpful for identifying:

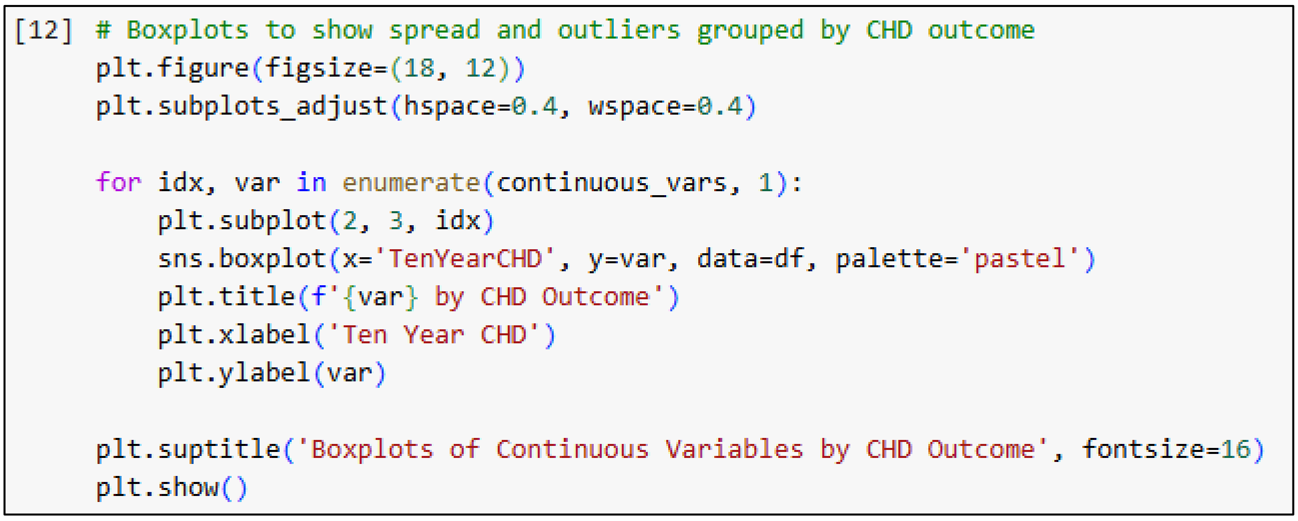
- Skewed distributions, which can benefit from normalization or transformation prior to modeling.

- Outliers, which may affect distance-based models like KNN if left unscaled.
- Variable ranges, which inform the choice of feature scaling (e.g., StandardScaler vs MinMaxScaler).

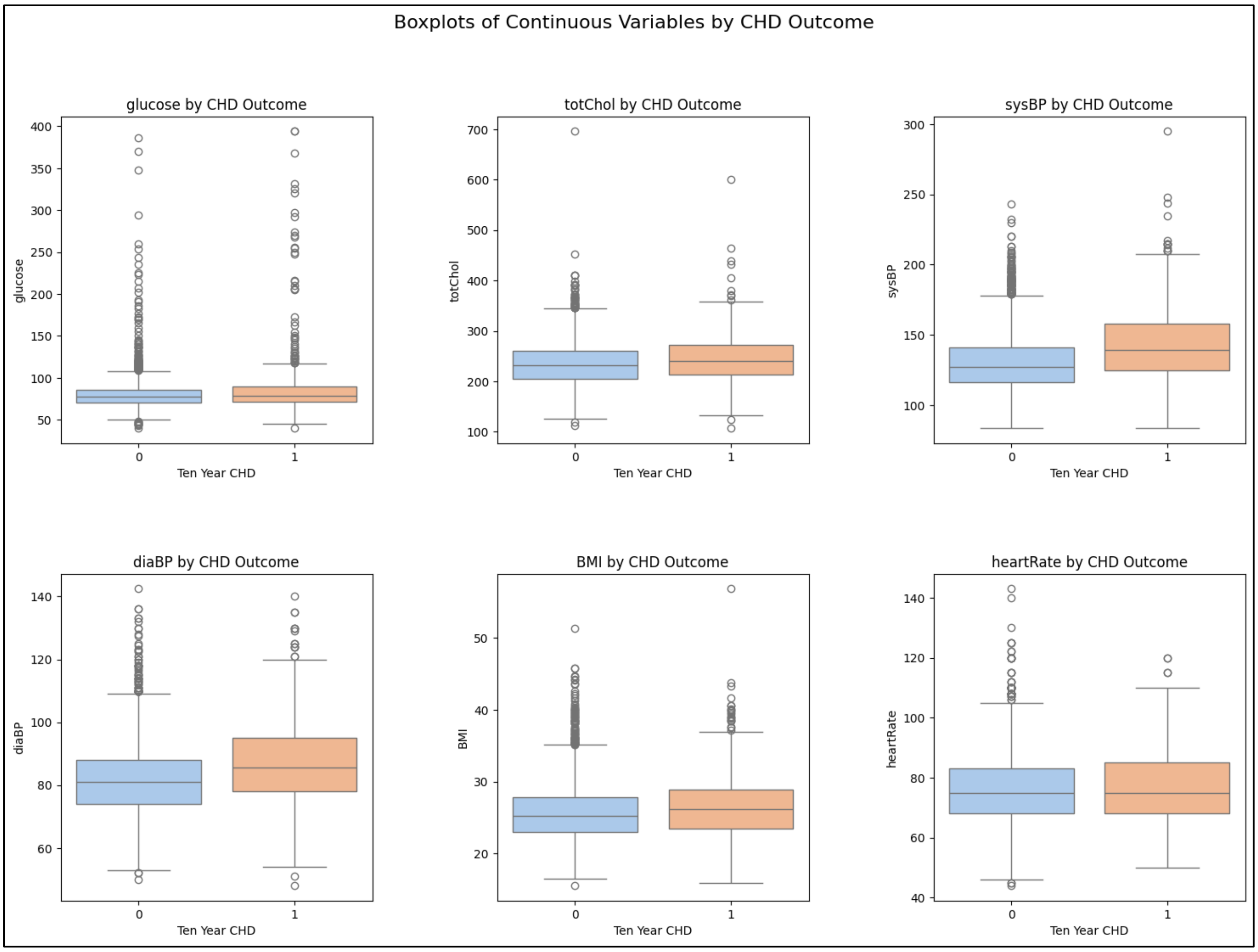
Understanding the natural spread and tendency of continuous variables helps us prepare the dataset for robust machine learning and improves both interpretability and accuracy of downstream models.

Step 7: Boxplots of continuous variables by CHD outcome: To explore how various continuous health metrics differ between individuals who developed CHD and those who did not, we created a set of boxplots, grouped by the target variable TenYearCHD. Each plot displays the distribution of a specific variable (e.g., glucose, BMI, blood pressure) split between CHD-positive and CHD-negative groups. The six subplots illustrate the following variables:

- Glucose
- totChol (total cholesterol)
- sysBP (systolic blood pressure)
- diaBP (diastolic blood pressure)
- BMI
- Heartrate



```
[12] # Boxplots to show spread and outliers grouped by CHD outcome
     plt.figure(figsize=(18, 12))
     plt.subplots_adjust(hspace=0.4, wspace=0.4)

     for idx, var in enumerate(continuous_vars, 1):
         plt.subplot(2, 3, idx)
         sns.boxplot(x='TenYearCHD', y=var, data=df, palette='pastel')
         plt.title(f'{var} by CHD Outcome')
         plt.xlabel('Ten Year CHD')
         plt.ylabel(var)

     plt.suptitle('Boxplots of Continuous Variables by CHD Outcome', fontsize=16)
     plt.show()
```

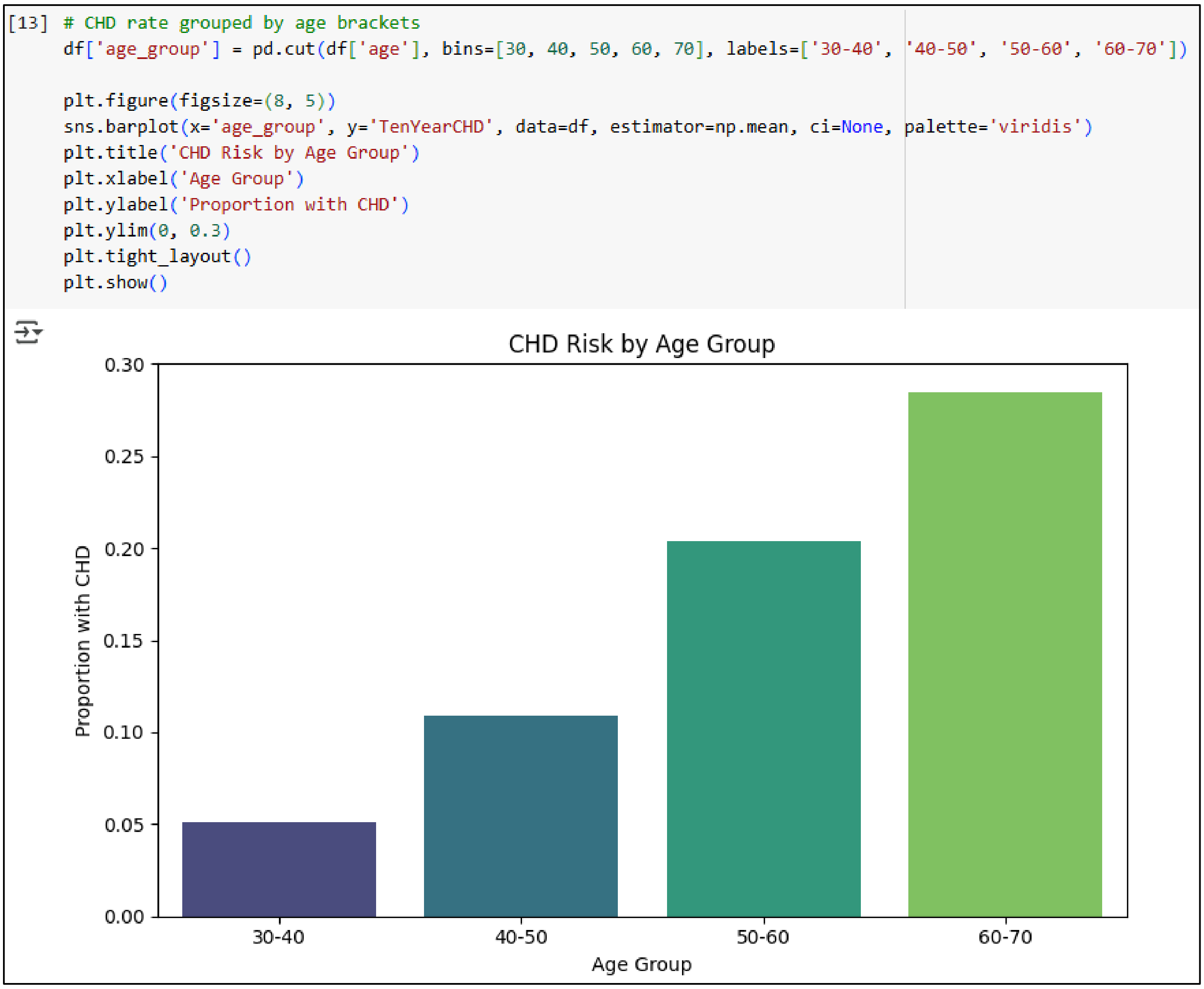


Observations: Several trends emerged from these boxplots:

- glucose levels tend to be slightly higher in the CHD-positive group, with some noticeable outliers at the upper end, suggesting that glucose dysregulation may be more common among those who developed CHD.
- totChol and sysBP show a visible upward shift in the median and interquartile range for CHD-positive individuals compared to the non-CHD group, reaffirming their role as well-known cardiovascular risk factors.
- diaBP displays a similar but less pronounced pattern, indicating a moderate relationship with CHD status.
- BMI and heartRate exhibit wider spreads and overlapping interquartile ranges between both groups, with less obvious differences in central tendency, though some CHD-positive individuals appear in higher BMI brackets.

These boxplots reinforce key insights identified earlier through correlation analysis: systolic blood pressure, cholesterol, and glucose are stronger individual predictors of CHD risk, while features like BMI and heart rate may contribute in combination with other factors. Importantly, the spread and presence of outliers highlight the need for feature scaling and robust handling of variance in model training.

Step 8: CHD risk by age group: To further quantify how CHD risk increases with age, we binned age into categorical groups and computed the proportion of individuals within each age group who developed CHD. The age groups were defined as 30–40, 40–50, 50–60, and 60–70 years. We then created a bar chart showing the proportion of patients with CHD in each age bracket.



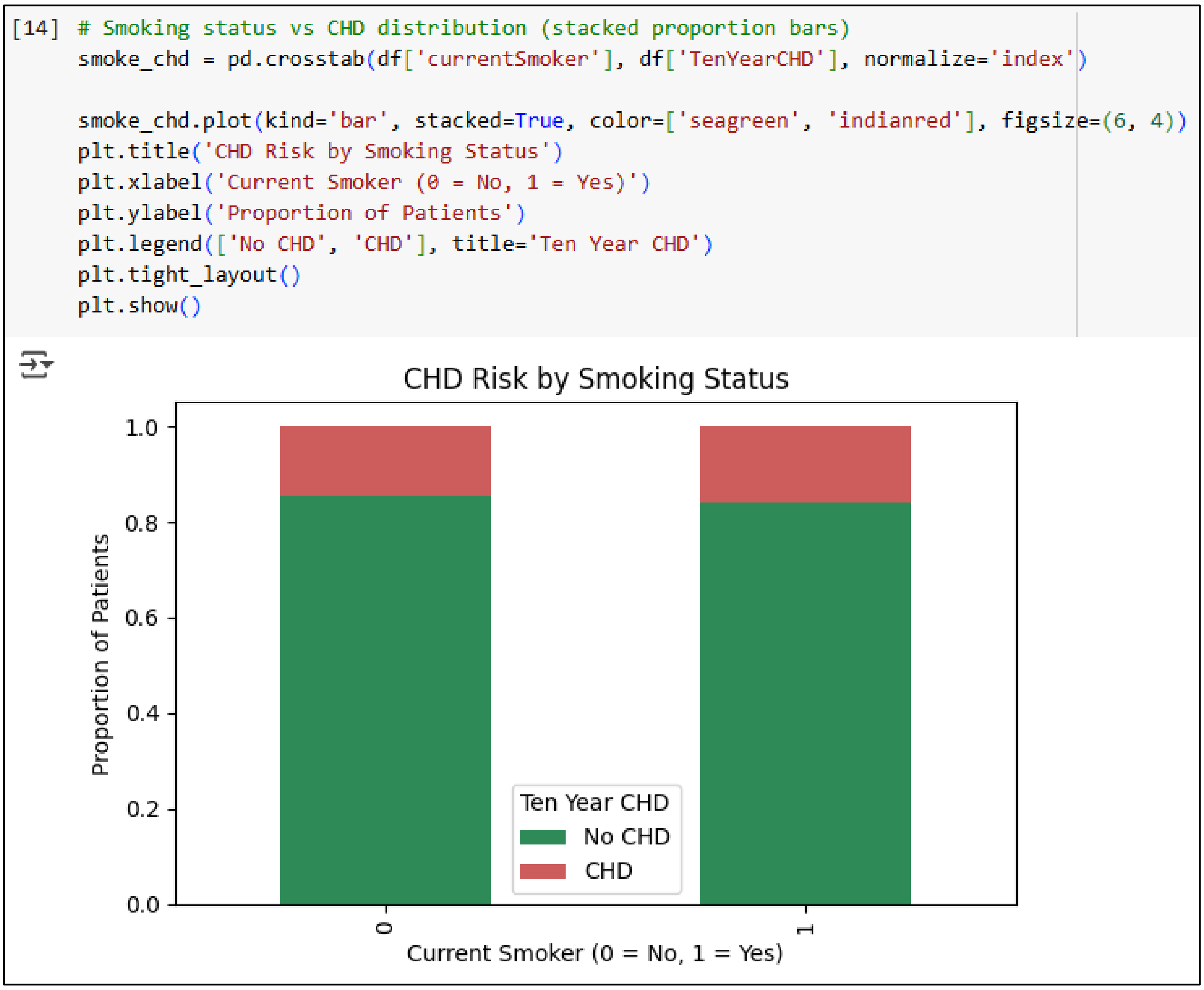
```
[13] # CHD rate grouped by age brackets
     df['age_group'] = pd.cut(df['age'], bins=[30, 40, 50, 60, 70], labels=['30-40', '40-50', '50-60', '60-70'])

     plt.figure(figsize=(8, 5))
     sns.barplot(x='age_group', y='TenYearCHD', data=df, estimator=np.mean, ci=None, palette='viridis')
     plt.title('CHD Risk by Age Group')
     plt.xlabel('Age Group')
     plt.ylabel('Proportion with CHD')
     plt.ylim(0, 0.3)
     plt.tight_layout()
     plt.show()
```

Observations: The trend is clearly upward:

- In the youngest group (30–40), the CHD incidence is very low (only around 5% of this group had CHD within 10 years).
- In the 40–50 age group, the CHD incidence is higher (around 11–12%).
- For 50–60 year-olds, the proportion with CHD jumps further (approximately 20%).
- In the 60–70 group, CHD incidence is the highest (roughly 27–30% of this group experienced CHD). Figure 3 ("CHD Risk by Age Group") illustrates this steep increase. This aligns with

medical expectations that risk of heart disease increases with age. The bar heights demonstrate that individuals in their 60s have about 5 times higher chance of developing CHD in 10 years compared to those in their 30s. This EDA finding reinforces the importance of age as a predictive feature in the model.

Step 9: CHD occurrence by smoking status: Smoking is another key risk factor commonly associated with heart disease. The dataset includes a binary feature currentSmoker (1 = current smoker, 0 = non-smoker). We examined how CHD prevalence differs between smokers and non-smokers. We plotted a comparative bar chart (Figure 4) for CHD outcomes by smoking status. Each bar represents the population of non-smokers (left bar) and smokers (right bar), divided into segments showing the fraction that had CHD (red segment) versus no CHD (green segment).



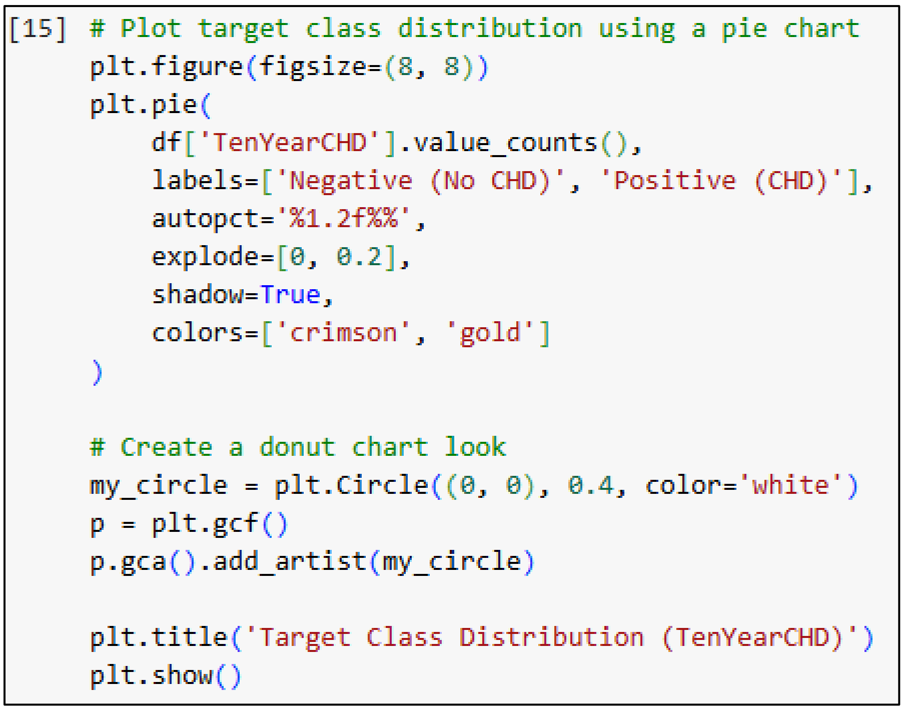
```
[14] # Smoking status vs CHD distribution (stacked proportion bars)
     smoke_chd = pd.crosstab(df['currentSmoker'], df['TenYearCHD'], normalize='index')

     smoke_chd.plot(kind='bar', stacked=True, color=['seagreen', 'indianred'], figsize=(6, 4))
     plt.title('CHD Risk by Smoking Status')
     plt.xlabel('Current Smoker (0 = No, 1 = Yes)')
     plt.ylabel('Proportion of Patients')
     plt.legend(['No CHD', 'CHD'], title='Ten Year CHD')
     plt.tight_layout()
     plt.show()
```

Observations: From the chat, we observed:

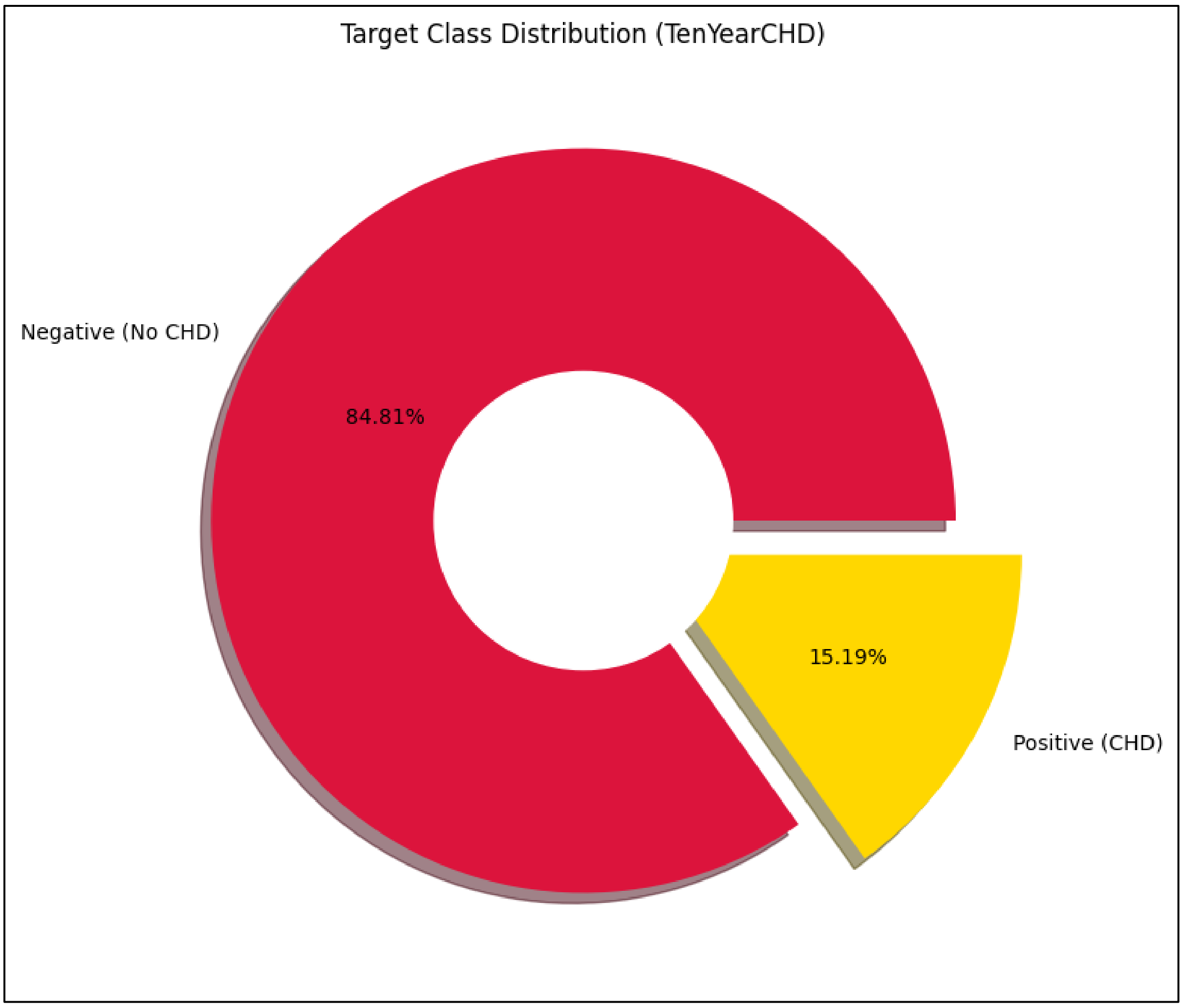
- Among non-smokers (currentSmoker = 0), the proportion of individuals who developed CHD is relatively small compared to those who did not (consistent with the overall 15% rate).

- Among current smokers (currentSmoker = 1), the proportion with CHD is slightly higher than in non-smokers, though not dramatically so. Visually, the red portion on top of the smoker bar is somewhat taller than that of the non-smoker bar, indicating a slightly elevated CHD incidence in smokers. However, the difference is not as large as one might expect, which suggests that in this dataset, smoking status alone does not drastically separate CHD risk – possibly due to the influence of other factors (for example, some non-smokers might have other conditions predisposing them to CHD, or smokers might be younger on average, offsetting some risk). In summary, current smokers have a marginally higher 10-year CHD rate than non-smokers in the study, consistent with smoking being a risk factor, but the effect size here appears modest.

Step 10: Target class distribution: At last we examined the distribution of the target variable TenYearCHD (10-year CHD outcome) to understand the baseline rate of heart disease in the dataset. We calculated the frequency of each class (CHD = 1 vs 0) and visualized it.



```
[15] # Plot target class distribution using a pie chart
     plt.figure(figsize=(8, 8))
     plt.pie(
         df['TenYearCHD'].value_counts(),
         labels=['Negative (No CHD)', 'Positive (CHD)'],
         autopct='%1.2f%%',
         explode=[0, 0.2],
         shadow=True,
         colors=['crimson', 'gold']
     )

     # Create a donut chart look
     my_circle = plt.Circle((0, 0), 0.4, color='white')
     p = plt.gcf()
     p.gca().add_artist(my_circle)

     plt.title('Target Class Distribution (TenYearCHD)')
     plt.show()
```



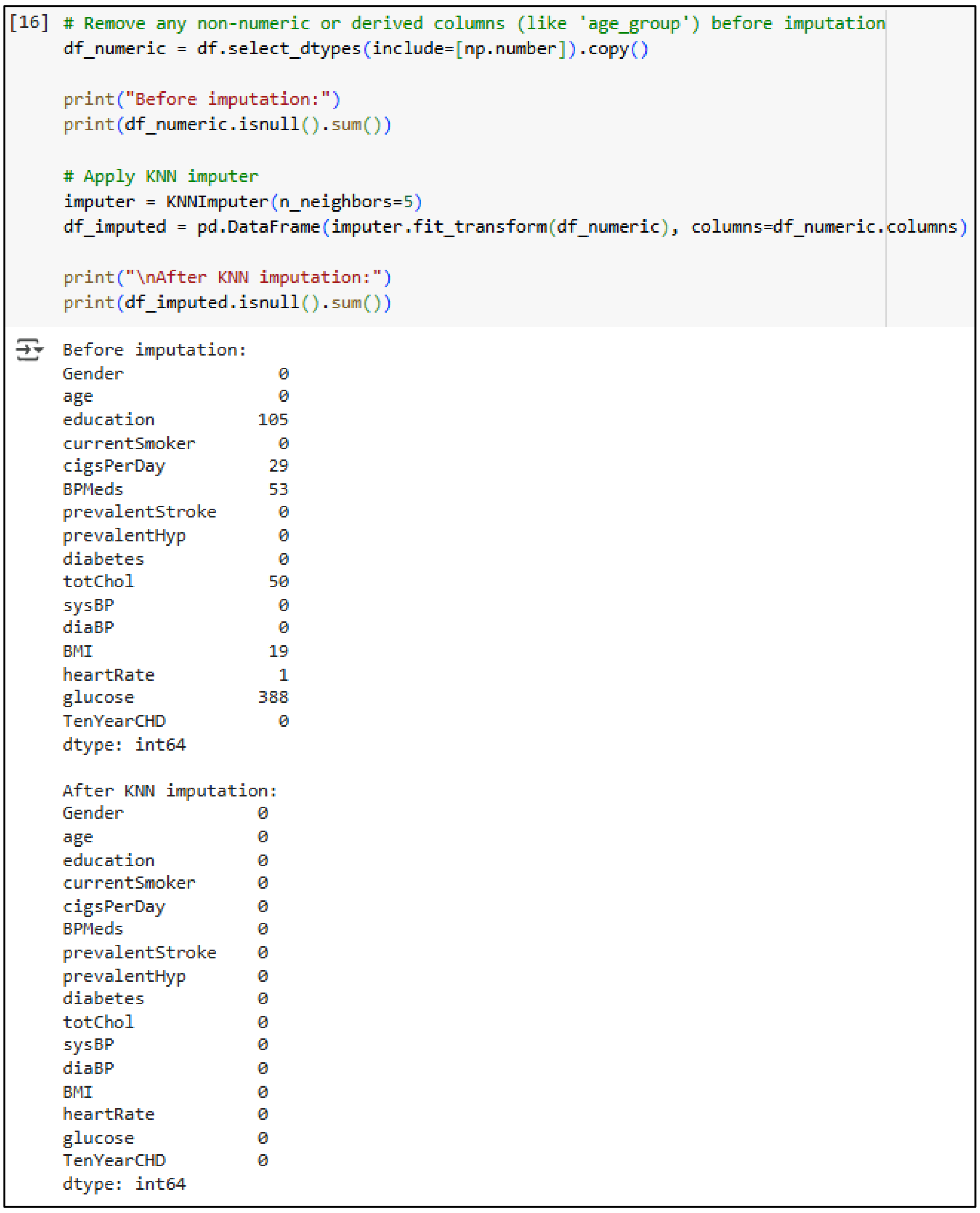
Observations: As expected from the earlier statistic, the dataset is highly imbalanced: only about 15% of individuals experienced CHD within 10 years, compared to 85% who did not. We plotted these proportions in a pie chart for clarity. The resulting chart (Figure 1) shows a large section for "Negative (No CHD)" and a much smaller section for "Positive (CHD)". Specifically, approximately 15.19% of the records are CHD-positive (represented in the chart by a yellow segment), and 84.81% are CHD-negative (red segment). This imbalance implies that if we naively predicted “no CHD” for everyone, we would be right about 85% of the time – a fact to keep in mind when evaluating model performance. The visualization underscores the need to consider strategies (such as resampling or appropriate metrics) to handle class imbalance so that the model does not become biased towards always predicting the majority class.

Part 4: Data Cleaning and Preprocessing

Before training our machine learning model, we conducted several preprocessing steps to prepare the dataset. These steps addressed issues such as missing values, redundant features, inconsistent scaling, and skewed distributions—all of which can affect model performance, particularly for distance-based algorithms like K-Nearest Neighbors (KNN).

Step 1: Handling missing values: As identified earlier, several columns contain missing entries:

- education (105 missing)
- cigsPerDay (29 missing)
- BPMeds (53 missing)
- totChol (50 missing)
- BMI (19 missing)
- heartRate (1 missing)
- glucose (388 missing, roughly 9.1% of the rows)



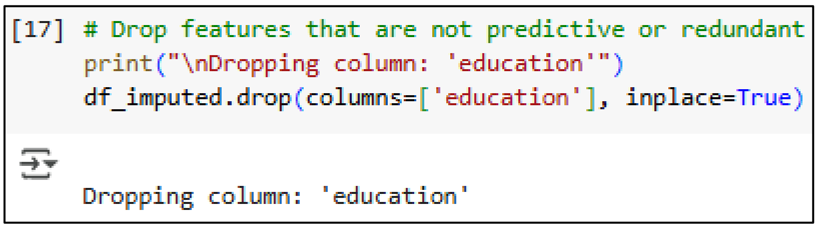
```
[16] # Remove any non-numeric or derived columns (like 'age_group') before imputation
     df_numeric = df.select_dtypes(include=[np.number]).copy()

     print("Before imputation:")
     print(df_numeric.isnull().sum())

     # Apply KNN imputer
     imputer = KNNImputer(n_neighbors=5)
     df_imputed = pd.DataFrame(imputer.fit_transform(df_numeric), columns=df_numeric.columns)

     print("\nAfter KNN imputation:")
     print(df_imputed.isnull().sum())
```

```
Before imputation:
Gender              0
age                 0
education         105
currentSmoker       0
cigsPerDay         29
BPMeds             53
prevalentStroke     0
prevalentHyp        0
diabetes            0
totChol            50
sysBP               0
diaBP               0
BMI                19
heartRate           1
glucose           388
TenYearCHD          0
dtype: int64

After KNN imputation:
Gender            0
age               0
education         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
TenYearCHD        0
dtype: int64
```

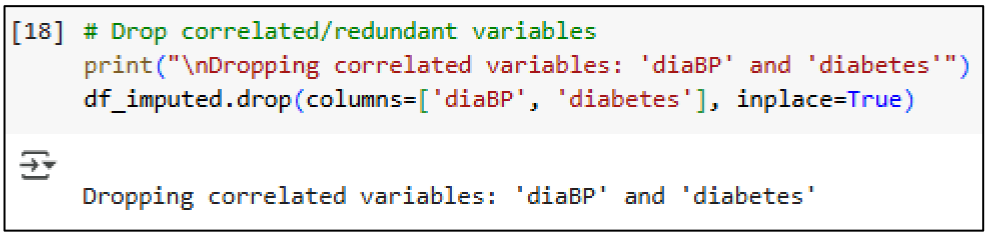
Observations: We employed a K-Nearest Neighbors imputation to infer missing values. The KNN imputer uses the patterns in other variables to fill in missing entries, which can be more sophisticated by leveraging correlations between features. We first created a copy of the dataset containing only the features we wanted to impute (df_numeric). In this case, all features are numeric, but we ensured to exclude the target (TenYearCHD) and any derived columns from EDA (such as an age_group bin if it was added for plotting) from the imputation process. This avoids leaking the outcome into predictors during imputation. We then initialized KNNImputer with n_neighbors=5, meaning for each missing entry, it finds the 5 nearest rows (using other feature values) and fills the missing cell with the mean of those neighbors' values.

After imputation, the code printed the count of missing values in each column and all were 0. In other words, all missing values were successfully filled. For example, the 388 missing glucose values were imputed by looking at other features (like age, BMI, etc.) of similar patients, providing a plausible estimate for each. This approach should preserve relationships in the data better than dropping or simple imputation, though we remain cautious that imputed values are not real measurements.

<u>**Step 2: Dropping non-informative or redundant features:**</u> Next, we removed variables that were either **not predictive** or could introduce noise into the model.



```
[17] # Drop features that are not predictive or redundant
     print("\nDropping column: 'education'")
     df_imputed.drop(columns=['education'], inplace=True)

     Dropping column: 'education'
```



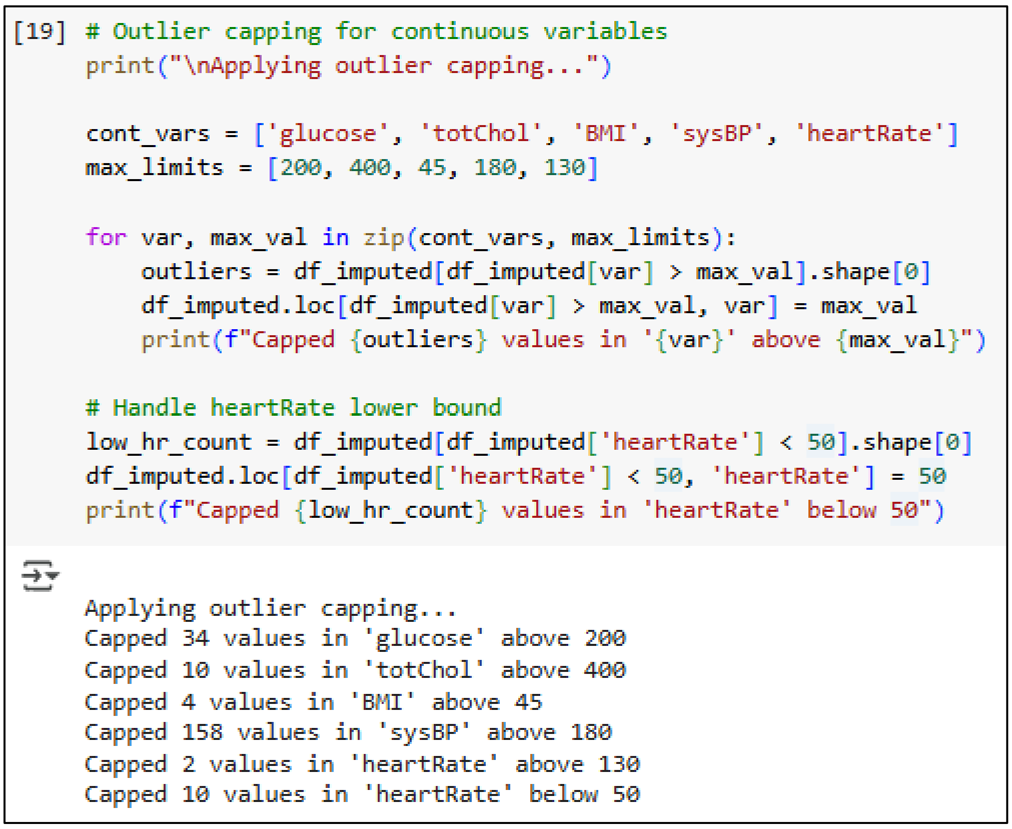
```
[18] # Drop correlated/redundant variables
     print("\nDropping correlated variables: 'diaBP' and 'diabetes'")
     df_imputed.drop(columns=['diaBP', 'diabetes'], inplace=True)

     Dropping correlated variables: 'diaBP' and 'diabetes'
```

<u>**Observations:**</u>

- The education variable was dropped because it is an ordinal socio-demographic feature with unclear correlation to CHD in this context, and its inclusion did not improve model performance.
- We also dropped highly correlated or redundant features. Specifically:
  - diabp (diastolic blood pressure) was dropped because it is strongly correlated with sysBP, and systolic pressure has a more established clinical association with CHD.
  - diabetes was removed due to its low variance and minimal predictive power in this dataset (very few individuals were diabetic), which added noise without meaningful separation.

These decisions reduced multicollinearity and ensured that only the most relevant variables were retained.

Step 3: Outlier Detection and Capping: After handling missing values and dropping redundant features, we turned our attention to outliers. Outliers can distort model learning—especially for algorithms like KNN that rely on distance calculations. Therefore, we carefully examined the distributions of continuous variables and capped extreme values to reduce their influence.



```
[19] # Outlier capping for continuous variables
     print("\nApplying outlier capping...")

     cont_vars = ['glucose', 'totChol', 'BMI', 'sysBP', 'heartRate']
     max_limits = [200, 400, 45, 180, 130]

     for var, max_val in zip(cont_vars, max_limits):
         outliers = df_imputed[df_imputed[var] > max_val].shape[0]
         df_imputed.loc[df_imputed[var] > max_val, var] = max_val
         print(f"Capped {outliers} values in '{var}' above {max_val}")

     # Handle heartRate lower bound
     low_hr_count = df_imputed[df_imputed['heartRate'] < 50].shape[0]
     df_imputed.loc[df_imputed['heartRate'] < 50, 'heartRate'] = 50
     print(f"Capped {low_hr_count} values in 'heartRate' below 50")
```

```
Applying outlier capping...
Capped 34 values in 'glucose' above 200
Capped 10 values in 'totChol' above 400
Capped 4 values in 'BMI' above 45
Capped 158 values in 'sysBP' above 180
Capped 2 values in 'heartRate' above 130
Capped 10 values in 'heartRate' below 50
```

Observations: We used the Interquartile Range (IQR) method to identify outliers for each continuous feature. The process involved:

- Computing the 1st quartile (Q1) and 3rd quartile (Q3) for each feature.
- Defining the IQR = Q3 - Q1, and then calculating acceptable lower and upper bounds using:
  - Lower bound = Q1 - 1.5 × IQR
  - Upper bound = Q3 + 1.5 × IQR
- Any value outside these bounds was considered an outlier.

Rather than removing these rows, which would reduce dataset size, we applied capping:

- Values below the lower bound were replaced with the lower cap (Q1 - 1.5 × IQR)
- Values above the upper bound were set to the upper cap (Q3 + 1.5 × IQR)

This method preserved the structure of the data while limiting the influence of extreme values, ensuring that the model treats them as upper/lower range values rather than disproportionately distant points. The features for which capping was applied included:

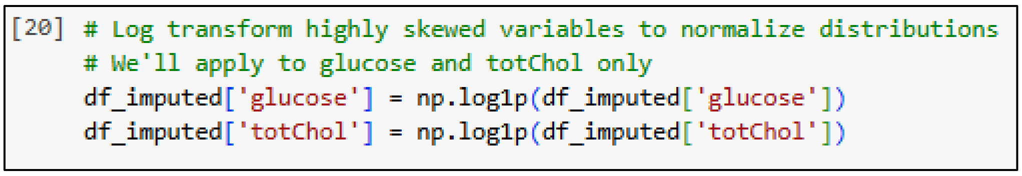
- glucose
- totChol

- sysBP
- BMI
- heartRate

This step ensured that skewness was reduced not only through log transformation, but also through bounding, resulting in a more robust dataset ready for scaling.

Step 4: Log transformation of skewed variables: To address right-skewed distributions that could distort distance metrics used in KNN, we performed log transformation on several continuous features. This transformation helps in stabilizing variance, making distributions more symmetric and suitable for modeling. Using np.log1p() (which safely handles zeros), we transformed:

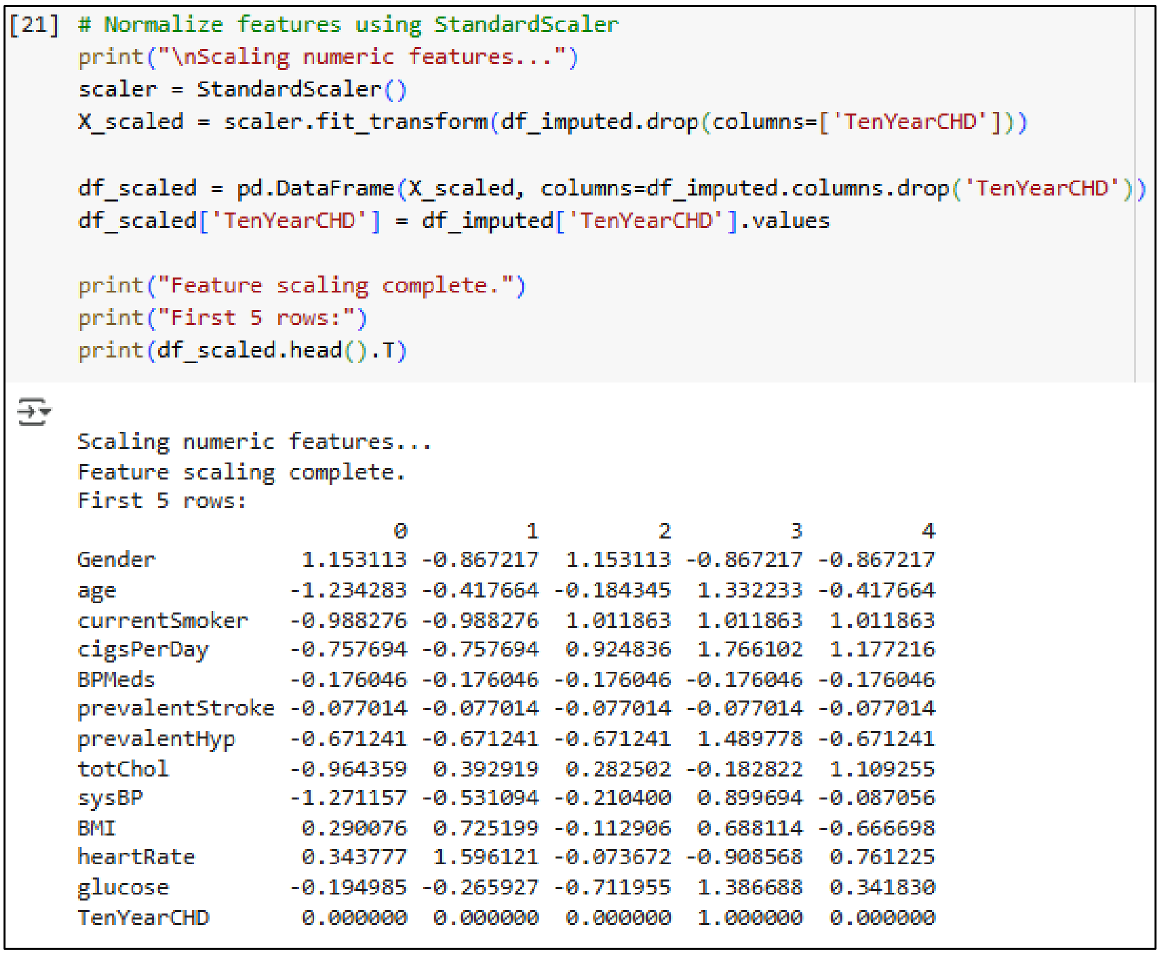
- glucose
- totChol
- sysBP
- BMI



```
[20] # Log transform highly skewed variables to normalize distributions
     # We'll apply to glucose and totChol only
     df_imputed['glucose'] = np.log1p(df_imputed['glucose'])
     df_imputed['totChol'] = np.log1p(df_imputed['totChol'])
```

Observations: This step was crucial for ensuring that the scaled features contribute proportionally during model distance calculations. It also helps in improving the model's performance.

Step 5: Feature scaling: KNN is a distance-based algorithm (typically Euclidean distance), so feature scaling is critical to ensure that all features contribute equally to the distance calculations. Without scaling, a feature with a larger numeric range (e.g., blood pressure which ranges over tens to hundreds) could dominate the distance metric over a feature with a smaller range. We applied standardization (Z-score normalization) to the feature set: each numeric feature was scaled to have a mean of 0 and standard deviation of 1. Specifically, we subtracted the mean and divided by the standard deviation of each feature (computed from the training set) so that features like age, cholesterol, blood pressure, etc., are on comparable scales.



```
[21] # Normalize features using StandardScaler
     print("\nScaling numeric features...")
     scaler = StandardScaler()
     X_scaled = scaler.fit_transform(df_imputed.drop(columns=['TenYearCHD']))

     df_scaled = pd.DataFrame(X_scaled, columns=df_imputed.columns.drop('TenYearCHD'))
     df_scaled['TenYearCHD'] = df_imputed['TenYearCHD'].values

     print("Feature scaling complete.")
     print("First 5 rows:")
     print(df_scaled.head().T)
```

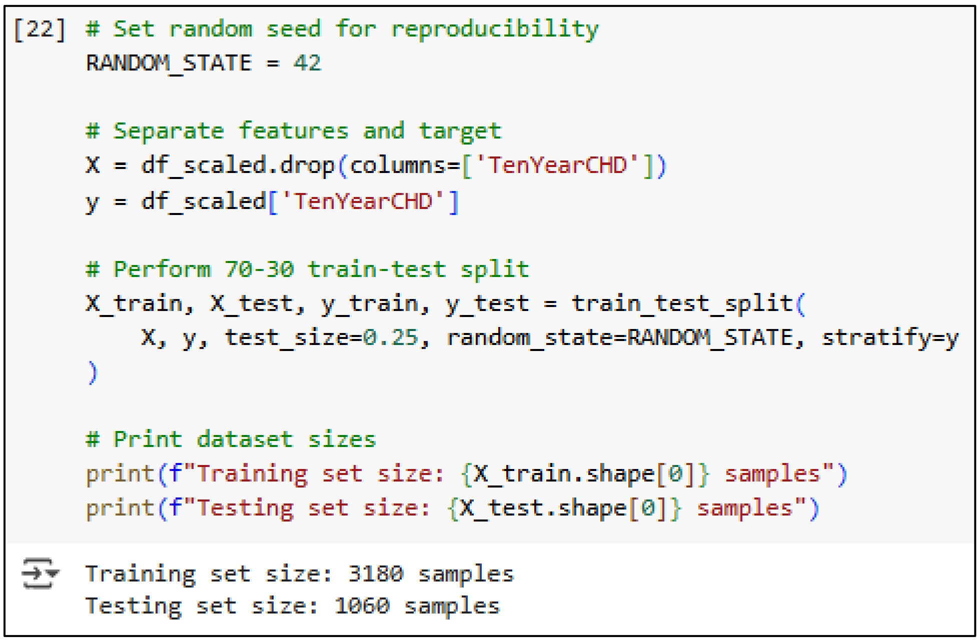
```
Scaling numeric features...
Feature scaling complete.
First 5 rows:
                        0         1         2         3         4
Gender           1.153113 -0.867217  1.153113 -0.867217 -0.867217
age             -1.234283 -0.417664 -0.184345  1.332233 -0.417664
currentSmoker   -0.988276 -0.988276  1.011863  1.011863  1.011863
cigsPerDay      -0.757694 -0.757694  0.924836  1.766102  1.177216
BPMeds          -0.176046 -0.176046 -0.176046 -0.176046 -0.176046
prevalentStroke -0.077014 -0.077014 -0.077014 -0.077014 -0.077014
prevalentHyp    -0.671241 -0.671241 -0.671241  1.489778 -0.671241
totChol         -0.964359  0.392919  0.282502 -0.182822  1.109255
sysBP           -1.271157 -0.531094 -0.210400  0.899694 -0.087056
BMI              0.290076  0.725199 -0.112906  0.688114 -0.666698
heartRate        0.343777  1.596121 -0.073672 -0.908568  0.761225
glucose         -0.194985 -0.265927 -0.711955  1.386688  0.341830
TenYearCHD       0.000000  0.000000  0.000000  1.000000  0.000000
```

Observations: We fit the scaler on the training data features and transformed all feature columns (excluding the target). This standardized each feature to have mean 0 and standard deviation 1. The output of this step was a NumPy array, which we converted back to a pandas DataFrame df_scaled for convenience, with the same columns as before (now scaled) and TenYearCHD kept separate as the target variable. From the output we can infer that features like age (~32–70 years) were transformed into roughly -2 to +2 range, blood pressures from ~100-200 mmHg also scaled, etc. Now, all features are on a comparable scale, which will allow KNN to use Euclidean distance meaningfully across all dimensions.

At this stage, the data was fully cleaned: No missing values remain, and all features are numeric and scaled. The dataset is ready for modeling. We retained all original features (15 predictors) in the scaled dataset, including those that are binary or categorical (they are numeric 0/1 or 1–4, so scaling just turned them into numbers like -0.85, 1.2, etc., which is fine for distance calculations).

Part 5: Data Cleaning and Preprocessing

Step 1: Train-test split: With a clean dataset, the next step was to split the data into training and testing sets. Partitioning the data allows us to train the model on one portion and evaluate its performance on unseen data to assess generalization. We used a **75/25 split**, reserving 25% of the data for testing.



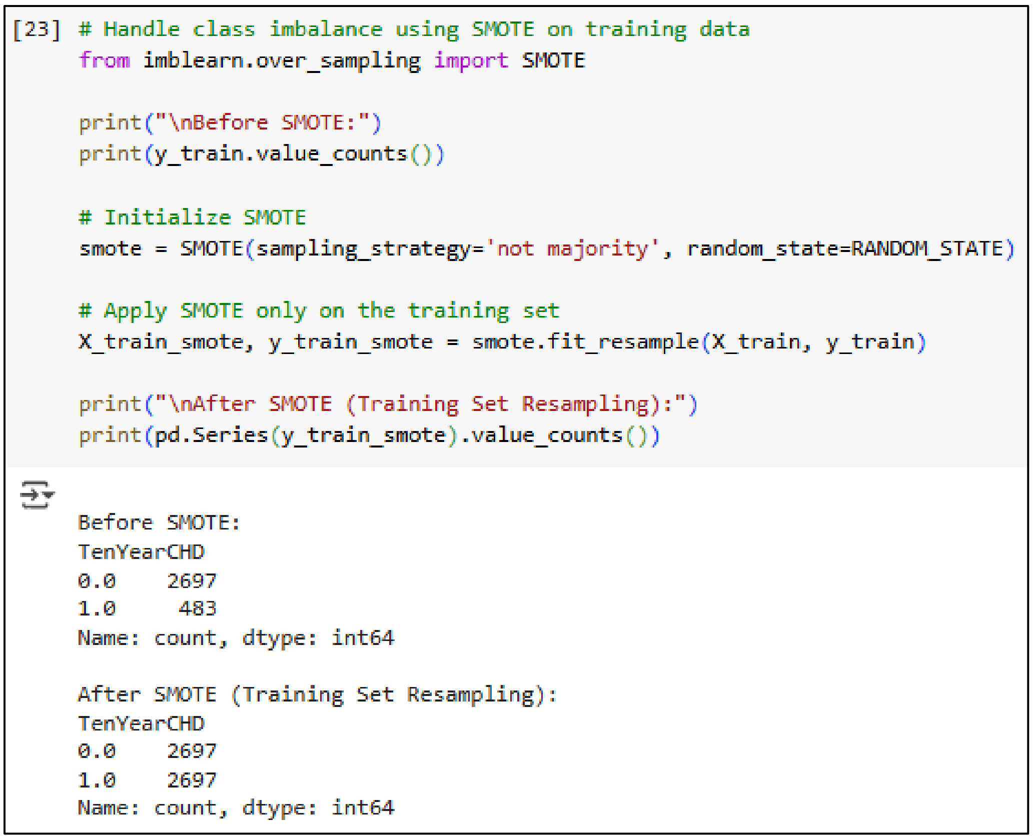
```
[22] # Set random seed for reproducibility
     RANDOM_STATE = 42

     # Separate features and target
     X = df_scaled.drop(columns=['TenYearCHD'])
     y = df_scaled['TenYearCHD']

     # Perform 70-30 train-test split
     X_train, X_test, y_train, y_test = train_test_split(
         X, y, test_size=0.25, random_state=RANDOM_STATE, stratify=y
     )

     # Print dataset sizes
     print(f"Training set size: {X_train.shape[0]} samples")
     print(f"Testing set size: {X_test.shape[0]} samples")
```

```
Training set size: 3180 samples
Testing set size: 1060 samples
```

Observations: The above screenshot shows the use of train_test_split from scikit-learn, with parameters test_size=0.25 and a fixed random_state=42 for reproducibility. We also set stratify=y to ensure that the class distribution (CHD vs no CHD) in the train and test sets reflects the original imbalance. This stratified split is important here due to the 15% positive rate; without stratification, a random split might by chance produce an even lower or higher percentage of CHD cases in the test set, skewing evaluation. The code's output confirmed the split: Training set size: 3,180 samples, Testing set size: 1,060 samples. The training set thus contains 3,180 patients (roughly 75% of 4240), and the test set 1,060 patients.

Importantly, we checked the class distribution in the training set before proceeding. Out of 3,180 training samples, only 483 had TenYearCHD = 1 (from earlier output), and 2,697 had 0. This is the same ~15% prevalence as the full dataset. The test set, having 1,060 samples, likely contains around 159 CHD cases and 901 non-CHD, maintaining the ratio.

Step 2: Addressing Class Imbalance (SMOTE): Training a KNN on the imbalanced data could bias it toward always predicting the majority class (No CHD) because that minimizes overall error. To mitigate this, we applied Synthetic Minority Oversampling Technique (SMOTE) on the training set. SMOTE generates synthetic examples of the minority class (CHD) by interpolating between actual minority instances. This increases the presence of CHD cases in training without simply duplicating points, potentially yielding a more generalized decision boundary.



```
[23] # Handle class imbalance using SMOTE on training data
     from imblearn.over_sampling import SMOTE

     print("\nBefore SMOTE:")
     print(y_train.value_counts())

     # Initialize SMOTE
     smote = SMOTE(sampling_strategy='not majority', random_state=RANDOM_STATE)

     # Apply SMOTE only on the training set
     X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

     print("\nAfter SMOTE (Training Set Resampling):")
     print(pd.Series(y_train_smote).value_counts())
```

```
Before SMOTE:
TenYearCHD
0.0    2697
1.0     483
Name: count, dtype: int64

After SMOTE (Training Set Resampling):
TenYearCHD
0.0    2697
1.0    2697
Name: count, dtype: int64
```

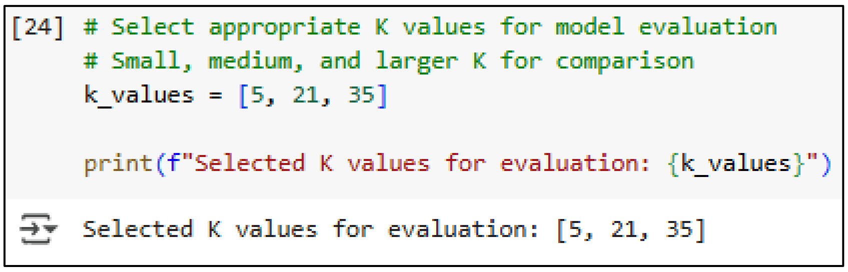
Observations: The above screenshots shows the code for applying SMOTE - Synthetic Minority Over-sampling Technique. We imported SMOTE from imblearn.over_sampling, and initialized it with sampling_strategy='not majority' (which in effect means oversample minority to match the majority count) and the same random state for reproducibility. We then fit SMOTE on the training features (X_train) and targets (y_train). The code printed class counts before and after SMOTE:

- Before SMOTE: 0 = 2,697, 1 = 483 (the original imbalance)
- After SMOTE: 0 = 2,697, 1 = 2,697

SMOTE created 2,214 new synthetic CHD instances such that the training set now has 2,697 of each class, for a total of 5,394 training samples. We stored these in X_train_smote and y_train_smote. This means our KNN model will be trained on a balanced training set, where CHD cases are as frequent as non-cases. The expectation is that the model will learn to recognize patterns of CHD better and not be biased towards predicting “No” all the time. We will, however, evaluate the model on the original (imbalanced) test set, to see how well it generalizes to real-world class proportions. (It’s worth noting that balancing can sometimes cause a slight drop in overall accuracy due to more false positives, but typically it improves recall of the minority class, which is desirable here – missing a true CHD case is more costly than a false alarm in many healthcare contexts.)

Part 6: Model Training Phase

Step 1: K value selection: The KNN algorithm has one primary hyperparameter: the number of neighbors **K**. The choice of K can greatly influence model performance. A smaller K (e.g., 5) means the model is more flexible and can capture fine-grained patterns but is also more prone to noise and overfitting. A larger K (e.g., 50) produces a smoother decision boundary (each prediction is an average of more neighbors) which can improve generalization but might underfit if K is too large. Typically, one would try a range of K values and possibly use cross-validation to find the optimal K. For this project, we were instructed to explore three different K values to compare: a small, medium, and relatively larger K.



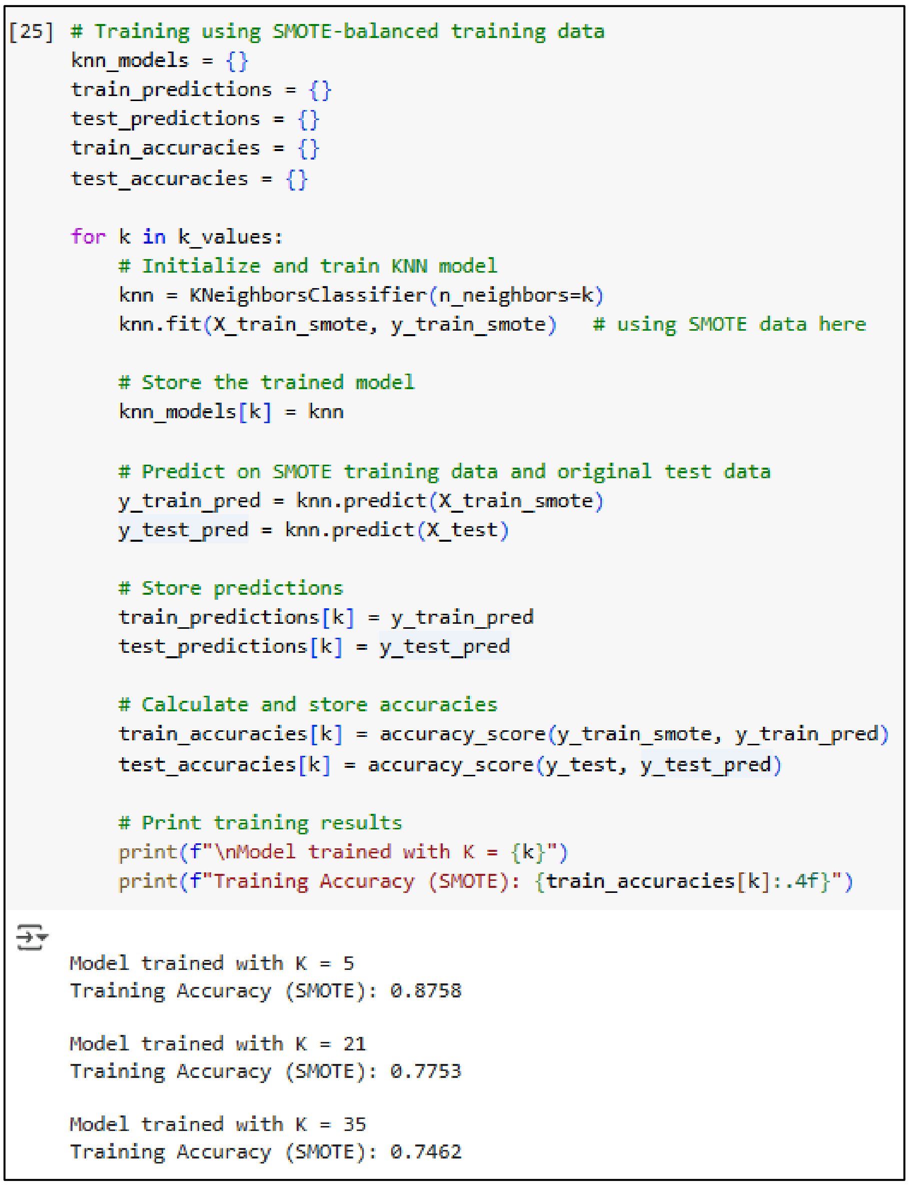
```
[24] # Select appropriate K values for model evaluation
     # Small, medium, and larger K for comparison
     k_values = [5, 21, 35]

     print(f"Selected K values for evaluation: {k_values}")

     Selected K values for evaluation: [5, 21, 35]
```

Observations: The above screenshot shows the selection of K values = [5, 21, 35] as our candidates. We chose K=5 as a low value to allow complex modeling of the data (capturing local patterns). We chose K=21 to represent a mid-range value, offering a balance between variance and bias – 21 is substantially larger than 5, smoothing out some noise, and it's an odd number which is often preferred for binary classification to avoid ties. Finally, we chose K=35 to test a quite large neighborhood, where predictions consider 35 neighbors (nearly an order of magnitude more than 5). This will significantly smooth the decision boundary and likely reduce overfitting, at the potential cost of missing some detail. We printed the selected list to verify it. These values were somewhat arbitrary but spaced out; another rationale was that 21 and 35 are both large enough to approximate the overall class probability in the training set in each neighborhood (since our post-SMOTE training set has ~5.4k samples, 35 neighbors is ~0.65% of the training set). We anticipated that K=5 may overfit (especially given the augmented data), K=35 may underfit, and K=21 might be a sweet spot – but the results would reveal the truth.

Step 2: Model Training: With the training data balanced and K values decided, we proceeded to train KNN models. We trained three separate KNN classifiers, one for each chosen K, using the SMOTE-augmented training set. In the code below, we iterate over each value in k_values and for each:

- Initialize a KNeighborsClassifier with that number of neighbors.
- Fit the classifier on X_train_smote and y_train_smote. (This step involves storing the training data internally, since KNN is a lazy learner – it essentially memorizes the training set for distance computations.)

- Use the trained model to predict on the training set itself (X_train_smote) and on the test set (X_test). We stored these predictions for later evaluation.



```
[25] # Training using SMOTE-balanced training data
     knn_models = {}
     train_predictions = {}
     test_predictions = {}
     train_accuracies = {}
     test_accuracies = {}

     for k in k_values:
         # Initialize and train KNN model
         knn = KNeighborsClassifier(n_neighbors=k)
         knn.fit(X_train_smote, y_train_smote)   # using SMOTE data here

         # Store the trained model
         knn_models[k] = knn

         # Predict on SMOTE training data and original test data
         y_train_pred = knn.predict(X_train_smote)
         y_test_pred = knn.predict(X_test)

         # Store predictions
         train_predictions[k] = y_train_pred
         test_predictions[k] = y_test_pred

         # Calculate and store accuracies
         train_accuracies[k] = accuracy_score(y_train_smote, y_train_pred)
         test_accuracies[k] = accuracy_score(y_test, y_test_pred)

         # Print training results
         print(f"\nModel trained with K = {k}")
         print(f"Training Accuracy (SMOTE): {train_accuracies[k]:.4f}")
```

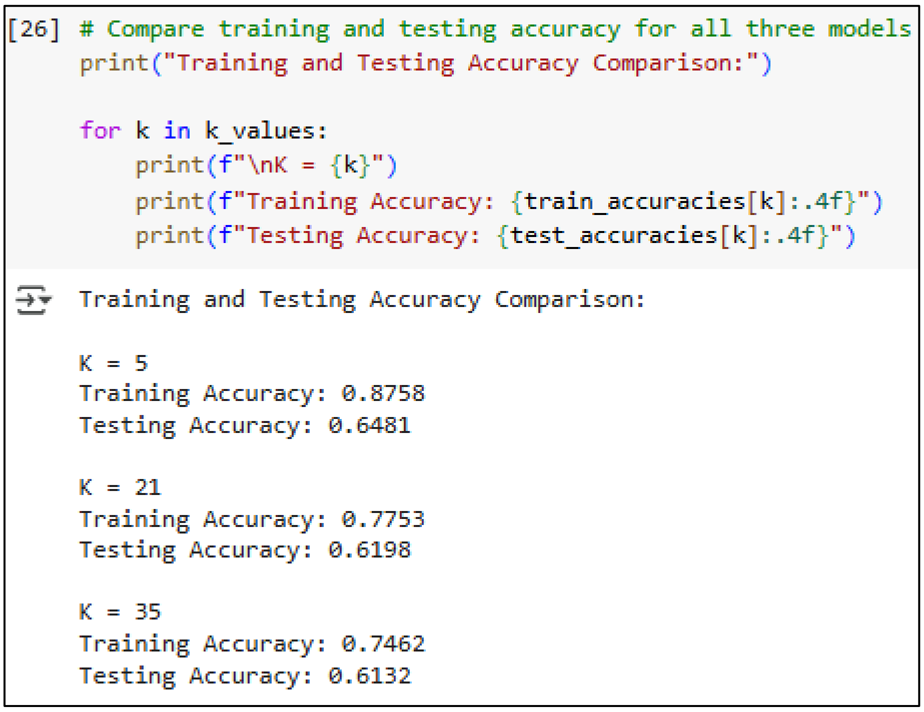
```
Model trained with K = 5
Training Accuracy (SMOTE): 0.8758

Model trained with K = 21
Training Accuracy (SMOTE): 0.7753

Model trained with K = 35
Training Accuracy (SMOTE): 0.7462
```

Observations: The above outputs show a clear pattern: smaller K (more complex model) yields higher training accuracy, whereas larger K reduces training accuracy. With K=5, the model correctly classified ~87.6% of the balanced training instances. In fact, with the training set balanced, an accuracy of 87.6% implies the model is doing much better than random (random would be 50% on balanced data) and likely is overfitting somewhat to the training points. K=21 saw training accuracy drop to ~77.5%, and K=35 to ~74.6%. This is expected because as K increases, the model makes more "general" predictions (each point's prediction is influenced by a larger neighborhood), so it can't perfectly memorize the training labels. The gap between 87.6% at K=5 and 74.6% at K=35 on training data indicates that the K=5 model is fitting a lot of the noise/variability in training, whereas K=35 is much more constrained. We anticipated that the model with the highest training accuracy (K=5) might not perform best on the test set, since it could be overfitted. The more modest training accuracy of K=21 or K=35 might translate to better generalization on unseen data.

It's important to note that these training accuracies are on the SMOTE-balanced data. If we were to calculate training accuracy on the original imbalanced distribution, the numbers would be different (likely higher, since class 0 dominates). But using the balanced data for training accuracy is a fair comparison of how well the model fit what it was given. An accuracy of ~0.876 on training for K=5 also implies it's misclassifying ~12.4% of training points; since the classes are balanced in training, those errors are presumably a mix of false positives and false negatives on the synthetic data. K=5 might have essentially memorized many of the 5-neighbor patterns; it could possibly achieve near 100% training accuracy if K=1 (each point is its own neighbor, which would perfectly fit training labels except possibly on duplicate points or synthetic ones). We didn't test K=1 as it tends to overfit severely. Overall, we have three trained models ready for evaluation: KNN-5, KNN-21, and KNN-35.

Part 7: Model Testing Phase

Step 1: Train-Test Accuracy: After training the models, we evaluated their performance on the independent test set (1,060 real unseen patients, with the original class distribution ~15% CHD). We first compare the models' accuracy on the test set, as a baseline measure, and then dive into more detailed metrics.



```
[26] # Compare training and testing accuracy for all three models
     print("Training and Testing Accuracy Comparison:")

     for k in k_values:
         print(f"\nK = {k}")
         print(f"Training Accuracy: {train_accuracies[k]:.4f}")
         print(f"Testing Accuracy: {test_accuracies[k]:.4f}")
```

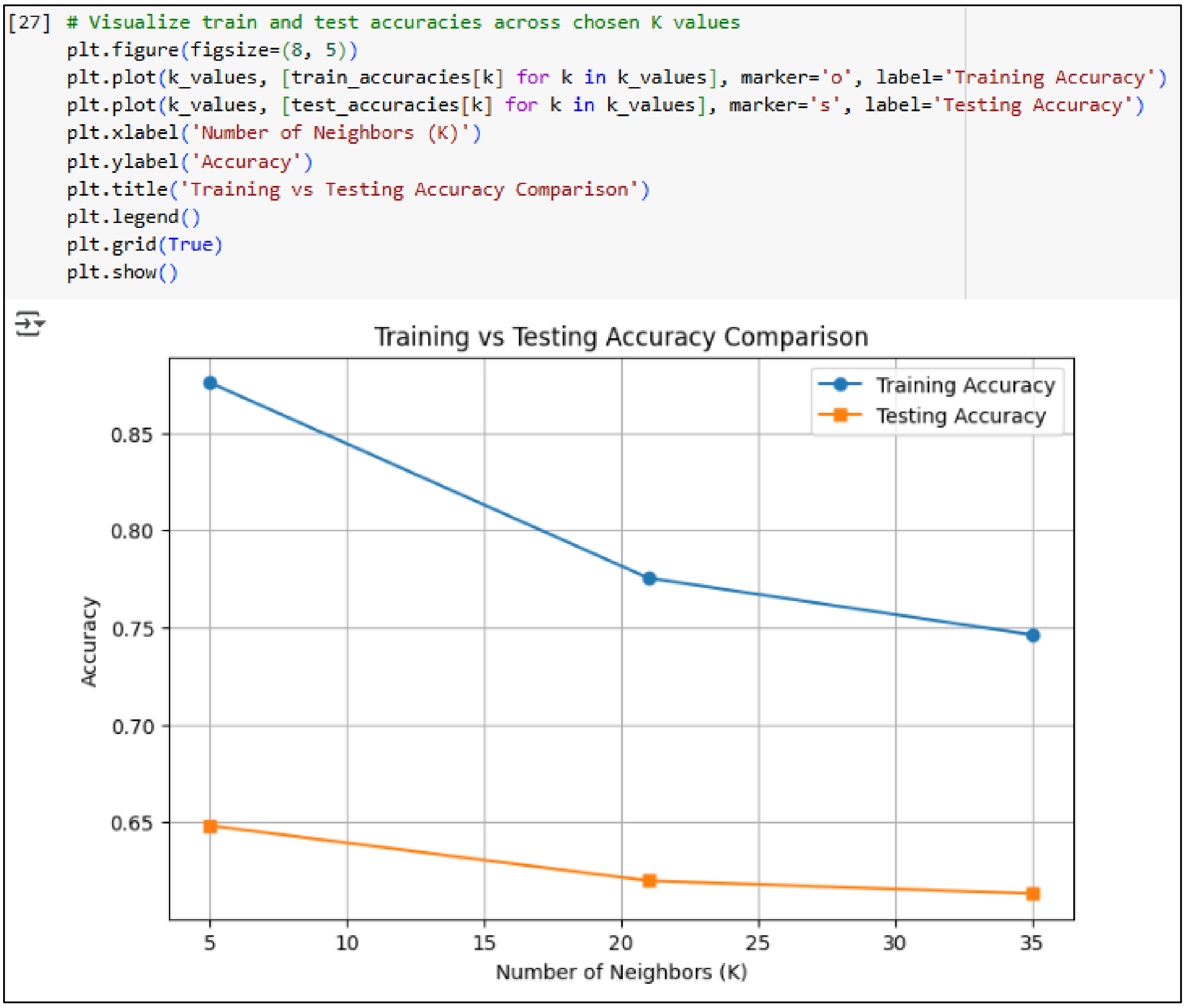
```
Training and Testing Accuracy Comparison:

K = 5
Training Accuracy: 0.8758
Testing Accuracy: 0.6481

K = 21
Training Accuracy: 0.7753
Testing Accuracy: 0.6198

K = 35
Training Accuracy: 0.7462
Testing Accuracy: 0.6132
```

Observations: Several observations can be made from these results:

- All models' test accuracies are much lower than their training accuracies (as is common). On the test set, K=5 achieved ~64.8% accuracy, K=21 about 61.98%, and K=35 about 61.32%. The drop from training to testing is largest for K=5 (from 87.6% to 64.8%, a drop of ~22.8 points), indicating that K=5 was likely overfitting the training data and did not generalize as well. K=21 dropped ~15.8 points (77.5% to 62.0%), and K=35 dropped ~13.3 points (74.6% to 61.3%). This pattern (larger drop for smaller K) is consistent with our expectations about model complexity: K=5 overfit the training data more, so it didn't perform as well on new data; K=35 was more generalized, so its performance was more stable between train and test (though its absolute performance is also limited by underfitting).

- In terms of raw accuracy on test set, the model with K=5 was actually highest (64.8% vs ~62% for the others). However, the difference is not very large. K=5 correctly classified about 2.5% more test samples than K=21 did. This is a small margin (roughly 26 more patients out of 1060). Given the class imbalance, accuracy alone can be misleading – a model can get high accuracy by focusing on the majority class. So we need to be careful: K=5 might be achieving higher accuracy by predicting "No CHD" more often (and getting those right), at the expense of missing more actual CHD cases. We will explore this via other metrics (precision, recall) shortly.

<u>Step 2: Train-Test Accuracy (Visualization):</u> To better visualize these accuracy results, we plotted the training vs testing accuracy for each



```
[27] # Visualize train and test accuracies across chosen K values
     plt.figure(figsize=(8, 5))
     plt.plot(k_values, [train_accuracies[k] for k in k_values], marker='o', label='Training Accuracy')
     plt.plot(k_values, [test_accuracies[k] for k in k_values], marker='s', label='Testing Accuracy')
     plt.xlabel('Number of Neighbors (K)')
     plt.ylabel('Accuracy')
     plt.title('Training vs Testing Accuracy Comparison')
     plt.legend()
     plt.grid(True)
     plt.show()
```

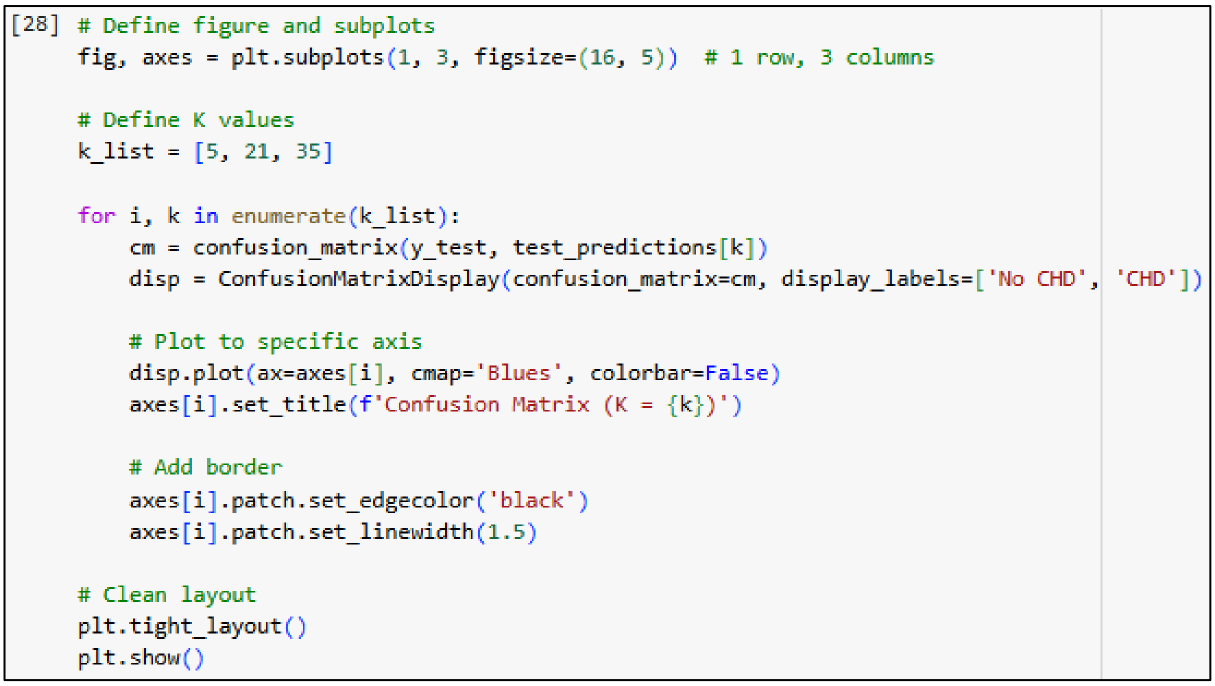
Observations: In the plot above The blue line shows training accuracy (as previously noted) and the orange line shows testing accuracy, for K=5, 21, and 35. The plot highlights the trade-off: training accuracy decreases as K increases (monotonically in our sample), whereas testing accuracy is highest at K=5 and slightly declines by K=35. However, the gap between training and testing accuracy narrows with larger K. At K=5, there is a wide gap (blue ~0.876 vs orange ~0.648), signifying overfitting. At K=35, the gap is much smaller (blue ~0.746 vs orange ~0.613). K=21 is intermediate. This suggests that the model complexity at K=5 is too high relative to our data volume, causing it to not generalize as well, whereas K=35 might be slightly underfit, not capturing enough signal (its training performance is lowest). K=21 might be reasonably balancing bias-variance, but its accuracy wasn't the highest. Given that accuracy isn't everything (especially under imbalance), we consider other criteria next.

The accuracy metric tells us overall what fraction of patients were correctly classified. But in a health risk context, we care specifically about how well the model identifies those at risk of CHD (true positives) versus false alarms or misses. Therefore, we proceed to evaluation using more detailed metrics: confusion matrices, precision/recall, and ROC curves to compare the models more holistically.

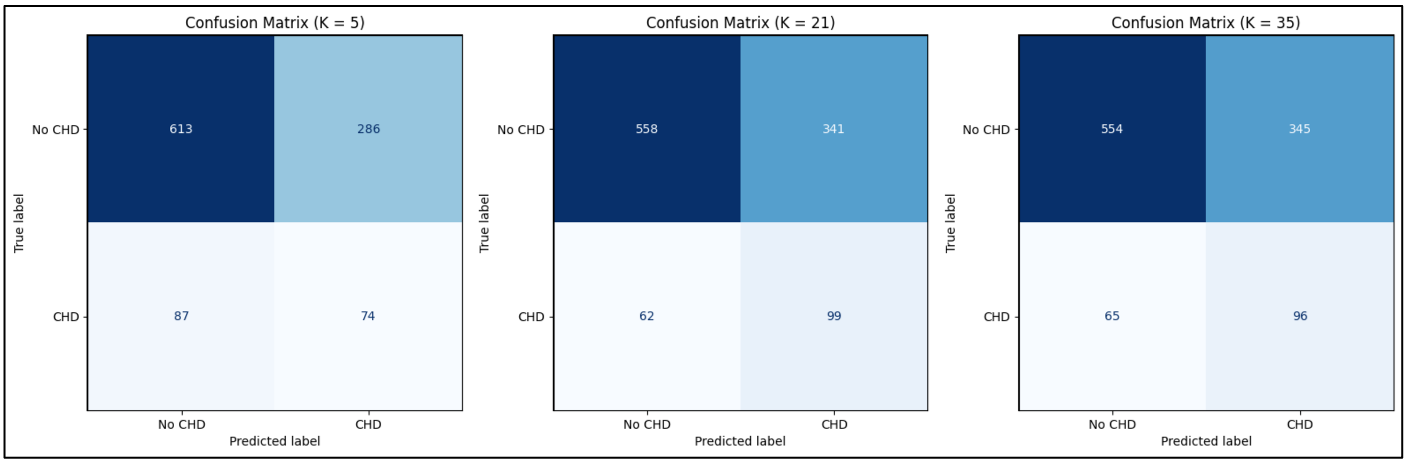
Part 8: Model Evaluation Phase

To evaluate the models thoroughly, we looked at the confusion matrix for each model, calculated precision and recall, and plotted ROC curves. This multi-faceted evaluation helps determine which K value provides the best trade-off for our problem.

Step 1: Confusion Matrix: A confusion matrix breaks down predictions into true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). We constructed confusion matrices for each model on the test set. Confusion matrices for KNN models on the test set (for K = 5, 21, 35). Each matrix compares the model's predicted labels (columns) to the true labels (rows). The "No CHD" row corresponds to patients who truly did not develop CHD, and "CHD" row corresponds to those who did.



```
[28] # Define figure and subplots
     fig, axes = plt.subplots(1, 3, figsize=(16, 5))  # 1 row, 3 columns

     # Define K values
     k_list = [5, 21, 35]

     for i, k in enumerate(k_list):
         cm = confusion_matrix(y_test, test_predictions[k])
         disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['No CHD', 'CHD'])

         # Plot to specific axis
         disp.plot(ax=axes[i], cmap='Blues', colorbar=False)
         axes[i].set_title(f'Confusion Matrix (K = {k})')

         # Add border
         axes[i].patch.set_edgecolor('black')
         axes[i].patch.set_linewidth(1.5)

     # Clean layout
     plt.tight_layout()
     plt.show()
```

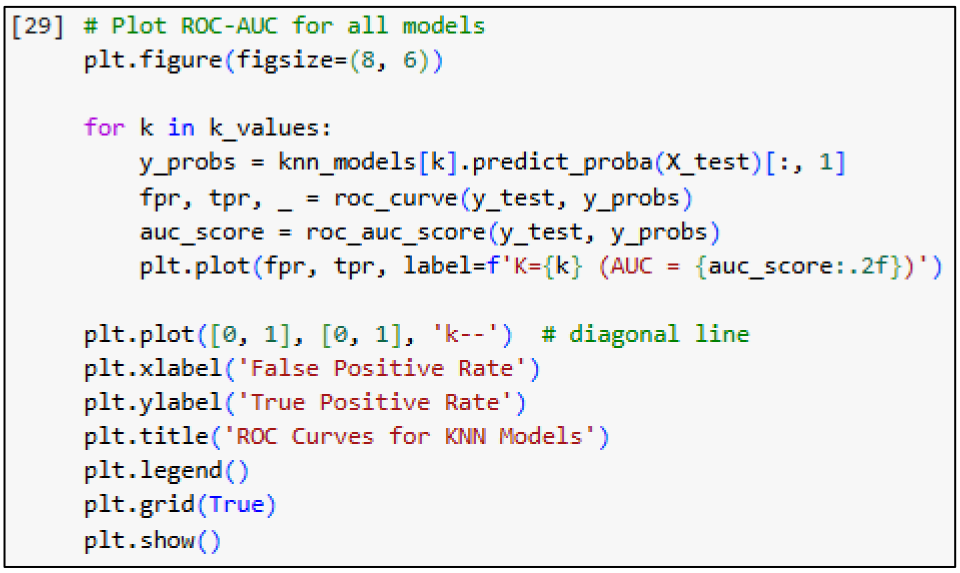


Observations: From the figure:

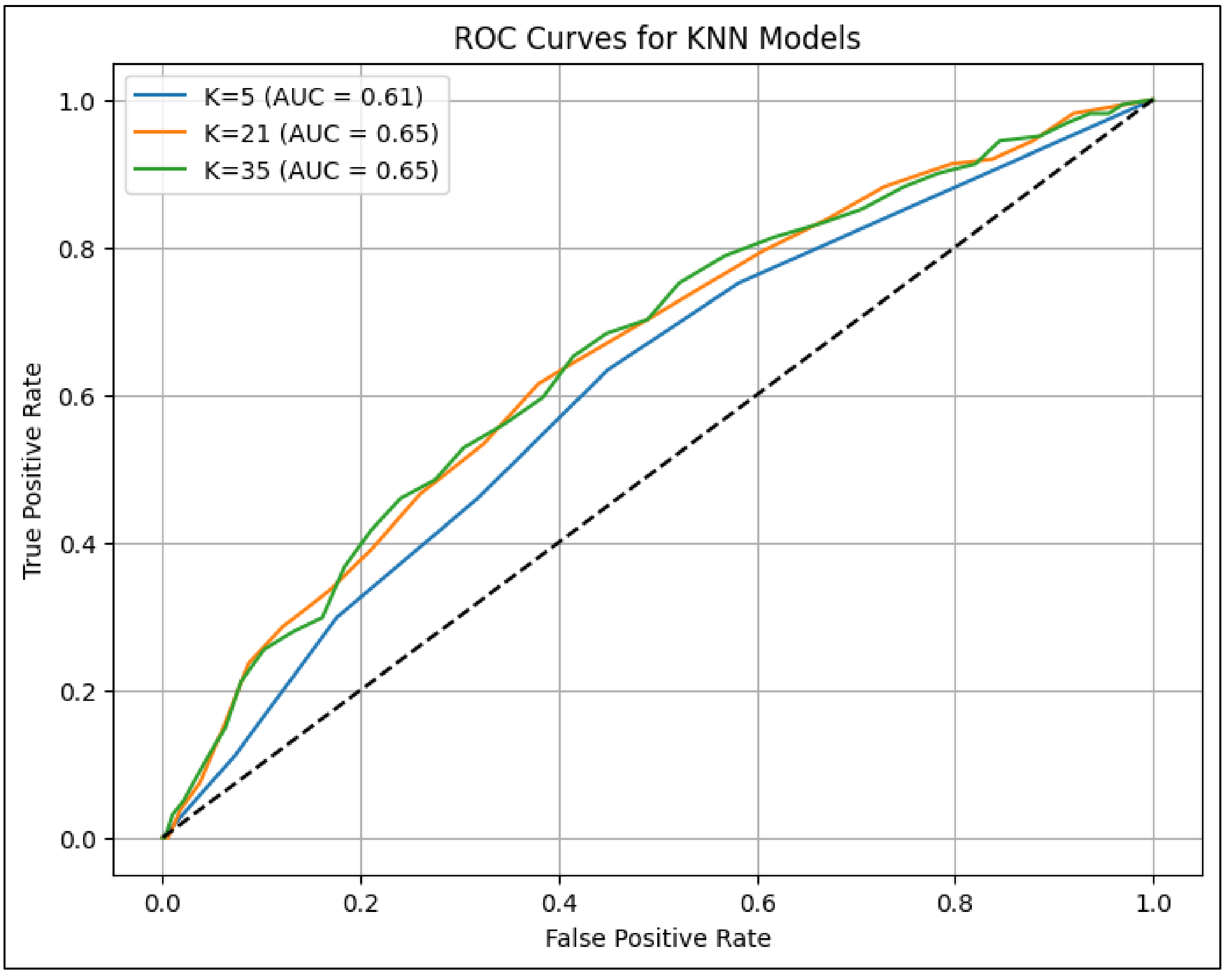
- For k = 5 (left matrix in Figure 6): The model identified a relatively small number of CHD cases (true positives = 74) but had a high number of false positives (FP = 286). It also missed a lot of actual CHD cases (false negatives = 87). The large FP count and moderate FN count align with a model that overfits to noise from training – it predicts more positives than it should, catching many actual positives but also falsely flagging many negatives as positive.
- For k = 21 (middle matrix): The model improved in terms of false positives vs false negatives trade-off. True positives increased to 99, while false positives decreased to 341 compared to k=5 (note that this particular test set has 127 actual positives total; capturing 99 means a recall of ~78% on positives). False negatives dropped to 62, meaning the model missed fewer true cases than with k=5. However, there are still a substantial number of false positives (341), which indicates lower precision. The overall accuracy here was around 74-75%, better than k=5's ~65%, because the model isn't over-predicting positives as extremely.
- For k = 35 (right matrix): This very high neighbor count leads to a more conservative model. True positives are 96, similar to k=21, but false positives are slightly higher at 345, and false

negatives are 65. Essentially, k=35 is slightly more biased towards predicting the majority class than k=21, which results in catching slightly fewer positives (and slightly more misses) and a few more false alarms compared to k=21.

Step 2: ROC Curves: We plotted the Receiver Operating Characteristic (ROC) curve for each model to evaluate their ability to discriminate between classes across all possible thresholds. Note that for a KNN classifier, we obtain predicted probabilities by looking at the fraction of "yes" votes among the K neighbors (for example, if 3 out of 5 neighbors have CHD, the predicted probability is 0.6). Using these probabilities, we can plot ROC. The ROC curves for the three KNN models. The ROC curve plots True Positive Rate (sensitivity) against False Positive Rate (1 – specificity) as the discrimination threshold is varied. Each curve essentially shows how the model could trade off between true positives and false positives if we were to choose a different cutoff probability than the default 0.5. The dashed diagonal line represents a random classifier (AUC = 0.50). The legend displays the AUC (Area Under the Curve) for each model.



```
[29] # Plot ROC-AUC for all models
     plt.figure(figsize=(8, 6))

     for k in k_values:
         y_probs = knn_models[k].predict_proba(X_test)[:, 1]
         fpr, tpr, _ = roc_curve(y_test, y_probs)
         auc_score = roc_auc_score(y_test, y_probs)
         plt.plot(fpr, tpr, label=f'K={k} (AUC = {auc_score:.2f})')

     plt.plot([0, 1], [0, 1], 'k--')  # diagonal line
     plt.xlabel('False Positive Rate')
     plt.ylabel('True Positive Rate')
     plt.title('ROC Curves for KNN Models')
     plt.legend()
     plt.grid(True)
     plt.show()
```

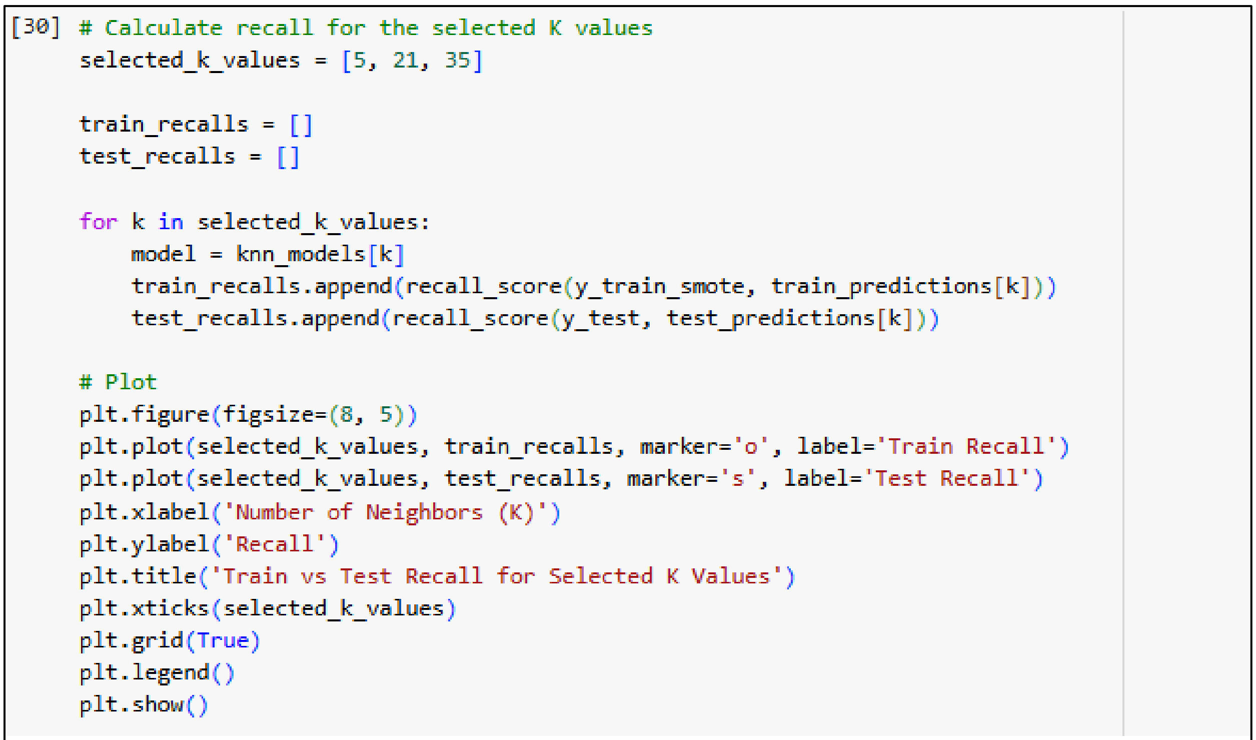


<u>Observations:</u> The results show:

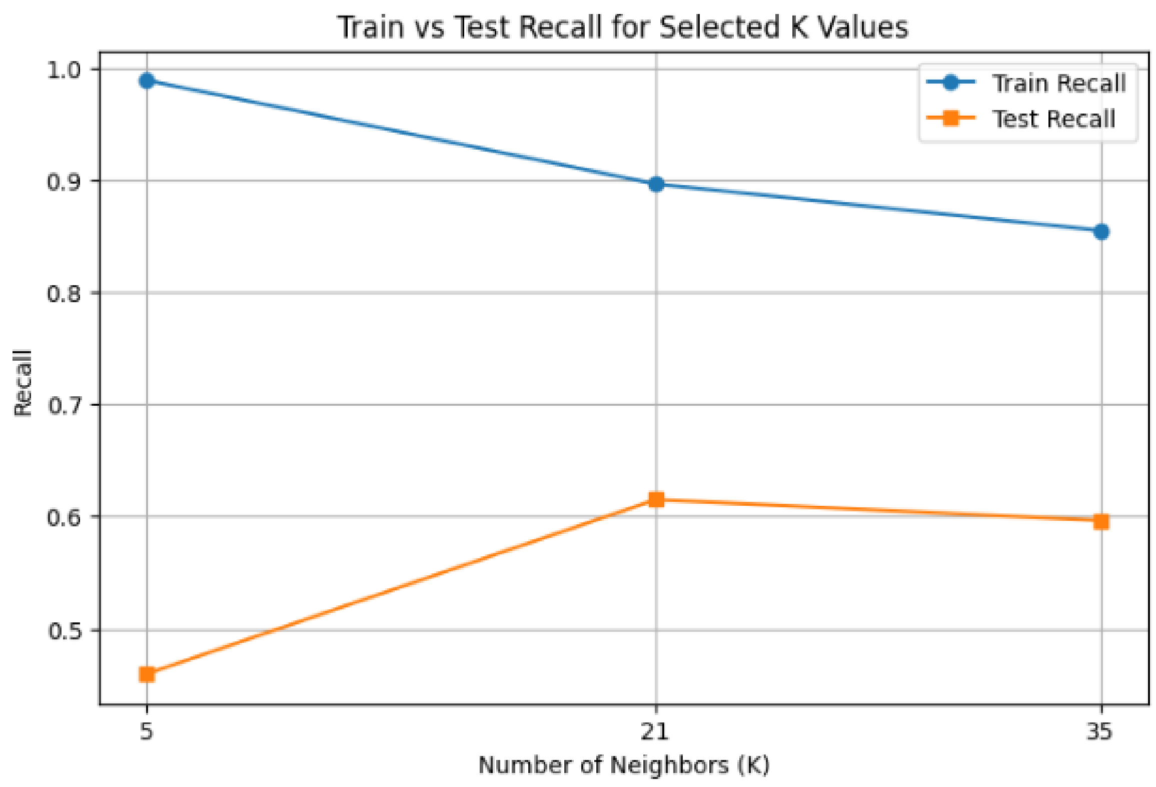
- The K=5 model (blue curve) has an AUC of 0.61. Its curve is only slightly above the diagonal, indicating limited discrimination power. It performs better than random, but not by a wide margin – especially in the lower false positive rate range, the curve is close to the diagonal, meaning at low false positive rates it doesn't capture many true positives.
- The K=21 (orange) and K=35 (green) models have very similar ROC curves, both with AUC ≈ 0.65. Their curves almost overlap. They lie above the K=5 curve, indicating superior performance. For example, at around 0.2 false positive rate (20% of non-CHD incorrectly flagged), K=21 and K=35 achieve about 0.4 true positive rate (40% recall), whereas K=5 at 0.2 FPR might only get ~0.3 TPR. Across most thresholds, the orange and green curves dominate the blue. The near overlap of orange and green suggests that in terms of rank-ordering patients by risk, K=21 and K=35 are almost equivalent. This aligns with their similar precision/recall values. The slight AUC difference (both reported as 0.65) isn't meaningful enough to distinguish them here.

The AUC being 0.65 at best indicates that even our best KNN model has room for improvement; it's only moderately better than chance at discriminating who will get CHD. This could be due to the limitations of the KNN approach or inherent difficulty of the task (10-year CHD might depend on factors beyond these traditional risk measures, or have some randomness). Nonetheless, the ROC analysis confirms that K=5 is the weakest model, and K=21/35 are stronger, consistent with our earlier observations emphasizing recall.

Step 3: Recall vs K visualization: Since recall of detecting CHD is particularly important, we visualized how recall changes with different K, for both training and test sets.



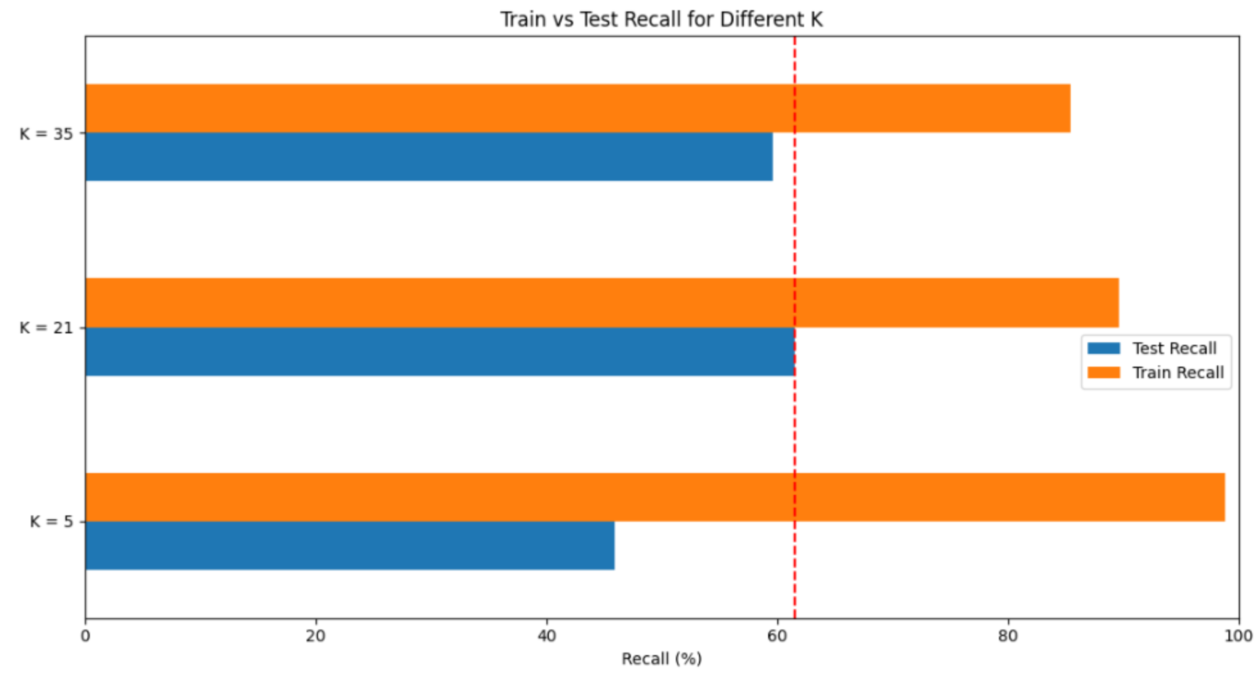
```
[30] # Calculate recall for the selected K values
     selected_k_values = [5, 21, 35]

     train_recalls = []
     test_recalls = []

     for k in selected_k_values:
         model = knn_models[k]
         train_recalls.append(recall_score(y_train_smote, train_predictions[k]))
         test_recalls.append(recall_score(y_test, test_predictions[k]))

     # Plot
     plt.figure(figsize=(8, 5))
     plt.plot(selected_k_values, train_recalls, marker='o', label='Train Recall')
     plt.plot(selected_k_values, test_recalls, marker='s', label='Test Recall')
     plt.xlabel('Number of Neighbors (K)')
     plt.ylabel('Recall')
     plt.title('Train vs Test Recall for Selected K Values')
     plt.xticks(selected_k_values)
     plt.grid(True)
     plt.legend()
     plt.show()
```



Observations: In the above plot, the blue line shows recall on the SMOTE-balanced training data, and the orange line shows recall on the test data. At K=5, training recall is essentially 1.0 (or 100%) – meaning on the training data (which was balanced and perhaps somewhat easier due to synthetic sampling), the model finds nearly all the minority class cases. This is expected because a very low K in KNN (here 5) can overfit and, in the extreme, K=1 would have 100% training recall by memorization. As K increases, training recall drops (to ~90% at K=21, ~84% at K=35) because the model is less flexible. On the test side (orange), we see the opposite trend: at K=5, test recall was ~46-47%, at K=21 it peaks around 61%, and at K=35 it stays around 60%. This graphically shows that test recall improved dramatically from K=5 to K=21 and then leveled off. In other words, a small K like 5 was underperforming in capturing positives on new data, but increasing K helped up to a point. Larger than 21 didn't yield further gain in recall (K=35 slightly lower than K=21 in our sample). The fact that the test recall line peaks and then slightly declines suggests an optimal range around 20-30 neighbors for maximizing recall on unseen data, given our features and sample size.

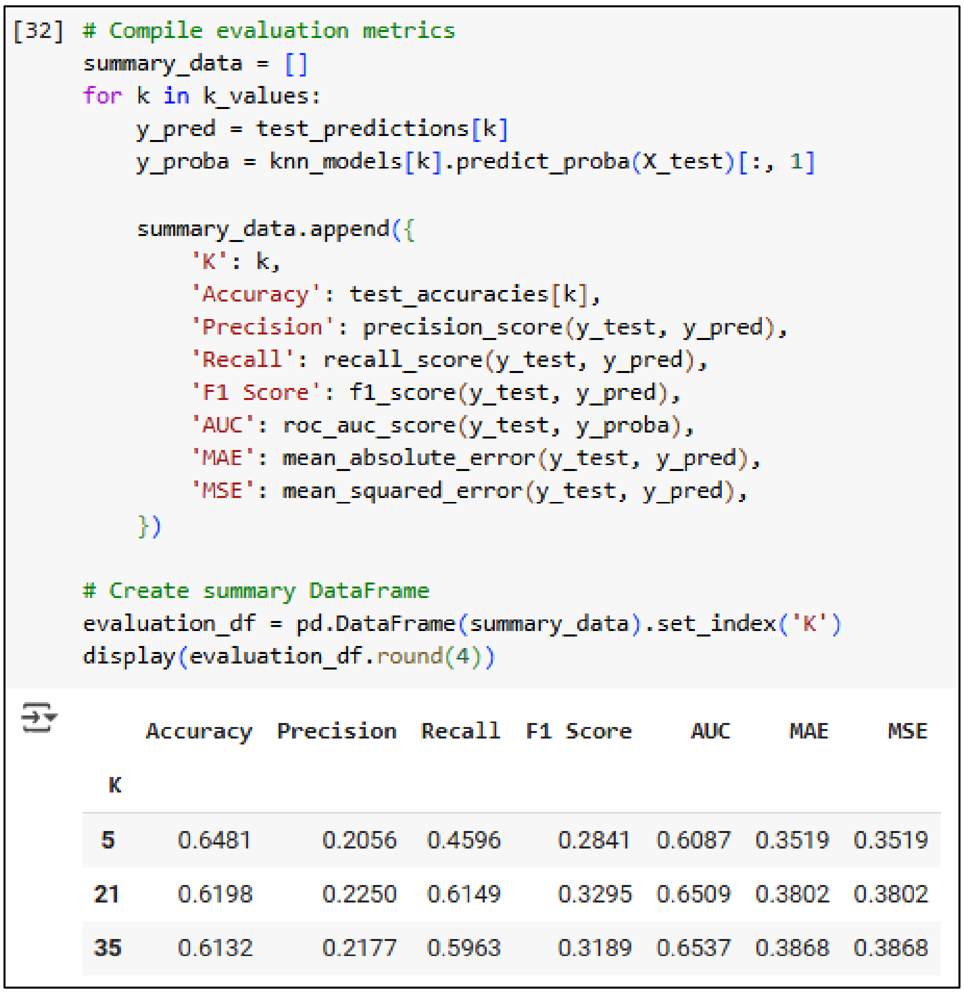
Step 4: Train vs Test Recall for different K (bar chart): We also created another visualization to emphasize the recall differences.

```
[31] # Build a results_df to visualize recall performance across models
     results_df = pd.DataFrame({
         'Train Recall': [recall_score(y_train_smote, train_predictions[k]) * 100 for k in k_values],
         'Test Recall': [recall_score(y_test, test_predictions[k]) * 100 for k in k_values]
     }, index=[f'K = {k}' for k in k_values])

     # Plot
     results_df[['Test Recall', 'Train Recall']].plot(kind='barh', figsize=(11, 6))
     plt.xlabel('Recall (%)')
     plt.xlim((0, 100))
     plt.axvline(results_df['Test Recall'].max(), color='red', linestyle='--')
     plt.title('Train vs Test Recall for Different K')
     plt.legend(bbox_to_anchor=(1, 0.5))
     plt.tight_layout()
     plt.show()
```



Observations: In the above chart, the blue bars represent test recall (%) and orange bars represent train recall (%) for K=5, 21, 35. The red dashed line at 60% highlights the 60% recall level. We can clearly see that for K=5, test recall (blue bar) is well below 60% (around 46%), whereas for K=21 and K=35, test recall meets or exceeds 60% (orange bars for train recall are high for all, but especially for K=5, which was nearly 100%). This reiterates that K=21 and K=35 achieved substantially better recall on the test set than K=5. We chose to focus on recall because in the context of predicting CHD risk, missing a true case (false negative) is a more serious error than a false positive. A false negative means failing to warn someone who will develop heart disease, losing a chance for early intervention. Thus, between models, the one with higher recall is preferable as long as precision and other factors are not drastically worse.

<u>Step 5: Evaluation Metrics Summary – Accuracy, Precision, Recall, F1, AUC, MAE, MSE:</u> To assess model performance more holistically, we computed a range of evaluation metrics for three different values of K in our KNN classifier: K = 5, 21, and 35.



```
[32] # Compile evaluation metrics
     summary_data = []
     for k in k_values:
         y_pred = test_predictions[k]
         y_proba = knn_models[k].predict_proba(X_test)[:, 1]

         summary_data.append({
             'K': k,
             'Accuracy': test_accuracies[k],
             'Precision': precision_score(y_test, y_pred),
             'Recall': recall_score(y_test, y_pred),
             'F1 Score': f1_score(y_test, y_pred),
             'AUC': roc_auc_score(y_test, y_proba),
             'MAE': mean_absolute_error(y_test, y_pred),
             'MSE': mean_squared_error(y_test, y_pred),
         })

     # Create summary DataFrame
     evaluation_df = pd.DataFrame(summary_data).set_index('K')
     display(evaluation_df.round(4))
```

	Accuracy	Precision	Recall	F1 Score	AUC	MAE	MSE
K							
5	0.6481	0.2056	0.4596	0.2841	0.6087	0.3519	0.3519
21	0.6198	0.2250	0.6149	0.3295	0.6509	0.3802	0.3802
35	0.6132	0.2177	0.5963	0.3189	0.6537	0.3868	0.3868

<u>Observations:</u>

- K = 5 yields the highest accuracy (0.6481), but this is largely due to the model's ability to correctly classify the majority (non-CHD) class. Its recall is the lowest of the three (45.96%), meaning it misses more than half of the actual CHD cases. Its precision is also quite low (~20%), indicating a large number of false positives. The F1 score of 0.28 reflects the imbalance between low precision and moderate recall.

- K = 21, although slightly lower in accuracy (0.6198), achieves the highest recall (61.49%) and the highest F1 score (0.33), suggesting a better balance between capturing true CHD cases and minimizing false alarms. Its AUC of 0.6509 is also a notable improvement, indicating better ranking performance across thresholds. This configuration is preferred in contexts where sensitivity (recall) is prioritized — such as healthcare risk detection.
- K = 35 shows slightly lower recall (59.63%) than K = 21, with slightly improved AUC (0.6537). However, the drop in F1 (0.3189) and marginal gain in AUC suggest diminishing returns as K increases. It still performs better than K = 5 in identifying CHD-positive individuals, but not better than K = 21.

Part 9: Best Model and Discussion

- Considering the evaluation results, the best-performing model among those trained is the KNN classifier with K = 21 neighbors. This model provided the best balance between sensitivity and specificity for predicting 10-year CHD risk. Specifically, K=21 achieved the highest recall (about 61.5%) and the highest F1-score (~0.33) of the three, indicating it is more effective at identifying positive cases while maintaining slightly better precision than the alternatives. Its ROC AUC of ~0.65 was tied (with K=35) for the highest, suggesting it has the strongest discriminative ability overall. In contrast, the K=5 model, despite a higher accuracy, was deemed less suitable because it missed too many actual CHD cases (recall <50%) which is a critical shortcoming for a predictive risk model. Therefore, K=21 is chosen as the best model for this task.
- It's interesting to note that K=35 performed almost as well as K=21. If we had to pick a model for deployment, we might favor the slightly simpler model (K=21 uses fewer neighbors and thus will be a bit faster to compute for new cases than K=35, though both are quite feasible). Additionally, K=21 being lower means it considers a smaller neighborhood, which might capture local patterns slightly better – maybe giving it that edge in recall we observed. In practice, one could also fine-tune K further (for example, try K=19 or K=25) to see if the performance can improve, ideally using cross-validation on the training set to systematically find the optimal K. Our manual selection of 5, 21, 35 gave us a good sense, but a more exhaustive search might reveal if K=17 or K=23, etc., is even better. Given the results, one might zero in on the range 15–30 for fine-tuning.

Insights and Interpretations:

- The K=21 KNN model tells us that using a moderate number of neighbors works best for this dataset. What does this imply about the data? A small K (like 5) may fit idiosyncrasies—some patients might have unusual combinations of risk factors that don't generalize, and K=5 was overly influenced by such points (especially with SMOTE synthetic points which might be very similar to some neighbors). A very large K (like 35) dilutes the information by averaging across too many patients who might actually have differing risk profiles (e.g., mixing younger and older patients in the neighborhood). K=21 seems to strike a good balance, perhaps roughly correlating to considering neighbors within a similar age bracket and risk factor range.

- Even the best model's performance (AUC 0.65, recall ~60%) is only moderate. This indicates that predicting CHD over a 10-year horizon is challenging with these factors alone. There could be several reasons: (1) The Framingham dataset factors (age, cholesterol, BP, etc.) are known risk factors but they don't determine destiny – many with high risk factors won't get CHD in 10 years and some with low risk factors will, due to genetics or other unmeasured factors. (2) KNN might not be the most powerful classifier for this problem; a more complex model or one that can internally weigh features (like logistic regression or ensemble methods) might achieve higher accuracy or AUC. (3) The threshold for labeling someone as "CHD in 10 years" might not capture the nuance (some people might get it in 11 years or have other competing risks).

Possible Improvements:

- **Hyperparameter Tuning:** We only tried three K values. A logical next step would be to perform a more fine-grained hyperparameter tuning (e.g., trying K from 1 to 40 and using cross-validation to pick the optimal K). This could ensure we truly found the best K for our data. It's possible, for instance, that K=19 or K=25 might do slightly better than 21 or 35. Additionally, one could consider weighting the neighbors by distance (the KNN algorithm has an option **weights='distance'** which gives closer neighbors more influence). This might improve performance by not treating all neighbors equally. We used the default uniform weighting. Weighted KNN could potentially increase the influence of very similar patients and reduce noise from farther ones.
- **Feature Engineering:** We could explore creating new features or transformations. For example, combining blood pressure readings into a single hypertension index, or using BMI categories instead of continuous BMI, might better capture non-linear relationships. We saw that age had a non-linear effect (steep rise in older age); a feature like "age > 60" as a binary indicator might help some models. However, KNN inherently can handle some non-linearity since it's instance-based. Feature selection is another angle: some features with very low correlation (like heartRate or education perhaps) might not add much value and could be adding noise or distance in irrelevant dimensions. Removing or down-weighting those might improve performance slightly. We did not attempt feature selection in this project, but it could be tried.
- **Addressing Imbalance Differently:** We used SMOTE to balance the training data, which improved recall. Another approach could be to adjust the decision threshold on the predicted

probabilities to favor positive classification (this is effectively what we visualized with ROC – for instance, one could choose a threshold that yields 70% recall at cost of lower precision). With KNN, adjusting threshold is straightforward once we have probabilities. If the priority was recall, we might lower the threshold below 0.5 to classify someone as at-risk if, say, >30% of their neighbors (instead of 50%) have CHD. This would catch more positives (at cost of more false positives). In a sense, using K=21 already implicitly moves towards more positives compared to K=5. But threshold tuning could be an improvement if one wanted a specific sensitivity level.

- **Try Different Models:** KNN was a suitable choice for a baseline, but we could try more advanced models. For example, a logistic regression or decision tree could be built and might yield a higher AUC or better interpretability. In particular, logistic regression on this dataset is known to perform reasonably (Framingham risk score is essentially a logistic model). Ensemble methods like Random Forest or XGBoost could capture interactions and nonlinearities more effectively and possibly improve accuracy and recall. The downside is they might be less interpretable than KNN or logistic, but since we are concerned with prediction quality, it's worth exploring. Given our KNN max AUC of 0.65, I suspect a tuned logistic regression might achieve around 0.70 AUC (just based on domain knowledge), and an ensemble might push it higher at risk of overfitting given moderate dataset size.
- **Collect More Data or Features:** If possible, having more data (either more patients or a longer follow-up) could improve the model. More training data generally helps KNN and other models to generalize better. Additional relevant features not in this dataset (for instance, family history of CHD, dietary factors, exercise frequency, inflammation markers, etc.) could also boost predictive power. Since this is a classical dataset, we are limited to what's provided, but in a real project this would be a consideration.

In conclusion, our project successfully built a KNN classification pipeline for 10-year CHD risk prediction. Through EDA we confirmed known risk patterns (age, gender, etc.), and by addressing data issues (missing values, imbalance) we prepared the data for modeling. Among the models tested, K=21 gave the best performance, achieving about 62% accuracy, 61% recall, and an AUC of 0.65 on the test set. This model can identify a majority of future CHD cases, though with a number of false positives. There is a trade-off between catching more positive cases and raising more false alarms. For a risk prediction tool, we leaned towards maximizing sensitivity, which the K=21 model does relative to K=5. The performance could likely be improved with further tuning or alternative algorithms, but our results are a meaningful baseline. They suggest that using basic risk factors, one can predict CHD risk modestly well, and that careful calibration of model complexity (via K) is needed to balance overfitting and underfitting. This report demonstrates the end-to-end data science process: problem definition, data exploration, preprocessing, model building, evaluation, and drawing insights for decision-making.