AI-Powered Heart Disease Risk Assessment App

Aktham Almomani
Shiley-Marcos School of Engineering - University of San Diego
Master of Science in Applied Artificial Intelligence

Author Note

This project was part of the AI-Powered Heart Disease Assessment App initiative, aimed at providing users with tailored risk scores and actionable recommendations to mitigate their heart disease risk. The app leverages advanced AI and machine learning models to deliver accurate assessments based on user input.

Correspondence concerning this project should be addressed to Dr. Dallin Munger Department of Applied Data Science, University of San Diego, 5998 Alcala Park, San Diego, CA 92110, United States. Email: dallinmunger@sandiego.edu.

Table of Contents

1. Introduction

The AI-Powered Heart Disease Risk Assessment App is a comprehensive tool designed to empower individuals with personalized insights into their cardiovascular health. While the app can evaluate multiple health indicators, it primarily focuses on predicting the risk of heart disease. This includes key factors such as:

- Age
- Gender
- BMI
- Physical activity levels
- Smoking status
- Medical history (e.g., previous heart attacks, strokes, diabetes)

By providing a thorough analysis of users' heart health status, the app aims to help users understand their risk and take proactive steps to maintain a healthy heart.

2. Objective

The AI-Powered Heart Disease Risk Assessment App aims to provide users with tailored risk scores and actionable recommendations to help them mitigate their risk of heart disease. Through easy-to-understand assessments and preventive measures, all powered by advanced AI and modeling techniques, the app makes safeguarding your cardiovascular health accessible and straightforward.

3. Dataset

The Behavioral Risk Factor Surveillance System (BRFSS) is the nation's premier system of health-related telephone surveys that collect state data about U.S. residents regarding their health-related risk behaviors, chronic health conditions, and use of preventive services. Established in 1984 with 15 states, BRFSS now collects data in all 50 states as well as the District of Columbia and three U.S. territories. CDC BRFSS completes more than 400,000 adult interviews each year, making it the largest continuously conducted health survey system in the world.

The dataset was sourced from Kaggle (Behavioral Risk Factor Surveillance System (BRFSS) 2022) and it was originally downloaded from the CDC BRFSS 2022 website.

To get more understanding regarding the dataset, please go to the data_directory folder.

4. Data Wrangling and pre-processing

4.1 Introduction

In this stage, I have undertaken a series of data wrangling steps to prepare our dataset for analysis. Data wrangling is a crucial step in the data science process, involving the transformation and mapping of raw data into a more usable format. Here's a summary of the key steps taken in this notebook:

- Dealing with Missing Data: Identified and imputed missing values in critical columns, such as the gender column, ensuring the dataset's completeness.

- Data Mapping: Transformed categorical variables into more meaningful representations, making the data easier to analyze and interpret.
- Data Cleaning: Removed or corrected inconsistent and erroneous entries to improve data quality.
- Feature Engineering: Created new features that may enhance the predictive power of our models. These steps are essential for building a reliable and robust model for heart disease prediction.

4.2 Extracting descriptive column Names for the dataset

The Behavioral Risk Factor Surveillance System (BRFSS) dataset available on Kaggle, found here, contains a wealth of information collected through surveys. However, the column names in the dataset are represented by short labels or codes (e.g., _STATE, FMONTH, IDATE), which can be difficult to interpret without additional context.

To ensure we fully understand what each column in the dataset represents, it is crucial to replace these short codes with their corresponding descriptive names. These descriptive names provide clear insights into the type of data each column holds, making the dataset easier to understand and analyze.

Process Overview:

- Identify the Source for Descriptive Names: The descriptive names corresponding to these short labels are typically documented in the codebook in HTML or metadata provided by the data collection authority. In this case, the descriptive names are found in an HTML document provided by the BRFSS.

- Parse the HTML Document: Using web scraping techniques, such as BeautifulSoup in Python, we can parse the HTML document to extract the relevant information. Specifically, we look for tables or sections in the HTML that list the short labels alongside their descriptive names.
- Match and Replace: We create a mapping of short labels to their descriptive names. This mapping is then applied to our dataset to replace the short labels with more meaningful descriptive names.
- Save the Enhanced Dataset: The dataset with descriptive column names is saved for subsequent analysis, ensuring that all users can easily interpret the columns.
- More details in this process and the custom python code can be found **here**

4.3 Correcting dataset column names

To replace the SAS Variable Names in your dataset with the corresponding labels (where spaces in the labels are replaced with underscores), you can follow these steps:

- Create a mapping from the SAS Variable Names to the modified labels.
- Use this mapping to rename the columns in your dataset.

4.4 Selecting Heart Disease related features

After several days of research and analysis of the dataset's features, we have identified the following key features for heart disease assessment:

- Target Variable (Dependent Variable):
    - Heart_disease: "Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease"

- Demographics:
    - Gender: Are_you_male_or_female
    - Race: Computed_race_groups_used_for_internet_prevalence_tables
    - Age: Imputed_Age_value_collapsed_above_80
- Medical History:
    - General_Health
    - Have_Personal_Health_Care_Provider
    - Could_Not_Afford_To_See_Doctor
    - Length_of_time_since_last_routine_checkup
    - Ever_Diagnosed_with_Heart_Attack
    - Ever_Diagnosed_with_a_Stroke
    - Ever_told_you_had_a_depressive_disorder
    - Ever_told_you_have_kidney_disease
    - Ever_told_you_had_diabetes
    - Reported_Weight_in_Pounds
    - Reported_Height_in_Feet_and_Inches
    - Computed_body_mass_index_categories
    - Difficulty_Walking_or_Climbing_Stairs
    - Computed_Physical_Health_Status
    - Computed_Mental_Health_Status
    - Computed_Asthma_Status
- Life Style:
    - Leisure_Time_Physical_Activity_Calculated_Variable

- Smoked_at_Least_100_Cigarettes
- Computed_Smoking_Status
- Binge_Drinking_Calculated_Variable
- Computed_number_of_drinks_of_alcohol_beverages_per_week
- Exercise_in_Past_30_Days
- How_Much_Time_Do_You_Sleep

4.5 Imputing Missing Data, Transforming Columns and Features Engineering

In this step, we address missing data, map categorical values, and rename columns for improved data quality and clarity. The key actions taken are as follows:

- Replace Specific Values with NaN: Identify and replace erroneous or placeholder values with NaN to standardize missing data representation.
- Calculate Value Distribution: Determine the distribution of existing values to understand the data's baseline state.
- Impute Missing Values: Use a function to impute missing values based on the calculated distribution, ensuring the data remains representative of its original characteristics.
- Map Categorical Values: Apply a mapping dictionary to convert numeric codes into meaningful categorical labels.
- Rename Columns: Update column names to reflect their contents accurately and improve dataset readability.
- Feature Engineering: Created new features that may enhance the predictive power of our models. These steps are essential for building a reliable and robust model for heart disease prediction.

4.6 Distribution-Based Imputation

To deal with missing data in this project, we'll be using Distribution-Based Imputation:

- Introduction:
  - Distribution-Based: The imputation process relies on the existing distribution of the categories in the dataset.
  - Imputation: The act of filling in missing values.
- Why This Method Works:
  - Preserves Original Distribution: By using the observed proportions to guide the imputation, the method maintains the original distribution of gender categories.
  - Random Imputation: Randomly selecting values based on the existing distribution prevents systematic biases that could arise from deterministic imputation methods.
  - Scalability: This approach can be easily scaled to larger datasets and applied to other categorical variables with missing values.
- Advantages:
  - Bias Minimization: Ensures that the imputed values do not skew the dataset in favor of any particular category.
  - Simplicity: The method is straightforward to implement and understand.
  - Flexibility: Can be adapted to any categorical variable with missing values.

This method is particularly useful in scenarios where preserving the natural distribution of data is crucial for subsequent analysis or modeling tasks (Saar-Tsechansky & Provost, 2007).

5. Exploratory Data Analysis

5.1 Introduction

This stage serves as a critical step in our data science workflow, aimed at uncovering insights and patterns within our dataset that will guide our predictive modeling efforts.

Here, we will:

- Validate the Dataset: Ensure the data is clean, consistent, and ready for analysis.
- Explore Feature Distributions: Analyze the distribution of various features in relation to heart disease.
- Convert Categorical Data: Transform categorical features into numeric format using CatBoost encoding for better analysis and modeling.
- Analyze Correlations: Examine both linear and non-linear relationships between features and the target variable (heart disease) using Pearson correlation and mutual information.
- Feature Selection: Identify and select key features that have the most predictive power for heart disease.

These steps will help us understand the data better, reveal important relationships, and prepare the data for building robust predictive models.

5.2 Objectives

- To get familiar with the features in our final DataFrame.
- Generally, understand the core characteristics of our cleaned DataFrame.
- Explore the data relationships of all the features and understand how the features compare to the response variable.

- Let's be creative and think about interesting figures and all the plots that can be created to help deepen our understanding of the data.

5.3 Final selection of the features

After several days of research and analysis of the dataset's features, we have identified the following key features for heart disease assessment:

- Target Variable (Dependent Variable):
  - Heart_disease: "Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease"
- Demographics:
  - Gender
  - Race
  - Age_category
- Medical History:
  - General_Health
  - Have_Personal_Health_Care_Provider
  - Could_Not_Afford_To_See_Doctor
  - Length_of_time_since_last_routine_checkup
  - Ever_Diagnosed_with_Heart_Attack
  - Ever_Diagnosed_with_a_Stroke
  - Ever_told_you_had_a_depressive_disorder
  - Ever_told_you_have_kidney_disease
  - Ever_told_you_had_diabetes

    - BMI
    - Difficulty_Walking_or_Climbing_Stairs
    - Physical_Health_Status
    - Mental_Health_Status
    - Asthma_Status
- Life Style:
    - Smoking_status
    - Binge_Drinking_status
    - Drinks_category
    - Exercise_in_Past_30_Days
    - Sleep_category

5.4 Converting features data type

In pandas, the object data type is used for text or mixed data. When a column contains categorical data, it's often beneficial to explicitly convert it to the category data type. Here are some reasons why:

Benefits of Converting to Categorical Type:

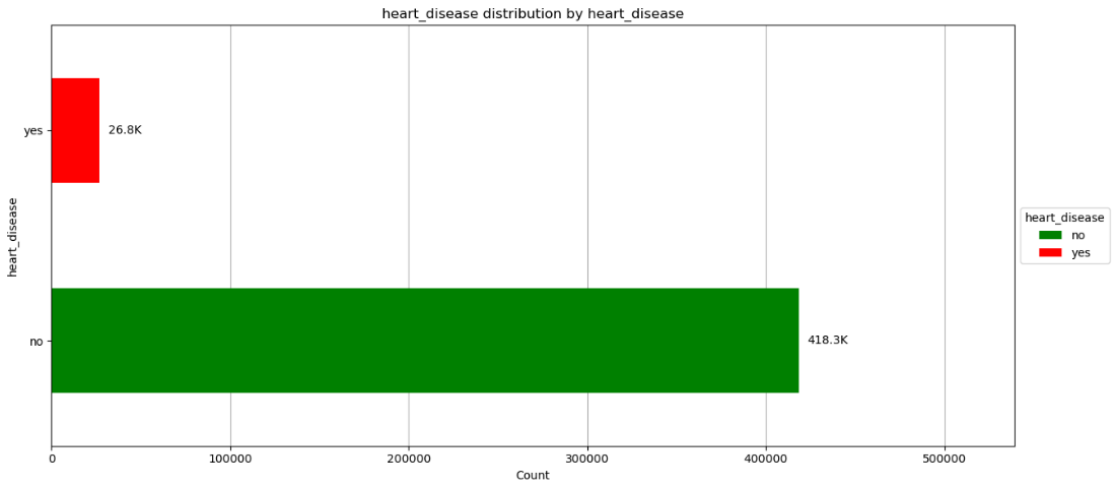
- Memory Efficiency: Categorical data types are more memory efficient. Instead of storing each unique string separately, pandas stores the categories and uses integer codes to represent the values.
- Performance Improvement: Operations on categorical data can be faster since pandas can make use of the underlying integer codes.

- Explicit Semantics: Converting to category makes the data's categorical nature explicit, improving code readability and reducing the risk of treating categorical data as continuous (pandas.pydata.org, 2023).

5.5 Analyzing categorical feature distributions against a target variable

In data analysis, understanding the distribution of categorical features in relation to a target variable is crucial for gaining insights into the data. One effective way to achieve this is by using horizontal stacked bar charts. These visualizations allow us to see how different categories of a feature are distributed across the levels of the target variable, providing a clear view of relationships and patterns within the data.

Figure 1



According to Figure 1 heart disease distribution:

- There is a significant imbalance between the two categories.
- A large majority of individuals do not have heart disease 418.3K, while a much smaller number have heart disease 26.8K.

- This imbalance can be visually observed in the chart, where the green bar is substantially longer than the red bar.

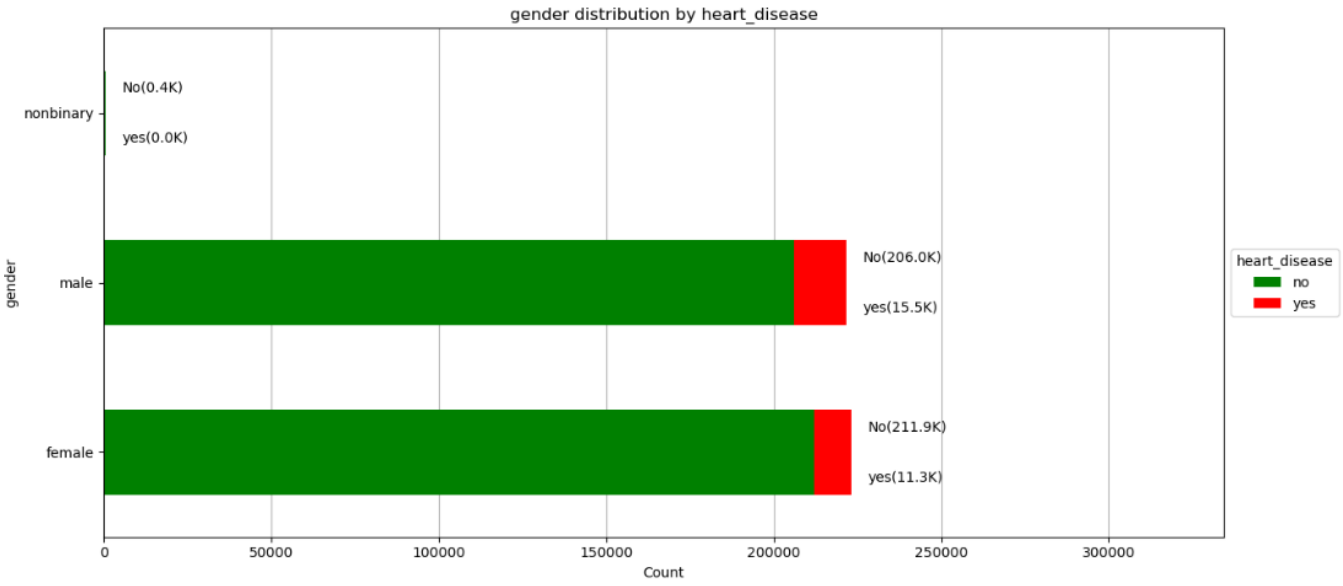
Imbalance Issue

- Model Bias: When training a classification model on this imbalanced dataset, the model might become biased towards predicting the majority class (No heart disease) more frequently because it is seen more often in the training data.
- Performance Metrics: Common performance metrics like accuracy can be misleading in imbalanced datasets. For instance, a model that always predicts "No heart disease" will have high accuracy because the majority class is well represented. However, this model would fail to correctly identify individuals with heart disease, which is critical for healthcare applications.
- Recall and Precision: Metrics such as recall (sensitivity) and precision are more informative in this context. Recall measures the ability to identify true positive cases (heart disease), while precision measures the accuracy of positive predictions. In an imbalanced dataset, a model might have low recall for the minority class (heart disease) even if it has high accuracy overall.

Strategy to Address Imbalance

The models from imabalnce-learn library library effectively handles class imbalance by using bootstrapped sampling to balance the dataset, ensuring robust classification of minority classes. It enhances model performance by focusing on underrepresented data, making it ideal for imbalanced datasets like heart disease prediction.

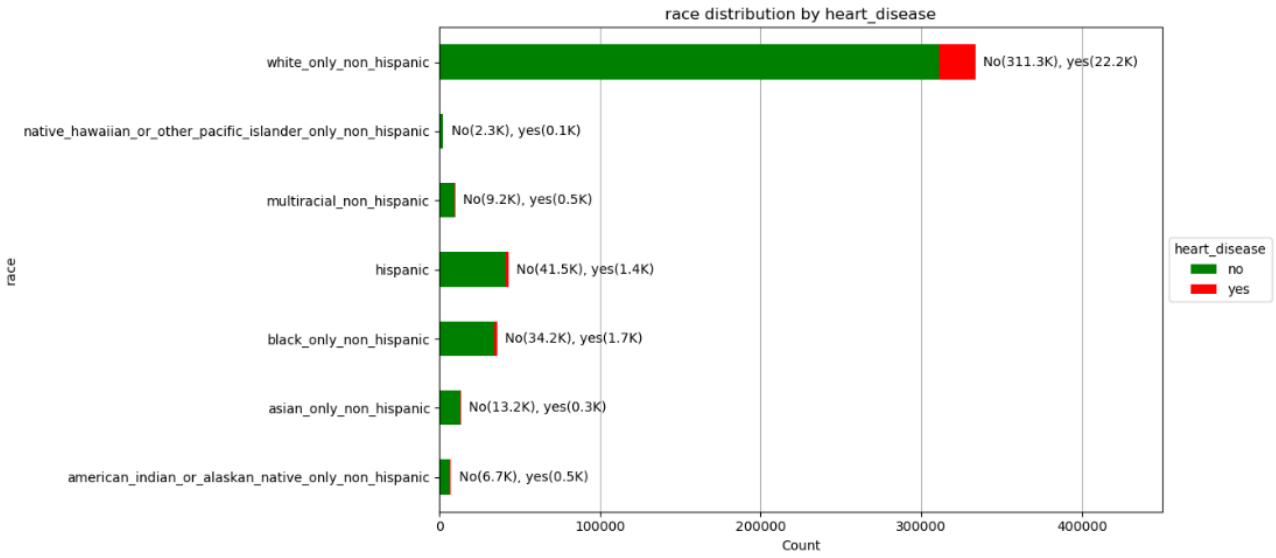
Figure 2



According to Figure 2 gender vs heart disease distribution:

- The majority of individuals with heart disease are male 15.5K, followed by female 11.3K.
- There are very few nonbinary individuals with heart disease 15 individuals.
- The significant difference in the number of heart disease cases among males and females compared to nonbinary individuals highlights a noticeable imbalance.

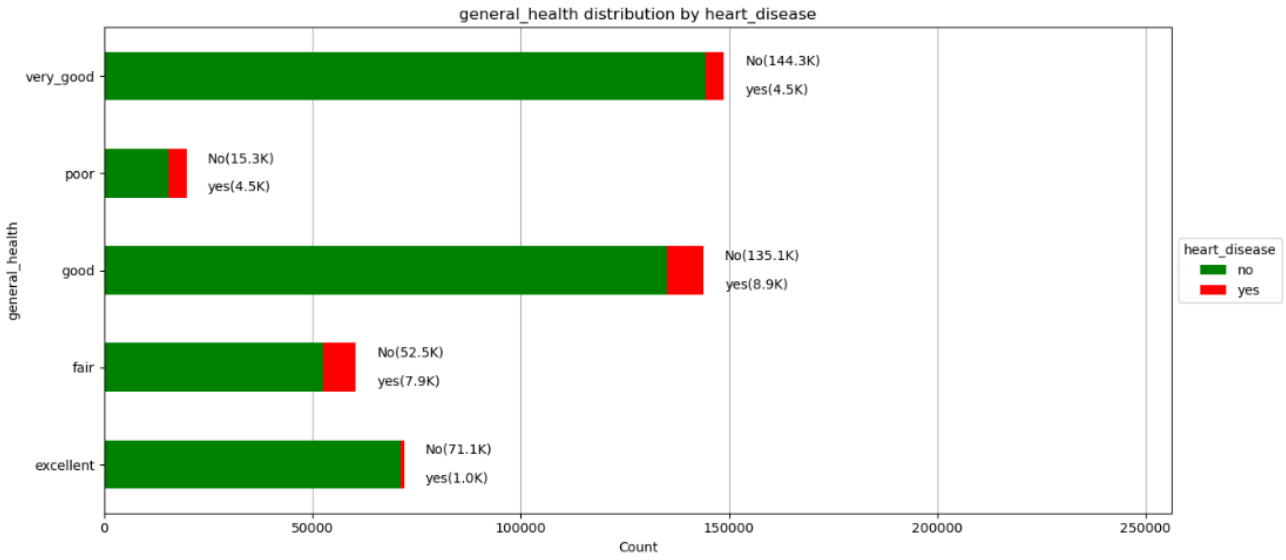
Figure 3



According to Figure 3 race vs heart disease distribution:

- The largest group with heart disease is "White Only, Non-Hispanic," with 22.2K individuals.
- Smaller groups, such as "Native Hawaiian or Other Pacific Islander Only, Non-Hispanic" and "Asian Only, Non-Hispanic," have very few individuals with heart disease (100 and 300 individuals, respectively).
- There is a notable imbalance in the number of heart disease cases across different racial categories, with significantly fewer cases in minority groups.

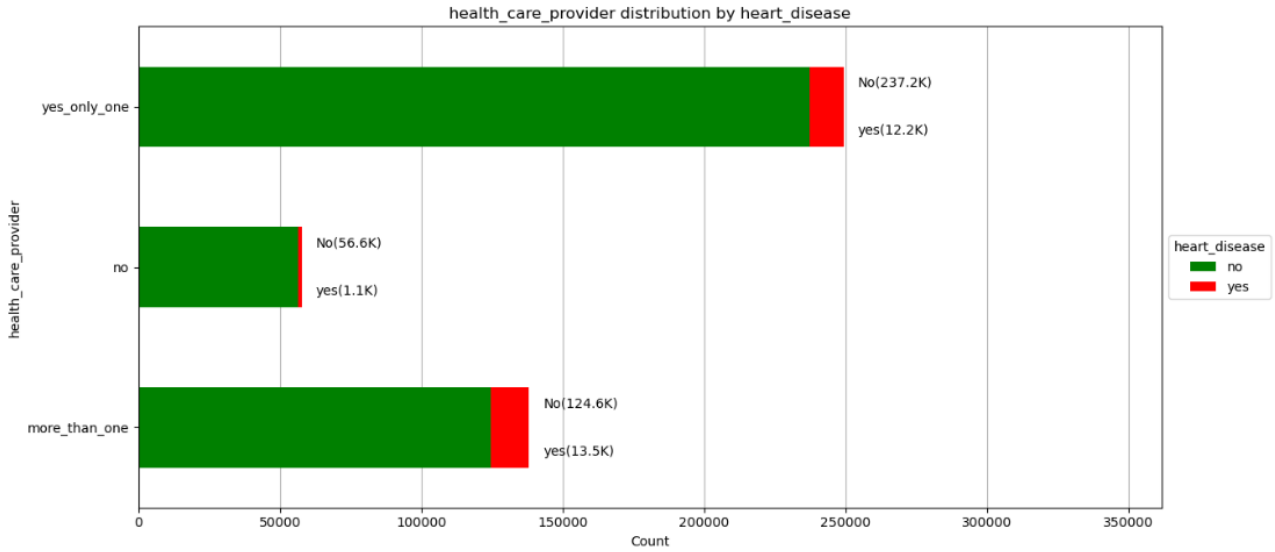
Figure 4



According to Figure 4 general health vs heart disease distribution:

- The highest number of individuals with heart disease falls into the "Good" health category 8.9K, followed by the "Fair" category 7.9K.
- Both "Very Good" and "Poor" health categories have the same number of individuals with heart disease 4.5K.
- The "Excellent" health category has the fewest individuals with heart disease 1.0K.
- There is a noticeable distribution of heart disease cases across different general health categories, with the highest incidence in individuals who self-report as having "Good" or "Fair" health.

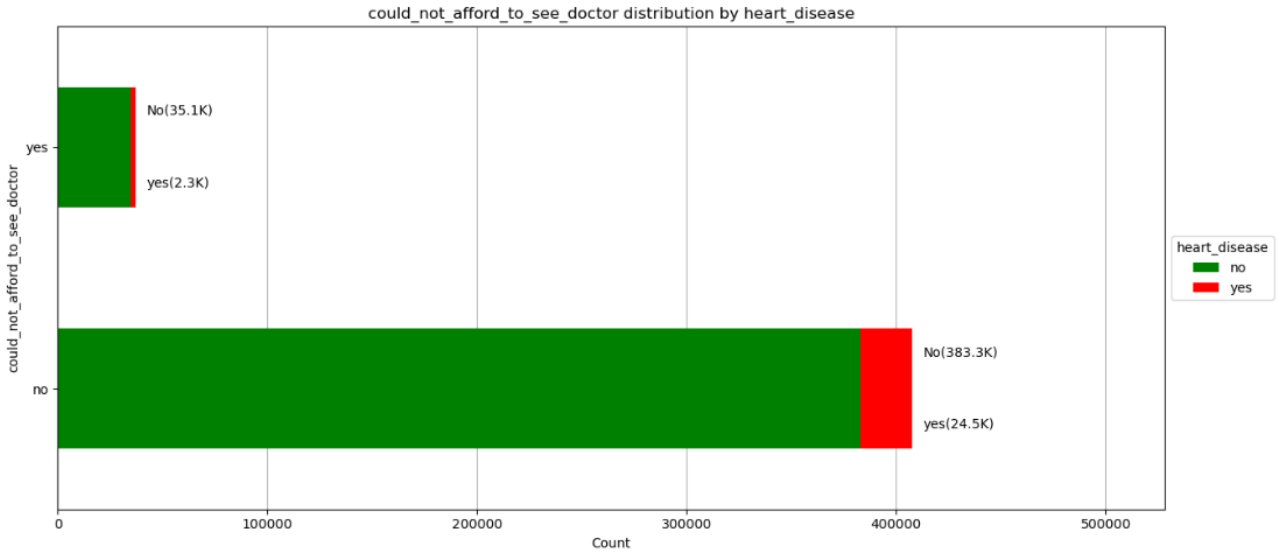
Figure 5



According to Figure 5 health care provider vs heart disease distribution:

- The highest number of individuals with heart disease is in the "More Than One" health care provider category 13.5K.
- The "Yes, Only One" category also has a significant number of individuals with heart disease 12.2K.
- The "No" health care provider category has the fewest individuals with heart disease 1.1K.
- This distribution suggests that individuals with multiple health care providers or at least one provider are more likely to have heart disease compared to those with no health care provider.

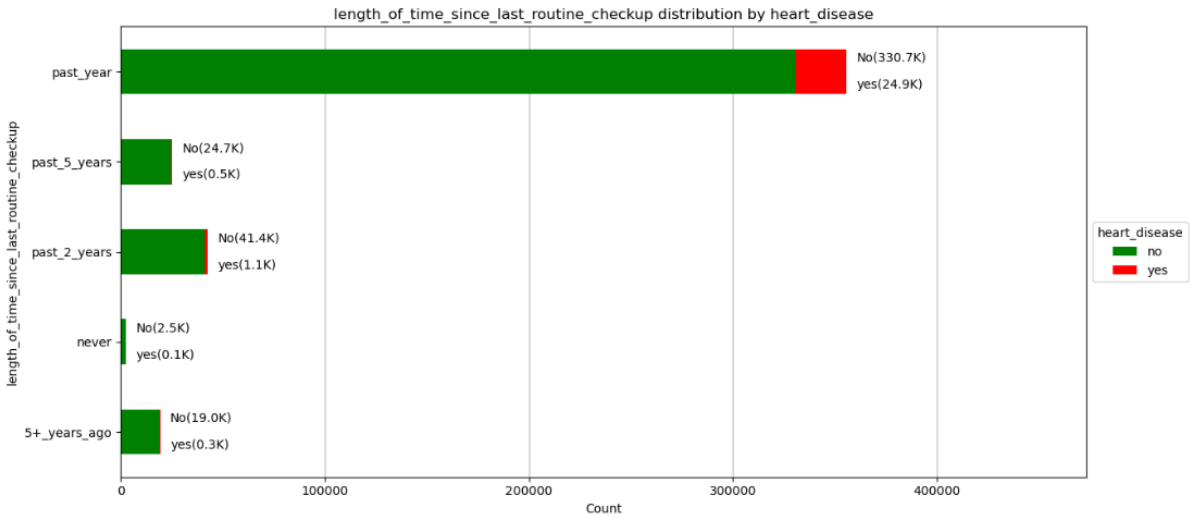
Figure 6



According to Figure 6 doctor availability vs heart disease distribution:

- The majority of individuals with heart disease fall into the category of those who could afford to see a doctor 24.5K.
- A smaller number of individuals with heart disease could not afford to see a doctor 2.3K.
- This distribution indicates that even among those with heart disease, most individuals could afford to see a doctor, suggesting access to healthcare does not completely mitigate the risk of heart disease.
- However, the presence of heart disease in individuals who could not afford to see a doctor highlights a potential issue with access to preventive care or treatment.

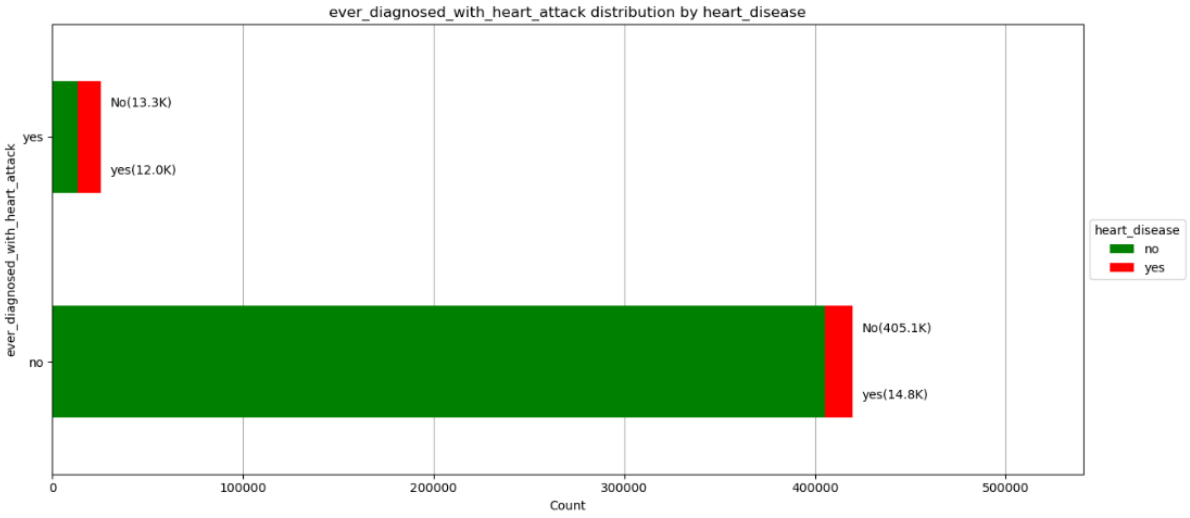
Figure 7



According to Figure 7 routine checkup vs heart disease distribution:

- The majority of individuals with heart disease had a routine checkup within the past year 24.9K. This indicates that individuals with heart disease are more likely to have had recent medical attention.
- There are significantly fewer individuals with heart disease who had a routine checkup in the past 2 years 1.1K and past 5 years 0.5K.
- A very small number of individuals with heart disease reported never having a routine checkup 0.1K or having their last checkup more than 5 years ago 0.3K.
- This distribution suggests that even those with recent medical checkups are at risk of heart disease, highlighting the importance of regular monitoring and early detection. However, individuals with infrequent or no checkups are less likely to be diagnosed, possibly due to lower health awareness or access to healthcare.

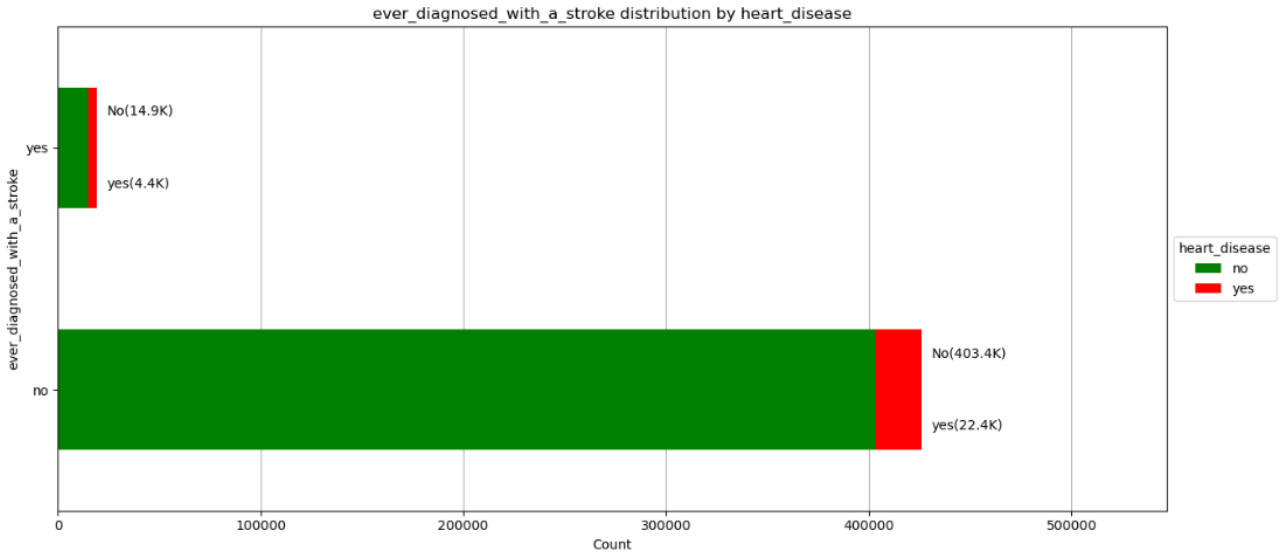
Figure 8



According to Figure 8 heart attack vs heart disease distribution:

- A significant number of individuals with heart disease have also been diagnosed with a heart attack 12.0K. This indicates a strong correlation between a previous heart attack and the presence of heart disease.
- There are also a substantial number of individuals with heart disease who have not been diagnosed with a heart attack 14.8K. This highlights that heart disease can develop without a prior heart attack diagnosis.
- The distribution suggests that while a prior heart attack is a significant indicator of heart disease, many individuals with heart disease have no history of a heart attack, emphasizing the need for comprehensive cardiovascular risk assessments beyond just heart attack history.

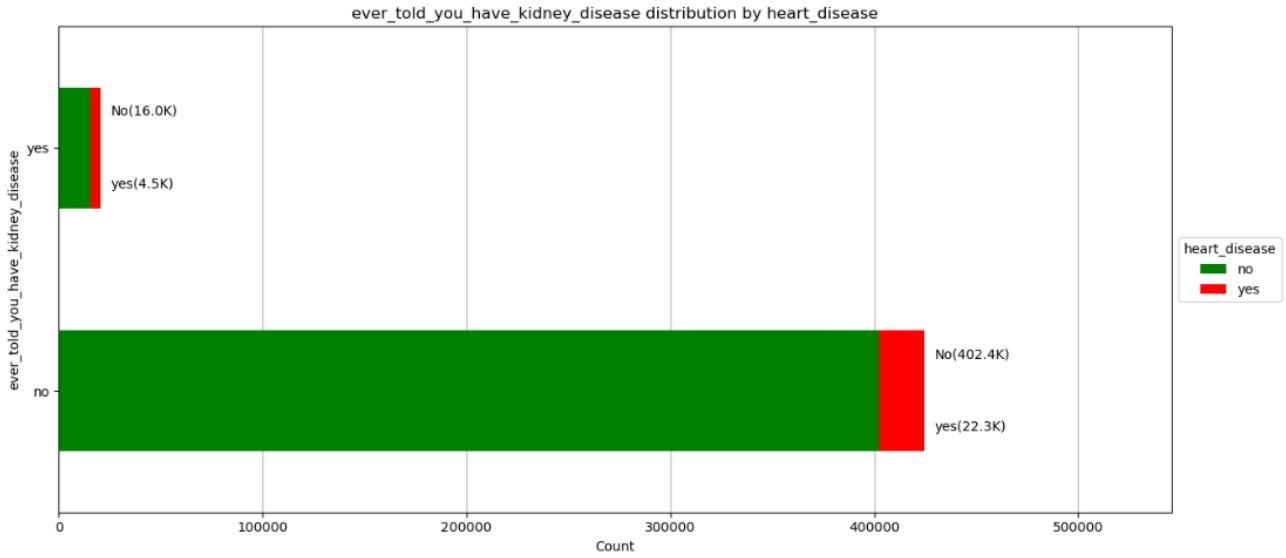
Figure 9



According to Figure 9 stroke vs heart disease distribution:

- A significant number of individuals with heart disease have also been diagnosed with a stroke 4.4K. This indicates a notable correlation between a previous stroke and the presence of heart disease.
- A larger number of individuals with heart disease have not been diagnosed with a stroke 22.4K. This shows that heart disease can occur independently of a stroke diagnosis.
- The distribution suggests that while a prior stroke is a significant risk factor for heart disease, many individuals with heart disease do not have a history of stroke. This highlights the importance of a comprehensive cardiovascular risk assessment, considering various risk factors beyond just stroke history.

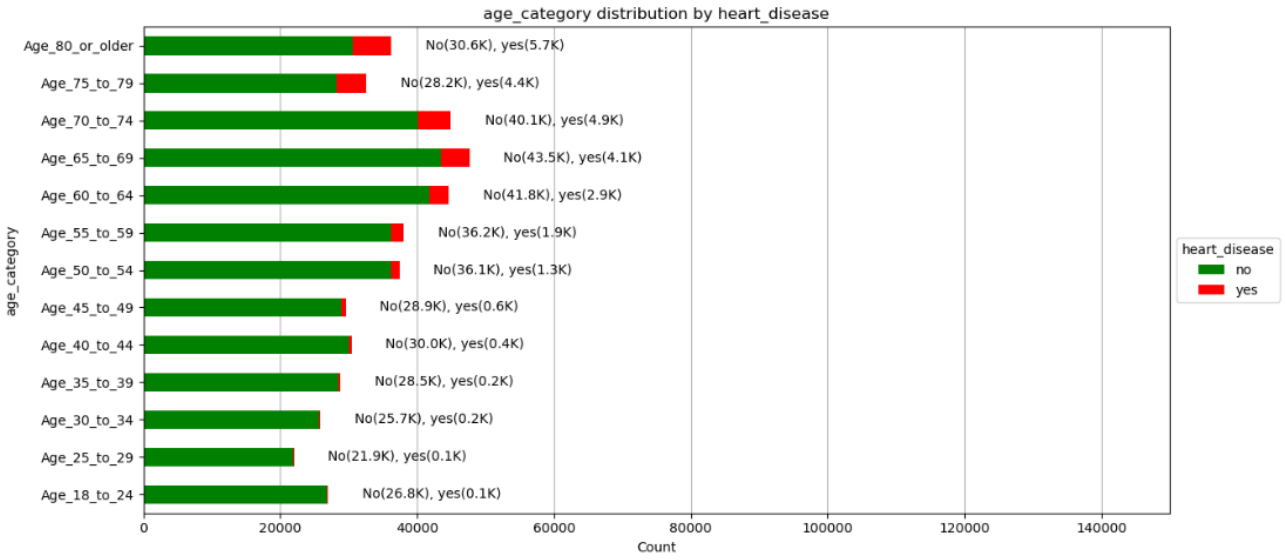
Figure 10



According to Figure 10 kidney vs heart disease distribution:

- A notable number of individuals with heart disease have also been diagnosed with kidney disease 4.5K. This indicates a correlation between kidney disease and heart disease.
- A larger number of individuals with heart disease have not been diagnosed with kidney disease 22.3K. This shows that heart disease frequently occurs independently of kidney disease.
- The distribution suggests that while kidney disease is a significant risk factor for heart disease, the majority of individuals with heart disease do not have a history of kidney disease. This underscores the importance of evaluating multiple risk factors for heart disease, including but not limited to kidney disease history.

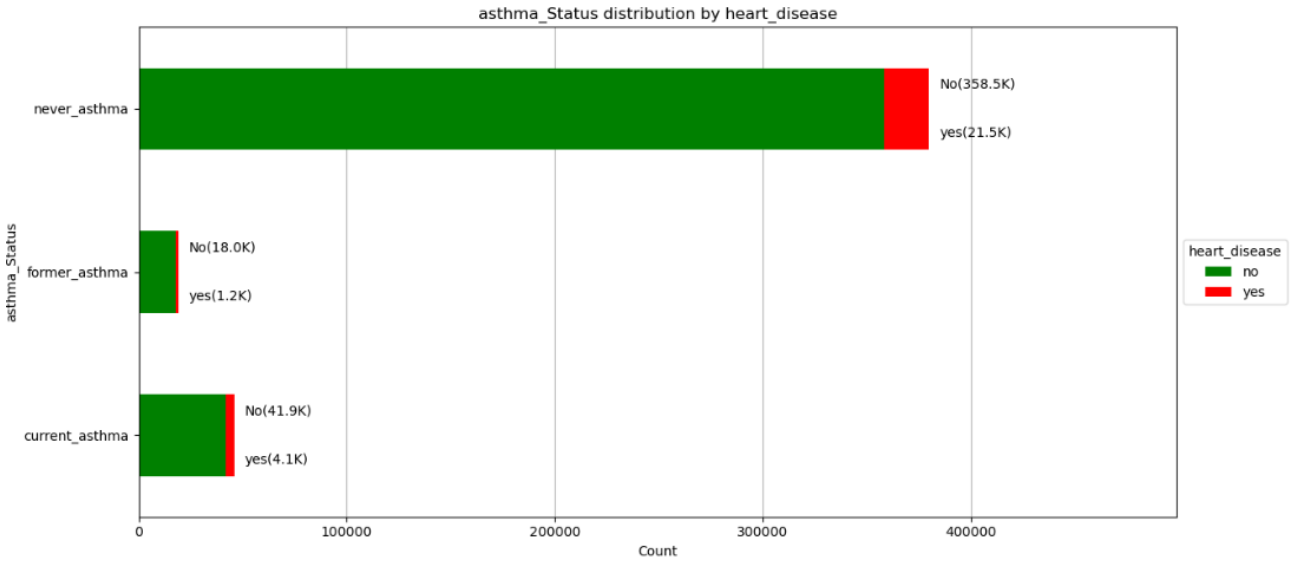
Figure 11



According to Figure 11 age vs heart disease distribution:

- The highest number of individuals with heart disease are in the age category of 70 to 74 4.9K, followed closely by the 80 or older category 5.7K, and 75 to 79 category 4.4K. This suggests a strong correlation between advanced age and the presence of heart disease.
- The number of individuals with heart disease generally increases with age, peaking in the older age categories. This indicates that age is a significant risk factor for heart disease.
- There are relatively few individuals with heart disease in the younger age categories (18 to 24 and 25 to 29), highlighting that while younger individuals can have heart disease, it is less common compared to older age groups.
- The distribution underscores the importance of age as a critical factor in heart disease risk assessment. It suggests that preventive measures and monitoring should be more rigorous as individuals age, particularly for those over 65.

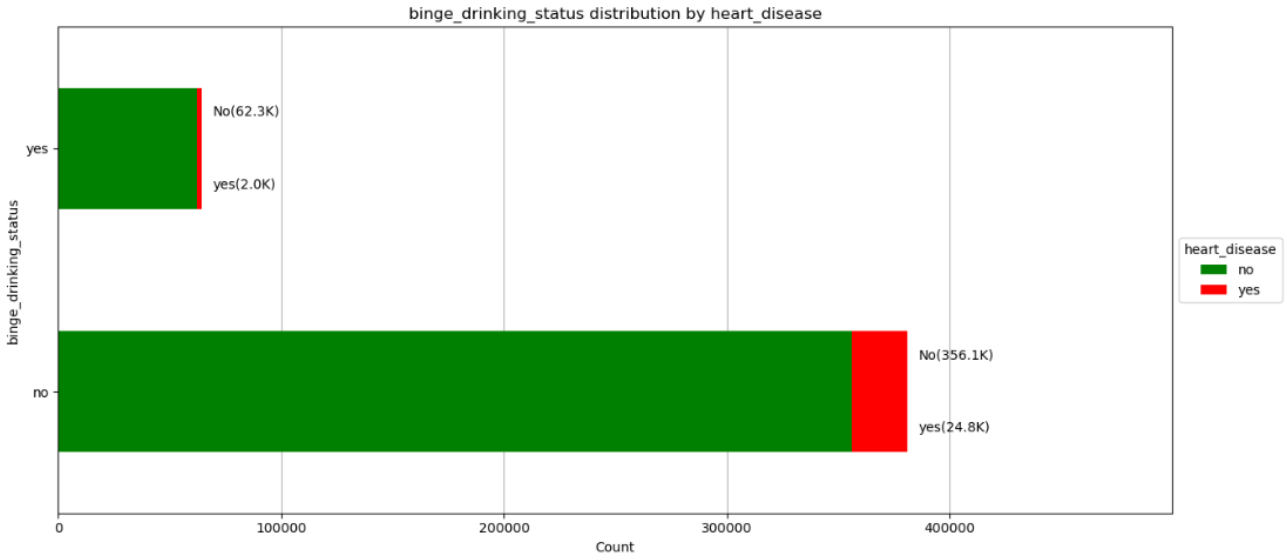
Figure 12



According to Figure 12 age vs heart disease distribution:

- The highest number of individuals with heart disease falls into the "Never Asthma" category 21.5K. This suggests that a large portion of individuals with heart disease do not have a history of asthma.
- A smaller number of individuals with heart disease have current asthma 4.1K, indicating a correlation between ongoing asthma and heart disease.
- The smallest number of individuals with heart disease are those with former asthma 1.2K. This suggests that having had asthma in the past is less common among individuals with heart disease compared to never having had asthma or currently having asthma.
- The distribution highlights the importance of considering asthma status in assessing the risk of heart disease. While many individuals with heart disease have never had asthma, there is still a significant group with current asthma, indicating the need for careful monitoring and management of both conditions.

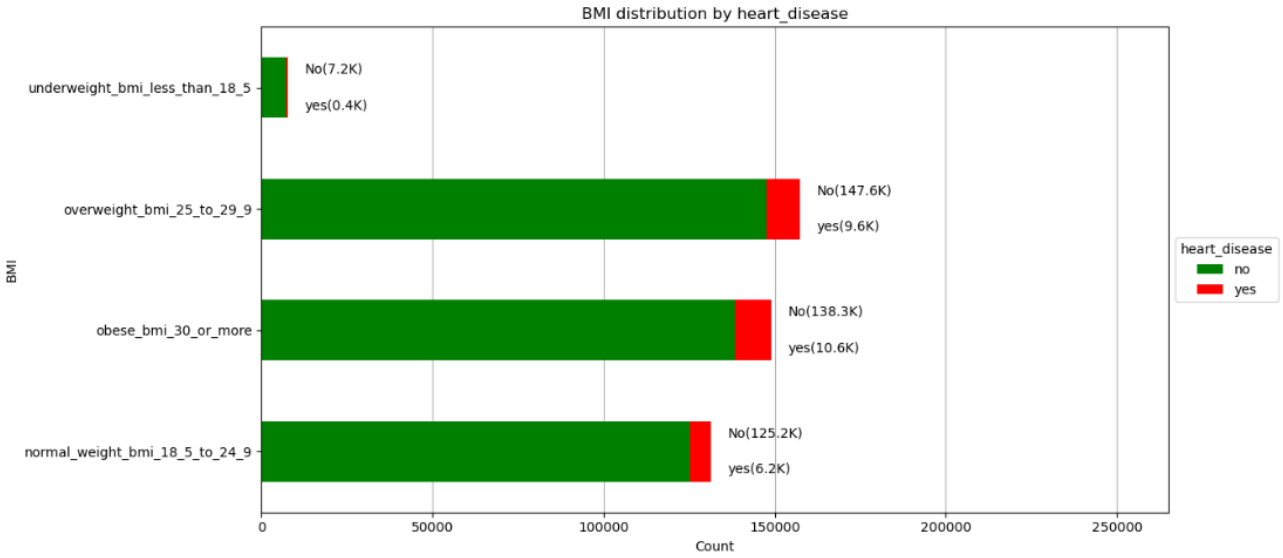
Figure 13



According to Figure 13 binge drinking vs heart disease distribution:

- The majority of individuals with heart disease do not engage in binge drinking 24.8K. This suggests that while binge drinking is a risk factor, many individuals with heart disease do not exhibit this behavior.
- A smaller number of individuals with heart disease report engaging in binge drinking 2.0K. This indicates that binge drinking is associated with heart disease, but it is less prevalent among heart disease patients compared to those who do not binge drink.
- The distribution highlights the importance of considering alcohol consumption patterns in assessing the risk of heart disease. While binge drinking is a significant risk factor, it is not the sole determinant of heart disease, as a substantial number of heart disease cases occur in individuals who do not binge drink. This underscores the multifactorial nature of heart disease risk and the need for comprehensive health assessments that include lifestyle factors such as alcohol consumption.

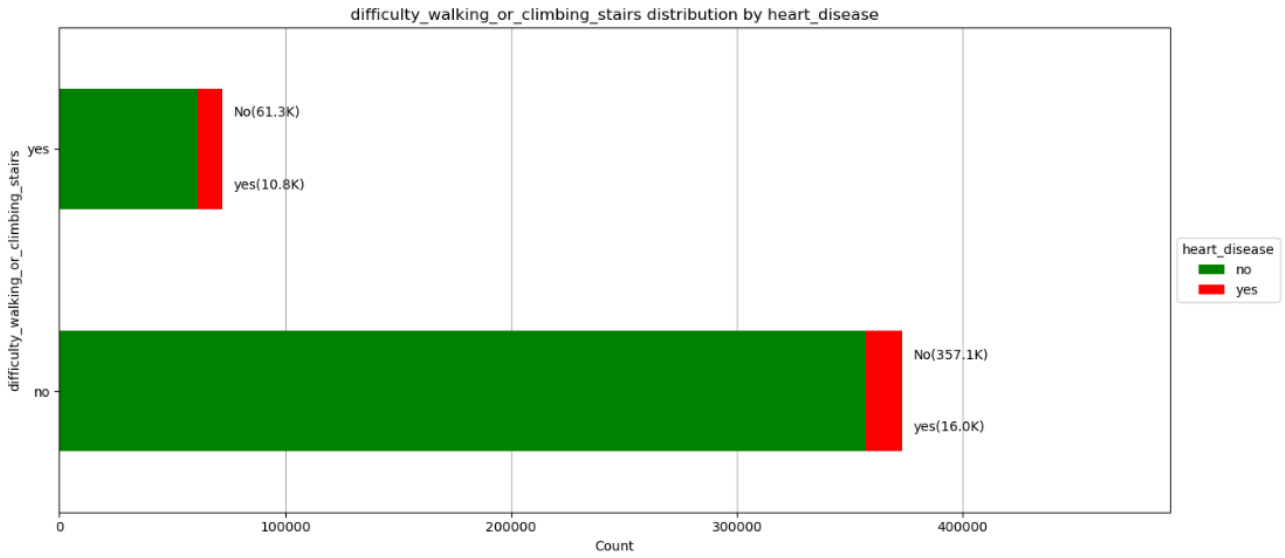
Figure 14



According to Figure 14 BMI vs heart disease distribution:

- The highest number of individuals with heart disease falls into the obese category 10.6K, indicating a strong correlation between obesity and heart disease.
- The overweight category also has a significant number of individuals with heart disease 9.6K, further highlighting the relationship between higher BMI and heart disease risk.
- Individuals with a normal weight 6.2K and underweight 0.4K have fewer cases of heart disease, suggesting that maintaining a normal weight may be associated with a lower risk of heart disease.
- This distribution underscores the importance of managing body weight as a critical factor in reducing the risk of heart disease, with obesity and overweight being key areas of concern.

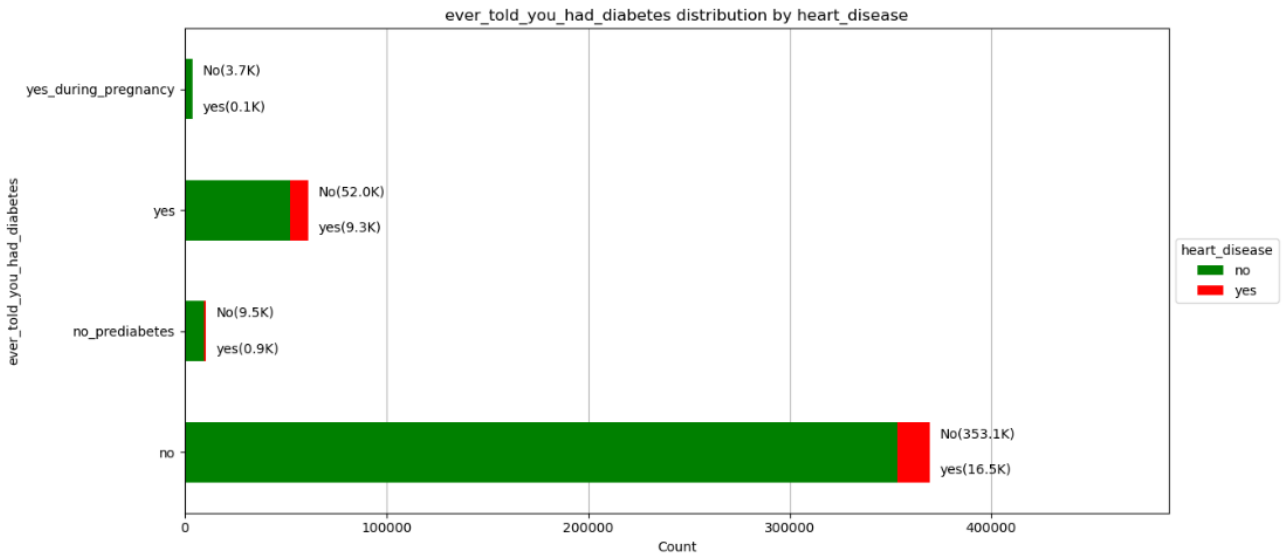
Figure 15



According to Figure 15 difficulty walking vs heart disease distribution:

- A notable number of individuals with heart disease report having difficulty walking or climbing stairs 10.8K. This indicates a strong association between mobility issues and the presence of heart disease.
- A slightly higher number of individuals with heart disease do not report difficulty walking or climbing stairs 16.0K. This shows that heart disease can occur even in those without significant mobility issues.
- The distribution suggests that difficulty in walking or climbing stairs is a significant risk factor for heart disease, but it also highlights that heart disease is present in a considerable number of individuals without mobility challenges. This underscores the importance of comprehensive cardiovascular risk assessments that consider a variety of health factors.

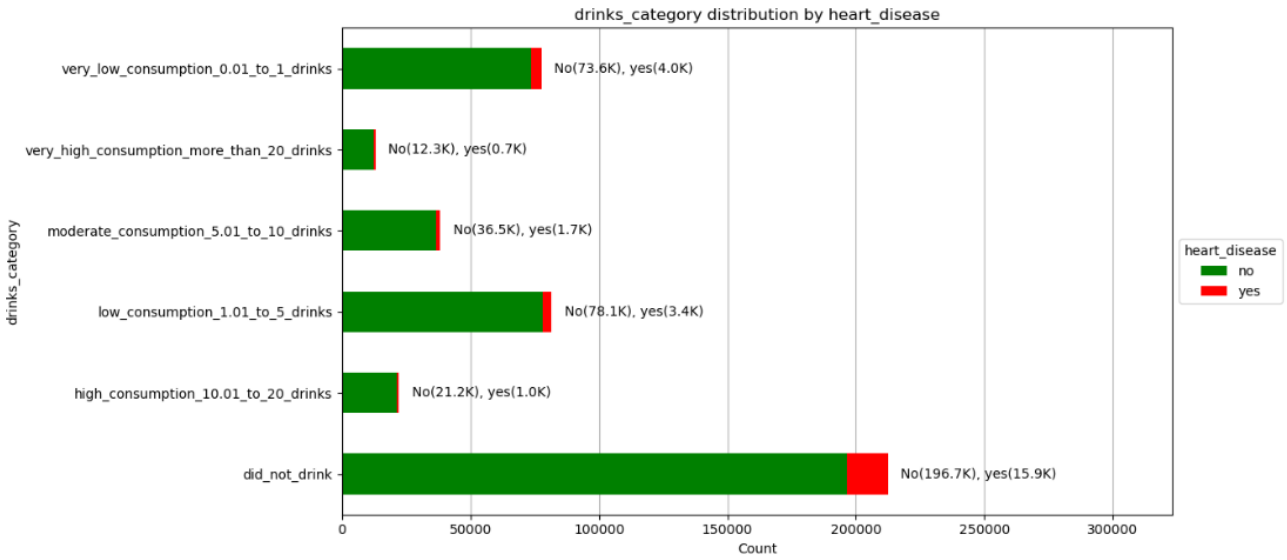
Figure 16



According to Figure 16 diabetes vs heart disease distribution:

- The highest number of individuals with heart disease are in the category of those who do not have diabetes 16.5K. This indicates that heart disease is prevalent even among those without a diabetes diagnosis.
- A significant number of individuals with heart disease have been diagnosed with diabetes 9.3K. This highlights the strong correlation between diabetes and heart disease.
- A smaller number of individuals with heart disease have prediabetes 0.9K or had diabetes during pregnancy 0.1K, suggesting these conditions are less common among those with heart disease compared to a full diabetes diagnosis.
- The distribution emphasizes the importance of monitoring and managing diabetes as a critical risk factor for heart disease. However, the presence of heart disease in individuals without diabetes underscores the multifactorial nature of cardiovascular risk.

Figure 17

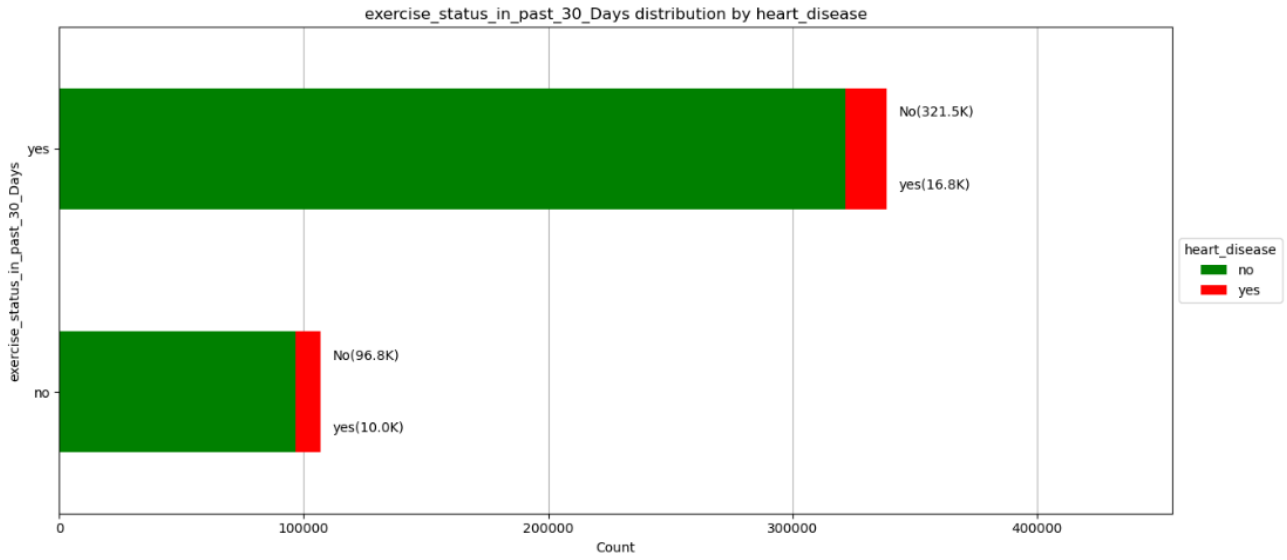


According to Figure 17 drinks category vs heart disease distribution:

- The highest number of individuals with heart disease fall into the "Did Not Drink" category with 15.9K individuals. This suggests that abstaining from alcohol is common among those with heart disease, possibly due to health reasons or pre-existing conditions.
- Very low alcohol consumption (0.01 to 1 drinks) is the second most common category among those with heart disease, with 4.0K individuals. This indicates that minimal alcohol consumption is still present among those with heart disease.
- Low consumption (1.01 to 5 drinks) includes 3.4K individuals with heart disease, highlighting that moderate drinking is also observed among this population.
- Moderate consumption (5.01 to 10 drinks) and high consumption (10.01 to 20 drinks) have fewer cases of heart disease with 1.7K and 1.0K individuals respectively.

- Very high consumption (more than 20 drinks) is the least common among those with heart disease, with 0.7K individuals.
- The distribution suggests that while many individuals with heart disease either do not drink or consume very little alcohol, moderate to high consumption is less common among this group. This emphasizes the importance of considering alcohol consumption habits in the context of heart disease risk and highlights the potential benefits of moderate or no alcohol consumption for heart health.

Figure 18

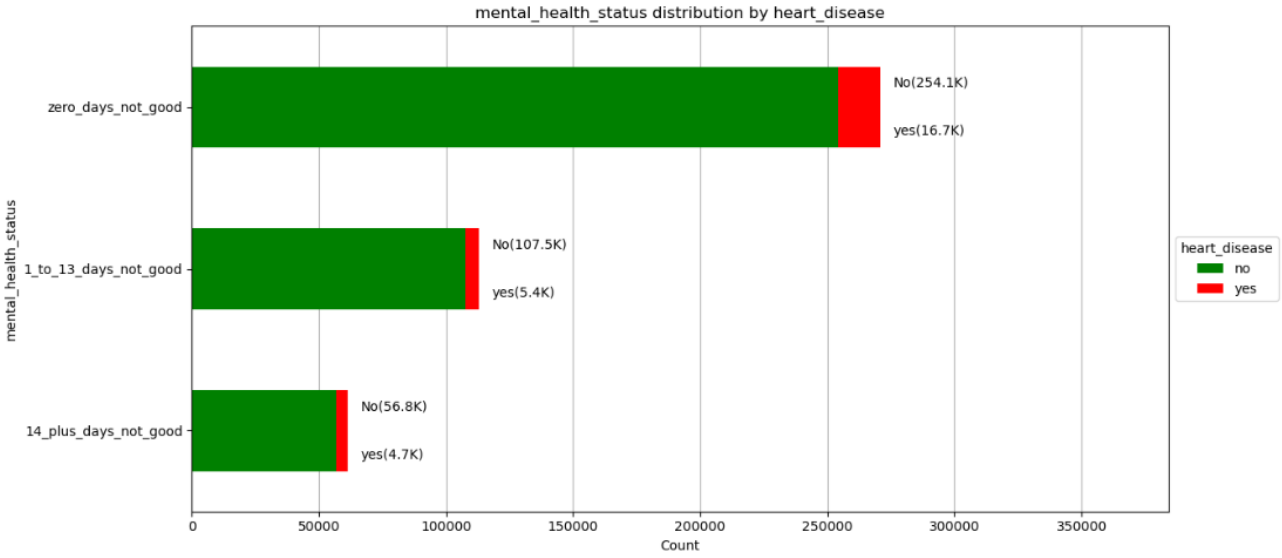


According to Figure 18 exercise vs heart disease distribution:

- A significant number of individuals with heart disease report having exercised in the past 30 days 16.8K. This suggests that exercise, while beneficial, does not entirely prevent the occurrence of heart disease, possibly due to other overriding risk factors.

- A notable number of individuals with heart disease report not having exercised in the past 30 days 10.0K. This indicates a correlation between lack of exercise and the presence of heart disease.
- The distribution highlights the importance of regular physical activity as a component of heart disease prevention. However, it also underscores that exercise alone is not sufficient to mitigate all risks associated with heart disease, emphasizing the need for a holistic approach to cardiovascular health that includes diet, lifestyle changes, and regular medical checkups.

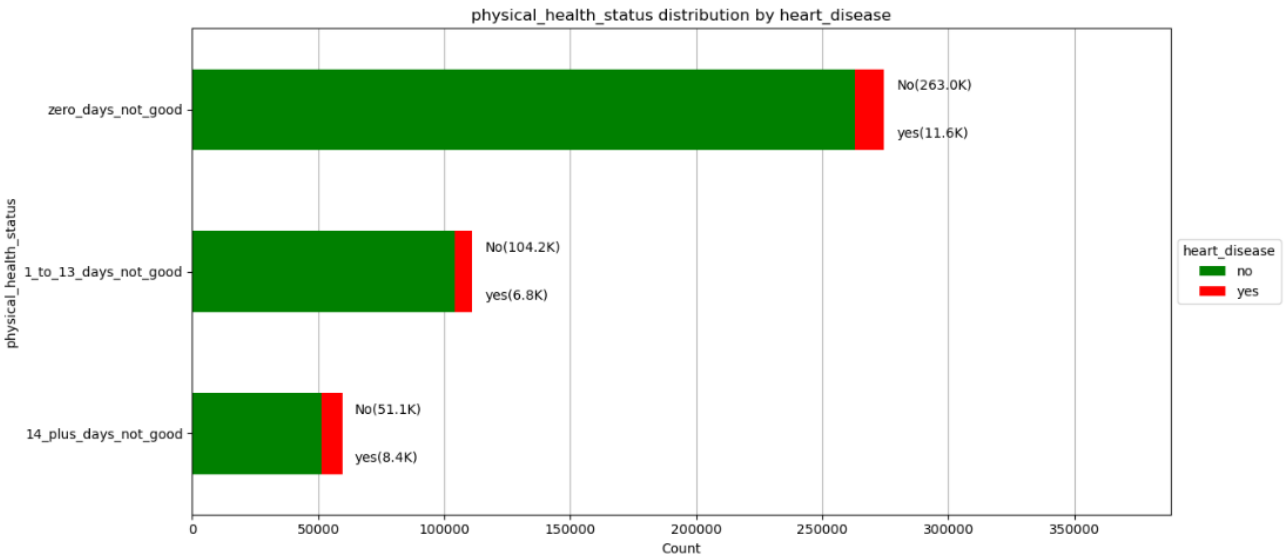
Figure 19



According to Figure 19 mental health vs heart disease distribution:

- The highest number of individuals with heart disease report having zero days of not feeling good mentally 16.7K. This suggests that many individuals with heart disease perceive their mental health as generally good.
- A significant number of individuals with heart disease report having 1 to 13 days of not feeling good mentally 5.4K. This indicates a noticeable correlation between moderate periods of poor mental health and the presence of heart disease.
- Individuals reporting 14 or more days of not feeling good mentally account for 4.7K cases of heart disease. This suggests that prolonged periods of poor mental health are also a factor among individuals with heart disease.
- The distribution highlights the importance of considering mental health status in assessing the risk of heart disease. While many individuals with heart disease report good mental health, there is a substantial group experiencing frequent poor mental health days, indicating the need for comprehensive health evaluations.

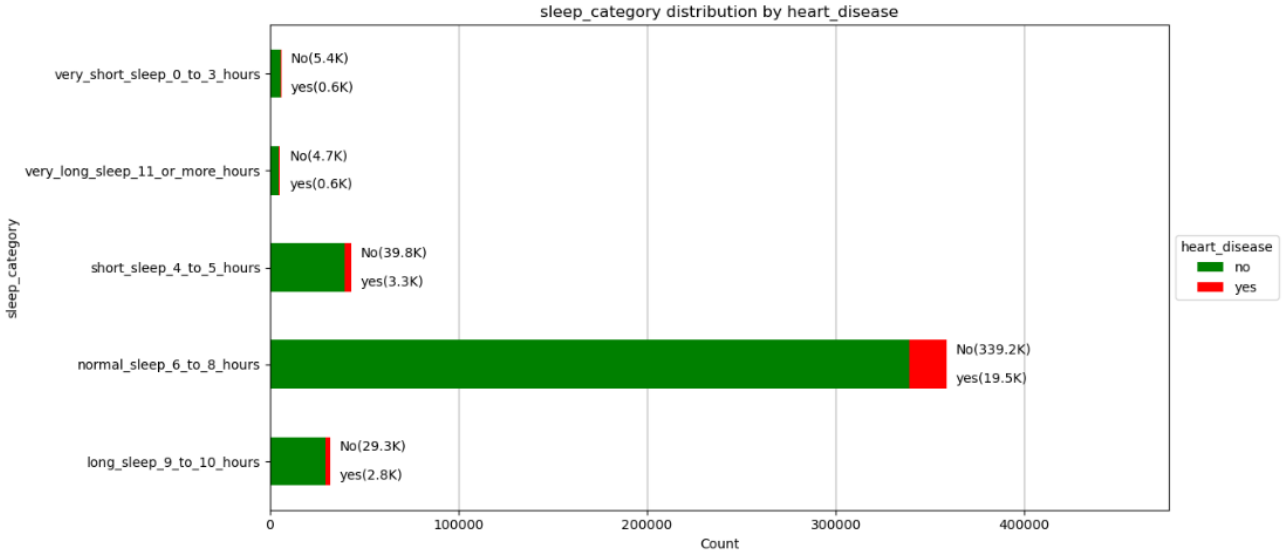
Figure 20



According to Figure 20 physical health vs heart disease distribution:

- The highest number of individuals with heart disease report having zero days of not feeling good physically 11.6K. This indicates that a significant portion of individuals with heart disease perceive their physical health as generally good.
- A notable number of individuals with heart disease report having 14 or more days of not feeling good physically 8.4K. This suggests a correlation between prolonged periods of poor physical health and heart disease.
- Individuals reporting 1 to 13 days of not feeling good physically account for 6.8K cases of heart disease.
- The distribution highlights the importance of considering physical health status in assessing the risk of heart disease. While many individuals with heart disease report good physical health, there is a substantial group experiencing frequent poor physical health days, which may be an indicator of underlying issues.

Figure 21

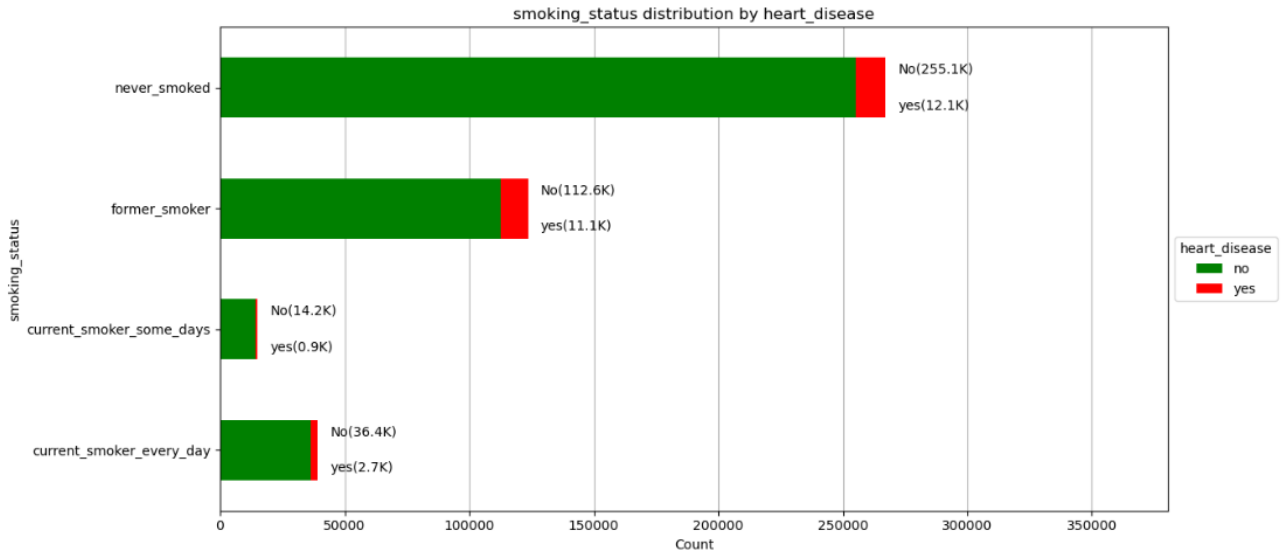


According to Figure 21 sleep vs heart disease distribution:

- The highest number of individuals with heart disease are in the normal sleep category (6 to 8 hours) with 19.5K individuals. This suggests that many individuals with heart disease report getting a standard amount of sleep.
- There are fewer individuals with heart disease in the very short sleep (0 to 3 hours) and very long sleep (11 or more hours) categories, each with 0.6K individuals. This indicates that extreme sleep durations are less common among those with heart disease, but they are still present.
- Short sleep (4 to 5 hours) has 3.3K individuals with heart disease, showing a significant correlation between insufficient sleep and heart disease.

- Long sleep (9 to 10 hours) includes 2.8K individuals with heart disease, suggesting that extended sleep duration is also associated with heart disease, though to a lesser extent than normal sleep duration.
- The distribution highlights the complex relationship between sleep duration and heart disease. While normal sleep duration is common among those with heart disease, both insufficient and excessive sleep are also important factors to consider in cardiovascular risk assessments. This underscores the importance of promoting healthy sleep habits as part of overall heart health.

Figure 22



According to Figure 22 smoking vs heart disease distribution:

- The highest number of individuals with heart disease are those who have never smoked 12.1K. This may reflect the larger population size of never smokers.

- Former smokers have a high number of heart disease cases 11.1K. This suggests that the health effects of smoking may persist even after quitting, leading to higher rates of heart disease among former smokers.
- Current smokers who smoke every day also have a notable number of heart disease cases 2.7K, indicating that ongoing smoking significantly contributes to heart disease risk.
- Interestingly, current smokers who smoke only on some days have the fewest heart disease cases 0.9K. This might be due to the smaller size of this subgroup or underreporting.

Why Former Smokers Have Higher Cases Compared to Current Smokers (Some Days)?

- Long-Term Effects of Smoking: Former smokers may have smoked for many years before quitting, leading to cumulative damage to their cardiovascular system. The adverse effects of prolonged smoking can persist long after quitting, increasing the risk of heart disease.
- Health Improvements: Current smokers who smoke only on some days might have a lower overall exposure to smoking-related toxins compared to those who smoked regularly for years before quitting.
- Population Size: The former smoker category likely includes a larger and more diverse group of individuals than the current smokers (some days) category, which might be relatively smaller and less representative of heavy, long-term smokers.

The distribution highlights the importance of considering smoking history in assessing heart disease risk. Even after quitting, former smokers continue to face a high risk, underscoring the long-term health impacts of smoking. Current smokers, especially those who smoke daily, also face significant risks, emphasizing the need for smoking cessation programs and ongoing monitoring of cardiovascular health.

5.6 Correlation: Heart Disease vs all features

In this analysis, we aim to understand the relationships between heart disease and various other features in our dataset. Specifically, we will be focusing on three main tasks:

- Encoding Categorical Variables: Converting all categorical features into numerical values using CatBoost encoding. This step ensures that we can effectively use these features in our analysis. Process: convert all categorical features to numerical values using the CatBoost encoder from the category_encoders package. This encoding method handles categorical variables effectively, preserving their informational content.
- Calculating Mutual Information: Assessing the predictive power of each feature with respect to heart disease by calculating the mutual information. Mutual information measures the dependency between the features and the target variable. Process: we calculate the mutual information for each feature with respect to heart disease. Mutual information provides a measure of the dependency between variables, allowing us to identify which features have the most predictive power.
- Calculating Pearson Correlation: Generating a heatmap to visualize the Pearson correlation coefficients between heart disease and all other features. This helps us understand the linear relationships in the dataset, although it is less informative for a binary target. Process: we calculate the Pearson correlation coefficients between heart disease and all other features. We visualize these correlations using a heatmap, which helps us easily identify strong linear relationships in the dataset.

5.7 Categorical Encoding with Catboost

Many machine learning algorithms require data to be numeric. Therefore, before training a model or calculating the correlation (Pearson) or mutual information (prediction power), we need to convert categorical data into numeric form. Various categorical encoding methods are available, and CatBoost is one of them. CatBoost is a target-based categorical encoder. It is a supervised encoder that encodes categorical columns according to the target value, supporting both binomial and continuous targets.

Target encoding is a popular technique used for categorical encoding. It replaces a categorical feature with average value of target corresponding to that category in training dataset combined with the target probability over the entire dataset. But this introduces a target leakage since the target is used to predict the target. Such models tend to be overfitted and don't generalize well in unseen circumstances.

A CatBoost encoder is similar to target encoding, but also involves an ordering principle in order to overcome this problem of target leakage. It uses the principle similar to the time series data validation. The values of target statistic rely on the observed history, i.e, target probability for the current feature is calculated only from the rows (observations) before it (Prokhorenkova et al., 2018; Dorogush, Ershov, & Gulin, 2018).

5.8 Mutual information – Prediction Power

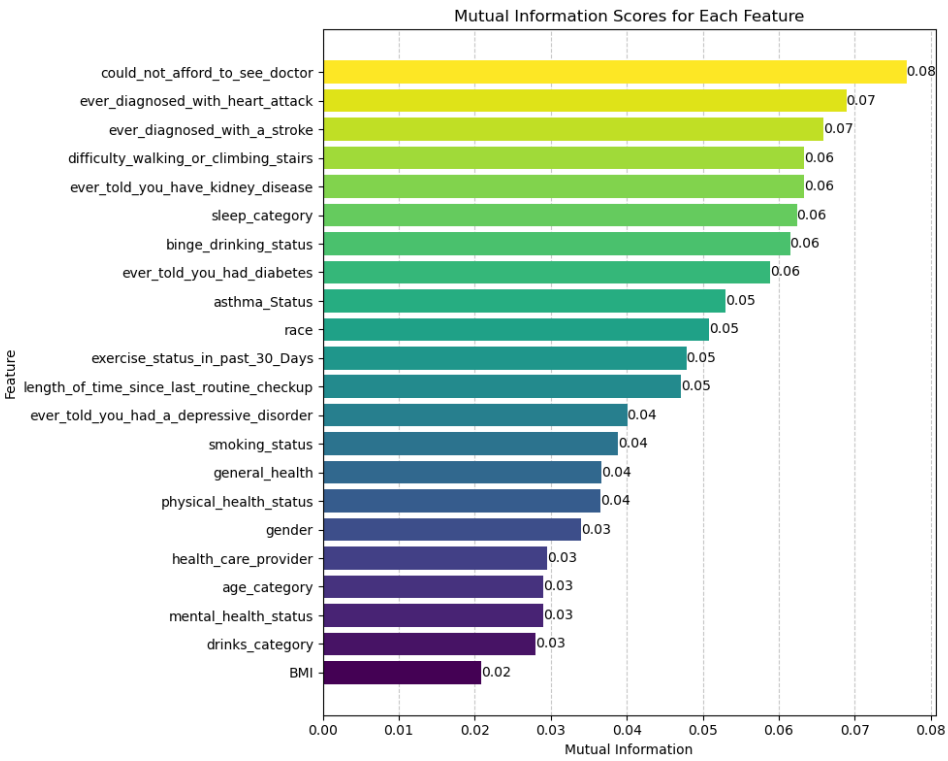
Mutual Information (MI) is a measure of the mutual dependence between two variables. It quantifies the amount of information obtained about one variable through another variable. Unlike correlation, which only captures linear relationships, mutual information can capture both linear and

non-linear relationships between variables, making it a powerful tool for feature selection in machine learning.

Mutual Information key advantages:

- Captures Non-Linear Relationships: Unlike traditional correlation measures (e.g., Pearson), mutual information can capture complex, non-linear relationships between features and the target variable. This is particularly useful in real-world datasets where relationships are rarely purely linear.
- Independence Detection: A mutual information score of zero indicates that two variables are completely independent. Non-zero mutual information indicates some level of dependency.
- Predictive Power: Higher mutual information scores suggest that a feature contains more information about the target variable, indicating higher predictive power. This helps in identifying the most relevant features for building robust predictive models (Brownlee, 2019; PLOS ONE, 2023).

Figure 23



According to Figure 23 mutual information measures the dependency between variables, capturing both linear and non-linear relationships. In this context, the mutual information score indicates how much information about the target variable heart_disease is provided by each feature. Higher scores imply stronger relationships between the feature and the target variable.

Top Features with High Mutual Information Scores:

- could_not_afford_to_see_doctor (0.08):

  This feature has the highest mutual information score, suggesting that the inability to afford to see a doctor is the most informative feature for predicting heart disease. This indicates a

significant association between financial barriers to healthcare and the likelihood of having heart disease.

- ever_diagnosed_with_heart_attack (0.07):

  Being previously diagnosed with a heart attack is highly informative for predicting heart disease. This is expected, as a history of heart attacks is a strong indicator of ongoing heart health issues.

- ever_diagnosed_with_a_stroke (0.07):

  Similarly, a history of strokes is another strong predictor of heart disease, suggesting that individuals with a history of stroke are more likely to have heart disease.

- difficulty_walking_or_climbing_stairs (0.06):

  Difficulty in physical activities like walking or climbing stairs is closely related to heart disease, likely reflecting the physical limitations caused by heart health issues.

- ever_told_you_have_kidney_disease (0.06):

  Kidney disease is significantly associated with heart disease, which aligns with the known comorbidities between cardiovascular and renal health issues.

- sleep_category (0.06):

  Sleep patterns or quality of sleep also show a notable relationship with heart disease, potentially indicating that poor sleep may be a risk factor.

- binge_drinking_status (0.06):

  Binge drinking is another informative feature, suggesting a relationship between alcohol consumption behaviors and heart disease risk.

- ever_told_you_had_diabetes (0.06):

  Diabetes is a well-known risk factor for heart disease, and its high mutual information score reflects this strong association.

- asthma_Status (0.05):

  The presence of asthma also provides information about the likelihood of having heart disease, possibly due to shared risk factors or comorbid conditions.

Features with Moderate Mutual Information Scores

- race (0.05), exercise_status_in_past_30_Days (0.05), length_of_time_since_last_routine_checkup (0.05):

  These features have moderate mutual information scores, indicating that they contribute useful but less substantial information about heart disease risk.

- ever_told_you_had_a_depressive_disorder (0.04), smoking_status (0.04), general_health (0.04), physical_health_status (0.04):

Mental health, smoking status, and general physical health also show meaningful associations with heart disease, reflecting the multifaceted nature of heart disease risk factors.

Features with Lower Mutual Information Scores

- gender (0.03), health_care_provider (0.03), age_category (0.03), mental_health_status (0.03), drinks_category (0.03), BMI (0.02): These features have lower mutual information scores, suggesting that while they do provide some information, their individual contributions to predicting heart disease are relatively smaller compared to the top features.

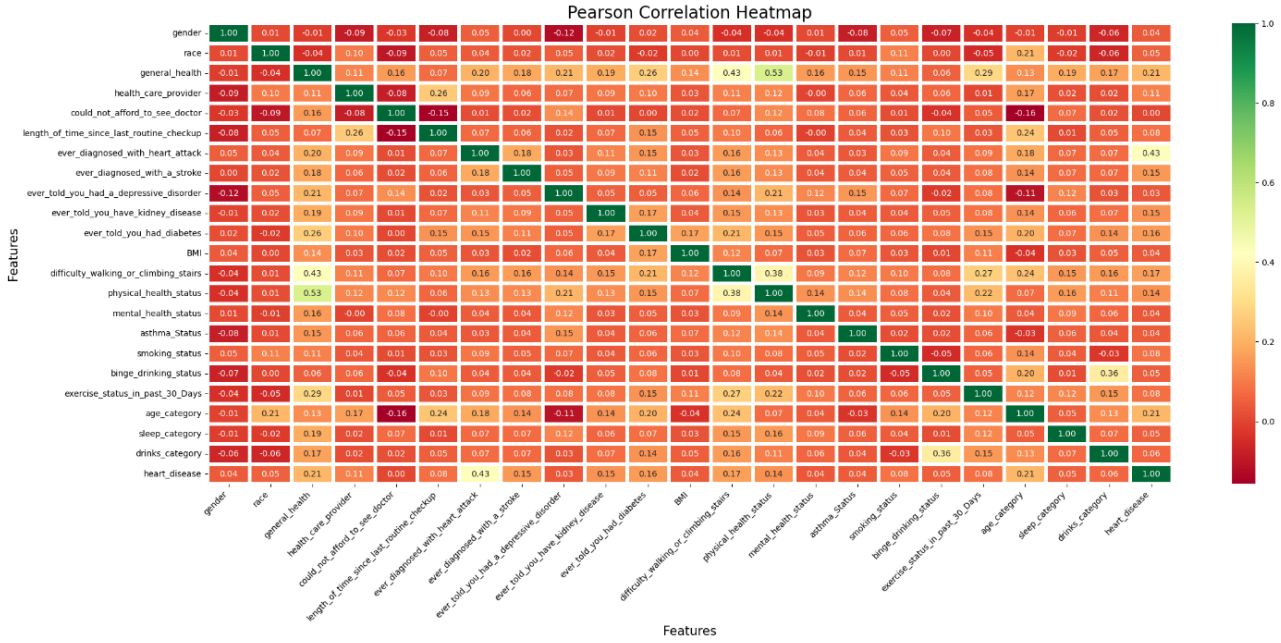
In summary, the mutual information scores provide a quantitative measure of how much each feature contributes to predicting heart disease. Features like financial barriers to healthcare, history of heart attacks or strokes, and physical limitations have the highest scores, highlighting their strong associations with heart disease. Understanding these relationships can help in prioritizing features for further analysis and in developing predictive models for heart disease.

5.9 Pearson Correlation

Pearson correlation, also known as Pearson's r, is a measure of the linear relationship between two continuous variables. It quantifies the degree to which a pair of variables are linearly related. The Pearson correlation coefficient can take values between -1 and 1, where:

- +1 indicates a perfect positive linear relationship.
- -1 indicates a perfect negative linear relationship.
- 0 indicates no linear relationship.

Figure 24



According to Figure 24 for collinearity the heatmap displays the Pearson correlation coefficients between various features, focusing on identifying independent features that exhibit high collinearity with each other. Here are the key observations regarding collinearity:

High Collinearity

- General Health and Physical Health Status (0.53): There is a strong positive correlation between general health and physical health status. This suggests that these two features are closely related, with better general health strongly associated with better physical health status.

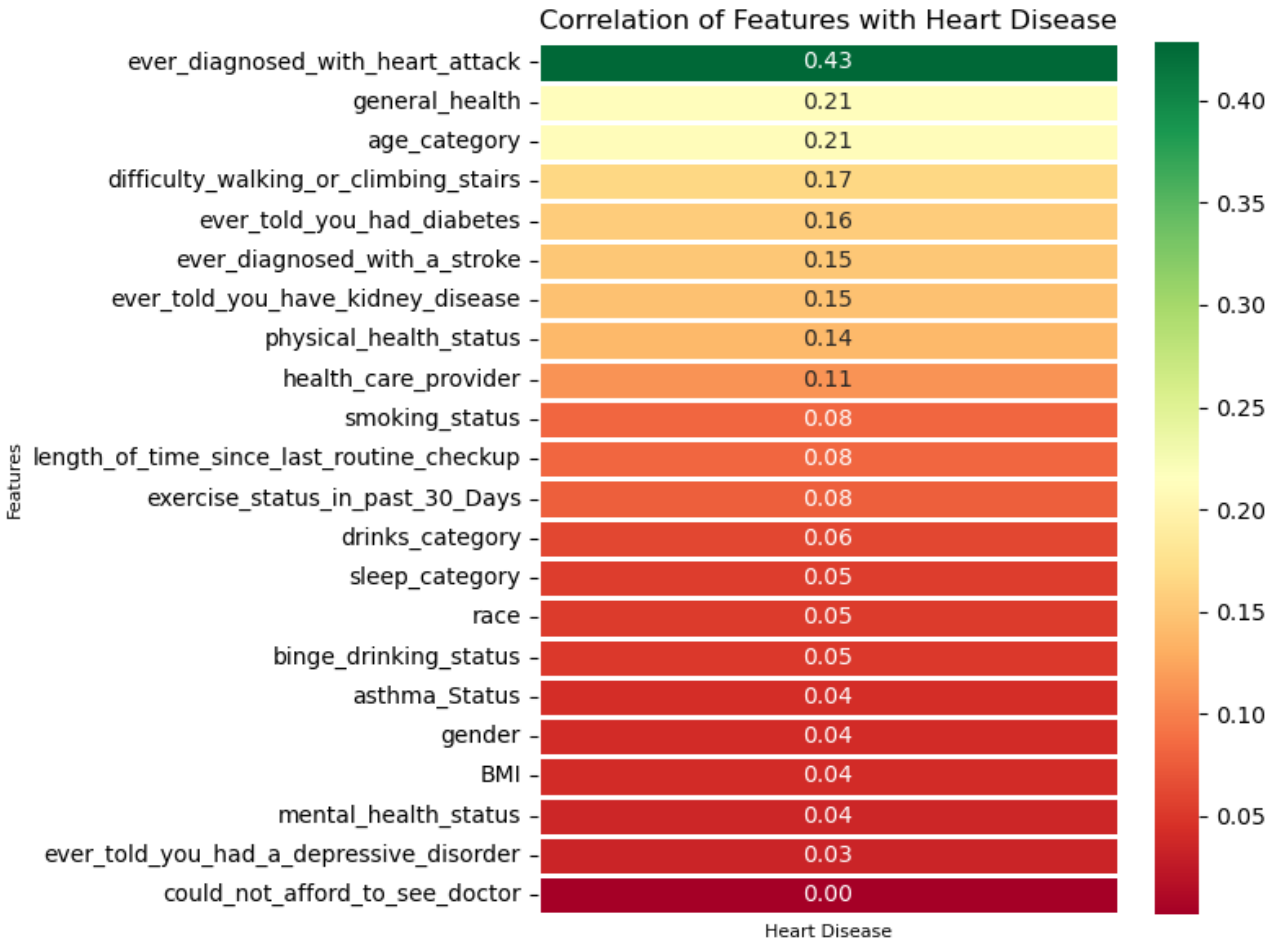
- Difficulty Walking or Climbing Stairs and Physical Health Status (0.43): A strong positive correlation indicates that difficulty in physical activities is a significant component of overall physical health.
- General Health and Difficulty Walking or Climbing Stairs (0.29): Moderate collinearity suggests that general health is significantly influenced by the ability to perform physical activities.
- Ever Told You Had Diabetes and General Health (0.26): Moderate collinearity suggests that diabetes status is an important factor in overall general health.
- Ever Told You Had Diabetes and Physical Health Status (0.26): Moderate collinearity indicates that diabetes has a considerable impact on physical health status.
- Length of Time Since Last Routine Checkup and General Health (0.26): Moderate collinearity suggests that routine checkups are related to overall health monitoring.
- General Health and Exercise Status in Past 30 Days (0.29): * Moderate collinearity suggests that exercise is an important factor in maintaining general health.

Moderate Collinearity

- Ever Diagnosed with Heart Attack and Ever Diagnosed with a Stroke (0.18): There is moderate collinearity, indicating that these conditions often occur together.
- Ever Told You Had Diabetes and Ever Diagnosed with Heart Attack (0.20): Moderate collinearity indicates a common comorbidity with heart disease.
- Ever Diagnosed with a Stroke and Ever Diagnosed with Heart Attack (0.18): Moderate collinearity indicating a common risk factor.

- Ever Told You Have Kidney Disease and Ever Told You Had Diabetes (0.19): Moderate collinearity suggests common comorbidities.
- Ever Diagnosed with Heart Attack and Ever Told You Had Kidney Disease (0.15): Moderate collinearity indicates a link between kidney disease and heart disease.
- Health Care Provider and General Health (0.26): Moderate collinearity indicates a link between having a healthcare provider and general health.
- Health Care Provider and Could Not Afford to See Doctor (0.16): Moderate collinearity suggests financial barriers to healthcare impact general health.
- Smoking Status and Binge Drinking Status (0.11): Weak collinearity suggests some lifestyle factors are related.

Figure 25



According to 25 the heatmap displays the Pearson correlation coefficients between various features and the target variable heart_disease. The values range from -1 to 1, where values closer to 1 indicate a strong positive linear correlation, values closer to -1 indicate a strong negative linear correlation, and values around 0 indicate no linear correlation. Here are the key observations:

Strongest Positive Linear Correlations

- Ever Diagnosed with Heart Attack (0.43): Interpretation: This feature has the highest positive linear correlation with heart disease, indicating that individuals who have had a

heart attack are significantly more likely to have heart disease. This is a strong relationship and aligns with medical knowledge that a history of heart attacks is a major risk factor for heart disease.

Moderate Positive Linear Correlations

- General Health (0.21): Interpretation: There is a moderate positive linear correlation between general health and heart disease. Poorer general health is associated with a higher likelihood of heart disease.
- Age Category (0.21): Interpretation: Age category has a moderate positive linear correlation with heart disease, suggesting that older individuals are more likely to have heart disease.
- Difficulty Walking or Climbing Stairs (0.17): Interpretation: This feature indicates that individuals who report difficulty with physical activities are more likely to have heart disease.
- Ever Told You Had Diabetes (0.16): Interpretation: There is a moderate positive linear correlation, indicating that individuals with diabetes are more likely to have heart disease.
- Ever Diagnosed with a Stroke (0.15): Interpretation: This feature has a moderate positive linear correlation with heart disease, suggesting that individuals who have had a stroke are also at higher risk for heart disease.
- Ever Told You Have Kidney Disease (0.15): Interpretation: There is a moderate positive linear correlation, indicating a link between kidney disease and heart disease.
- Physical Health Status (0.14): Interpretation: Individuals reporting poor physical health status are more likely to have heart disease.

Weak Positive Linear Correlations

- Health Care Provider (0.11): Interpretation: A weak positive linear correlation, suggesting some association between having a healthcare provider and heart disease, potentially due to increased diagnosis rates.
- Smoking Status (0.08): Interpretation: There is a weak positive linear correlation between smoking and heart disease, reflecting the known risk of smoking for cardiovascular conditions.
- Length of Time Since Last Routine Checkup (0.08): Interpretation: A weak positive linear correlation suggests that longer intervals between checkups are slightly associated with heart disease.
- Exercise Status in Past 30 Days (0.08): Interpretation: A weak positive linear correlation indicating that less frequent exercise might be associated with heart disease.
- Drinks Category (0.06): Interpretation: There is a weak positive linear correlation between drinking status and heart disease.
- Sleep Category (0.05): Interpretation: There is a weak positive linear correlation between sleep category and heart disease, suggesting that poor sleep might be associated with heart disease.
- Race (0.05): Interpretation: A weak positive linear correlation indicating a slight association between race and heart disease.
- Binge Drinking Status (0.05): Interpretation: A weak positive linear correlation between binge drinking and heart disease.
- Asthma Status (0.04): Interpretation: A weak positive linear correlation between asthma status and heart disease.

- Gender (0.04): Interpretation: A weak positive linear correlation between gender and heart disease.
- BMI (0.04): Interpretation: A weak positive linear correlation between BMI and heart disease.
- Mental Health Status (0.04): Interpretation: A weak positive linear correlation between mental health status and heart disease.
- Ever Told You Had a Depressive Disorder (0.03): Interpretation: A weak positive linear correlation suggesting a slight association between depression and heart disease.
- Could Not Afford to See Doctor (0.00): Interpretation: This feature has a negligible linear correlation with heart disease, indicating that financial barriers to healthcare do not have a significant direct linear correlation with heart disease in this dataset.

In Summary the heatmap reveals that the strongest predictor of heart disease is a history of heart attacks, followed by general health, age category, and difficulty with physical activities. Other features show weaker linear correlations, suggesting they have a less direct but still notable relationship with heart disease. These insights can be useful for identifying risk factors and guiding further analysis and modeling efforts.

5.10 Comparison between Pearson correlation and mutual information

- Linear Relationships: Pearson correlation is effective for identifying linear relationships. "Ever Diagnosed with Heart Attack" shows the highest linear correlation with heart disease, indicating a direct linear dependency.
- Non-Linear Relationships: Mutual information captures both linear and non-linear dependencies. "Could Not Afford to See Doctor" shows the highest score, suggesting that

financial barriers, although not strongly linearly correlated, have a significant impact on heart disease prediction.

- Overlap: Some features like "Ever Diagnosed with Heart Attack" and "General Health" are significant in both Pearson correlation and mutual information, indicating their strong overall predictive power.
- Unique Insights: Mutual information highlights features like "Could Not Afford to See Doctor" and "Sleep Category" that Pearson correlation does not emphasize, suggesting these features have non-linear relationships with heart disease.

In conclusion Pearson correlation is useful for identifying strong linear relationships but might miss non-linear dependencies. Mutual information provides a more comprehensive view by capturing both linear and non-linear relationships, highlighting features that might not be evident with Pearson correlation alone. Using both methods provides a holistic understanding of the relationships between independent features and the target variable, aiding in better feature selection for predictive modeling.

6. Modeling

6.1 Introduction

Here, we will be fitting and evaluating multiple machine learning models to classify heart disease:

- Logistic Regression
- Random Forest
- XGBoost
- LightGBM
- Balanced Bagging
- Easy Ensemble
- Balanced Random Forest
- Balanced Bagging (LightGBM): Balanced Bagging as a Wrapper and LightGBM as a base estimator
- Easy Ensemble (LightGBM): Easy Ensemble as a Wrapper and LightGBM as a base estimator

Our goal is to accurately predict heart disease risk using these models. We will employ hyperparameter tuning with Optuna to optimize each model's performance. Additionally, we will leverage
the BalancedRandomForestClassifier, BalancedBaggingClassifier and EasyEnsembleClassifier fro m the imbalanced-learn library to address class imbalance. These classifiers use bootstrapped sampling to balance the dataset, ensuring robust classification of minority classes. By focusing on

underrepresented data, it enhances model performance, making it particularly suitable for imbalanced datasets like heart disease prediction.

Through this comprehensive approach, we aim to develop a reliable and effective model for heart disease risk assessment, contributing to better health outcomes.

6.2 Baseline Modeling

Here, we will fit the following models listed below and compare their performance at the class-specific level:

- Logistic Regression
- Random Forest
- XGBoost
- LightGBM
- Balanced Bagging
- Easy Ensemble
- Balanced Random Forest
- Balanced Bagging (LightGBM): Balanced Bagging as a Wrapper and LightGBM as a base estimator
- Easy Ensemble (LightGBM): Easy Ensemble as a Wrapper and LightGBM as a base estimator

6.3 Class-specific level Metrics Summary

Table 1

Model	Class	Accuracy	Precision	Recall	F1 Score	ROC AUC
Logistic Regression	0	0.942826	0.952530	0.988374	0.970121	0.881984
Logistic Regression	1	0.942826	0.573123	0.240642	0.338961	0.881984
Random Forest	0	0.939535	0.951774	0.985551	0.968368	0.841807
Random Forest	1	0.939535	0.508143	0.230131	0.316791	0.841807
LightGBM	0	0.943624	0.952778	0.988984	0.970543	0.884863
LightGBM	1	0.943624	0.589938	0.244330	0.345547	0.884863
XGBoost	0	0.942546	0.952736	0.987824	0.969962	0.881696
XGBoost	1	0.942546	0.565700	0.244514	0.341445	0.881696
Balanced Bagging	0	0.839273	0.975463	0.850234	0.908554	0.847909
Balanced Bagging	1	0.839273	0.224994	0.670293	0.336902	0.847909
Easy Ensemble	0	0.815045	0.982147	0.817915	0.892539	0.880714
Easy Ensemble	1	0.815045	0.215431	0.770791	0.336744	0.880714
Balanced Random Forest	0	0.781999	0.983462	0.780991	0.870610	0.872684
Balanced Random Forest	1	0.781999	0.191076	0.797529	0.308290	0.872684
Balanced Bagging (LightGBM)	0	0.799746	0.984274	0.799531	0.882336	0.885894
Balanced Bagging (LightGBM)	1	0.799746	0.206251	0.803061	0.328209	0.885894
Easy Ensemble (LightGBM)	0	0.793546	0.984505	0.792630	0.878209	0.885778
Easy Ensemble (LightGBM)	1	0.793546	0.201685	0.807671	0.322771	0.885778

According to Table 1:

- High Recall, Low Precision and F1 Score: The majority of models show poor recall for class 1 (patients with heart disease), except for Balanced Bagging, Easy Ensemble, Balanced Random Forest, and when these models are combined with LightGBM. This indicates that most models struggle to identify positive cases (patients with heart disease), resulting in a significant number of false negatives (patients incorrectly identified as not having heart disease).
- Balanced Bagging and Easy Ensemble:

    - Balanced Bagging and Easy Ensemble models, along with Balanced Random Forest, are designed to handle class imbalance by balancing the classes during training.
    - Performance:
        - They achieve higher recall for class 1, meaning they capture most of the actual positive cases.
        - The trade-off is typically lower precision, leading to a lower F1 score.
- Medical Context Implication: In a medical context, high recall is crucial as it is important to identify as many true positive cases as possible, even at the cost of some false positives. Missing a true positive (false negative) could be more critical than having a false positive.
- Using LightGBM as Base Estimator:
    - Performance with LightGBM:
        - When using LightGBM as the base estimator in Balanced Bagging and Easy Ensemble, the results show improved recall for class 1.
        - These models also have slightly better ROC AUC scores (0.885894 and 0.885778, respectively), indicating a good balance between sensitivity and specificity.
        - LightGBM is a powerful gradient boosting framework known for its efficiency and performance, which helps in achieving better overall performance metrics.
    - Improvement:
        - When using Easy Ensemble as a wrapper and LightGBM as a base estimator, the Recall for class 1 (heart disease patients) improves significantly from 24.4% (in standalone LightGBM) to 80.7%.

    - The ROC AUC improves from 88.4% to 88.6% for class 1, showing a better balance between correctly identifying true positives and minimizing false positives.

- Practical Implications: Heart Disease Classification Task:
    - Identifying patients with heart disease (true positives) is critical.
    - High recall is generally more desirable, even at the cost of having more false positives.
    - High Recall ensures most patients with heart disease are identified, which is crucial for early intervention and treatment.
    - False Positives, while not ideal, can be managed through follow-up testing and further medical evaluation (Uyar et al., 2022; Abbas et al., 2022).
- Conclusion: Balanced Bagging, Easy Ensemble, and Balanced Random Forest models, particularly with LightGBM as the base estimator, provide a good balance between identifying true positives and maintaining a reasonable rate of false positives. For a medical application such as heart disease prediction, these approaches ensure that most cases of heart disease are identified, enabling timely medical intervention, which is crucial for patient care.

6.4 Model Selection

Based Table 1, the **Easy Ensemble (LightGBM) model** is the best choice for being the final model:

- Accuracy: 0.793546
- Precision: 0.201685
- Recall: 0.807671

  - F1 Score:0.322771
  - ROC AUC: 0.885778
- Why Easy Ensemble (LightGBM)?
  - High Recall (0.807671): This ensures we capture the majority of the true positive cases (patients with heart disease).
  - Good ROC AUC (0.885778): Indicates good overall model performance in distinguishing between classes.
  - Balanced Handling of Class Imbalance: Easy Ensemble effectively manages class imbalance, which is often a challenge in medical datasets.

6.5 Hyper Parameter Tuning using OPTUNA

Using Optuna for hyperparameter tuning is beneficial due to its ability to dynamically construct search spaces, efficient sampling and pruning algorithms, and ease of integration with various machine learning frameworks. These features enable faster and more effective hyperparameter optimization, leading to improved model performance (Akiba et al., 2019; Analytics Vidhya, 2020).

6.6 Fitting best model (Tuned)

Table 2

Model	Class	Accuracy	Precision	Recall	F1 Score	ROC AUC
Easy Ensemble (LightGBM)	0	0.786862	0.984788	0.785166	0.873720	0.883942
Easy Ensemble (LightGBM)	1	0.786862	0.197094	0.813019	0.317274	0.883942

According to Table 1:

- Class 0:
  - The model has a high precision (0.984788) for class 0, indicating that when it predicts class 0, it is correct 98.48% of the time.
  - The recall for class 0 is also reasonably high (0.785166), meaning it correctly identifies 78.52% of all actual class 0 instances.
  - The F1 score (0.873720) shows a good balance between precision and recall.
  - The ROC AUC for class 0 is 0.785166, indicating good discriminative ability.
- Class 1:
  - The precision for class 1 is low (0.197094), meaning many of the predicted class 1 instances are actually class 0.
  - However, the recall for class 1 is high (0.813019), indicating the model is good at identifying actual class 1 instances.
  - The F1 score for class 1 is relatively low (0.317274), suggesting a trade-off between precision and recall.

- The ROC AUC for class 1 is the same as for class 0 (0.883942), indicating overall good model performance in distinguishing between classes.

Comparison to Separate and Combined Models:

- LightGBM Alone:
  - LightGBM typically has strong performance due to its gradient boosting capabilities. It may achieve high accuracy and good precision/recall balances for both classes.
  - However, LightGBM alone might struggle with class imbalance, often resulting in lower recall for minority classes (class 1: Recall 24.4%).
- EasyEnsemble Alone:
  - EasyEnsemble without LightGBM as the base estimator focuses on balancing the data using under-sampling and creating multiple models.
  - This approach improves recall for minority classes but might not achieve the high precision that LightGBM offers.
  - The combined approach of using EasyEnsemble with LightGBM leverages the strengths of both techniques, enhancing both precision and recall, particularly for the minority class.
- Combined (Tuned):
  - When tuned, EasyEnsemble with LightGBM as the base estimator provides a balanced approach to handle class imbalance.
  - The combined method shows improved recall for class 1 from24.4% to 81.3%while maintaining a good precision for class 0(0.984788)`.

- This combination also ensures that the model has a robust overall performance as indicated by the ROC AUC of 0.883942.

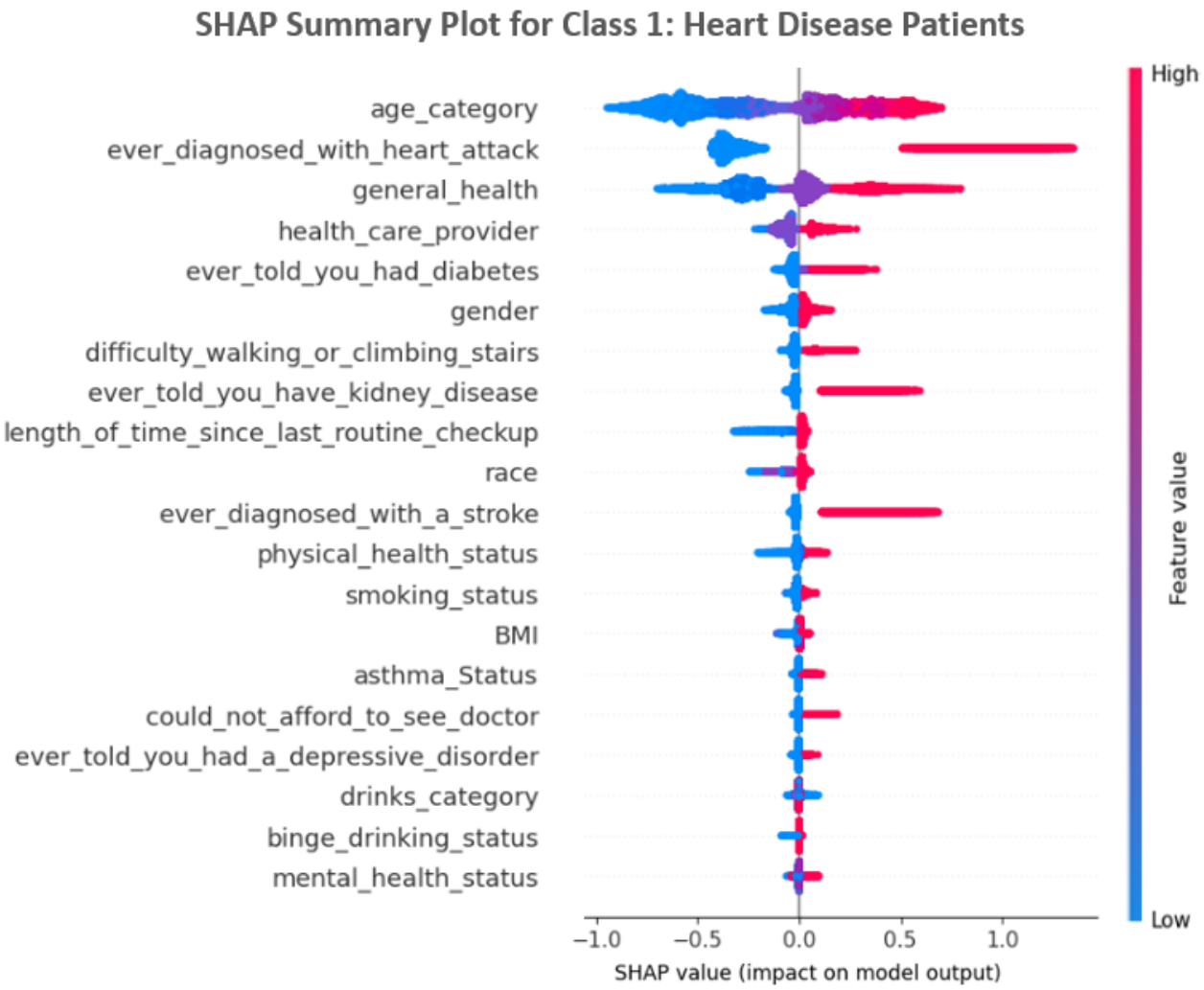
In conclusion Using EasyEnsemble with LightGBM as the base estimator, especially when hyperparameters are tuned, offers a comprehensive solution to handling class imbalance. It ensures high precision and recall for class 0 and significantly improves recall for class 1, although precision for class 1 remains a challenge. This combined approach outperforms using LightGBM or EasyEnsemble separately by effectively leveraging the strengths of both methods.

6.7 Tuned Best Model Features Importance Using SHAP

SHAP (SHapley Additive exPlanations) is a powerful tool for interpreting machine learning models as it provides a consistent and objective way to understand how a model makes predictions and the influence of each feature.

SHAP values are grounded in cooperative game theory and provide explanations that adhere to properties such as local accuracy, missingness, and consistency, ensuring reliable feature attribution and it's particularly useful for debugging models, identifying feature importance, and detecting biases, thereby making machine learning models more interpretable and trustworthy (Lundberg & Lee, 2017; Datacamp, 2023).

Figure 26



According to Figure 26 SHAP summary plot shows the impact of each feature on the model's output for predicting heart disease (class 1). Each dot represents a SHAP value for a feature, with the color indicating the feature's value (red for high and blue for low). Here's a detailed interpretation:

- **High Positive Impact:**

- Age Category: Higher age (represented by red dots) increases the likelihood of heart disease. Age is a significant risk factor, with older individuals being more prone to heart disease.
- Ever Diagnosed with Heart Attack: A history of heart attacks (red dots) greatly increases the likelihood of heart disease. This past medical history is a strong indicator of recurring or persistent heart issues.

- **Moderate Positive Impact:**
  - General Health: Poor general health (red dots) increases the likelihood of heart disease. Individuals with overall poor health are at higher risk.
  - Health Care Provider: Frequent visits to a healthcare provider (red dots) indicate a higher likelihood of heart disease, possibly due to ongoing health issues necessitating regular check-ups.
  - Ever Told You Had Diabetes: Being told by a healthcare provider that you have diabetes (red dots) increases the likelihood of heart disease. Diabetes is a well-known risk factor for cardiovascular diseases.
  - Gender: Certain gender-related factors (likely male, indicated by red dots) increase the likelihood of heart disease. Men generally have a higher risk of heart disease at a younger age compared to women.
  - Difficulty Walking or Climbing Stairs: Difficulty in these activities (red dots) indicates a higher risk of heart disease, possibly due to underlying cardiovascular issues.

- Ever Told You Have Kidney Disease: A history of kidney disease (red dots) increases the likelihood of heart disease. Kidney disease can be associated with cardiovascular complications.
- Length of Time Since Last Routine Checkup: A longer time since the last checkup (red dots) increases the likelihood of heart disease. Regular check-ups can help manage and prevent health issues.
- Race: Certain racial factors (red dots) increase the likelihood of heart disease, highlighting the role of demographic and genetic factors.
- Ever Diagnosed with a Stroke: A history of stroke (red dots) increases the likelihood of heart disease, as both share common risk factors.
- Physical Health Status: Poor physical health (red dots) increases the likelihood of heart disease, reflecting the impact of overall physical well-being on heart health.
- Smoking Status: Being a smoker (red dots) significantly increases the likelihood of heart disease. Smoking is a major risk factor for cardiovascular diseases.
- BMI: Higher BMI (red dots) increases the likelihood of heart disease. Obesity is closely linked to cardiovascular risk.
- Asthma Status: Higher severity of asthma (red dots) increases the risk, possibly due to the overall impact of chronic respiratory conditions on health.
- Could Not Afford to See Doctor: Financial barriers to healthcare (red dots) increase the likelihood of heart disease, likely due to untreated health conditions.
- Ever Told You Had a Depressive Disorder: A history of depressive disorder (red dots) increases the likelihood of heart disease, indicating a connection between mental and cardiovascular health.

  - Drinks Category: Higher alcohol consumption (red dots) is associated with an increased risk of heart disease. Excessive drinking can negatively impact heart health.
  - Binge Drinking Status: Higher frequency of binge drinking (red dots) increases the likelihood of heart disease, highlighting the adverse effects of excessive alcohol intake.
  - Mental Health Status: Poor mental health (red dots) increases the likelihood of heart disease, emphasizing the importance of mental well-being for heart health.

- **Mixed Impact:**
  - Race: Certain racial factors have a mixed influence but can increase the likelihood of heart disease, showing the importance of considering demographic variables in health risk assessments.
  - Asthma Status: While generally having a moderate positive impact, higher severity of asthma (red dots) increases the risk of heart disease. This indicates that severe respiratory issues can contribute to cardiovascular risk.

In conclusion SHAP summary plot illustrates that various factor such as age, history of heart attacks, general health, and diabetes significantly impact the likelihood of heart disease. The analysis emphasizes the importance of regular health check-ups, managing chronic conditions, and addressing both physical and mental health to mitigate the risk of heart disease.

7. AI-Powered Heart Disease Risk Assessment App

7.1 Overview

The AI-powered heart disease assessment app is designed to provide users with tailored risk scores and actionable recommendations to mitigate their heart disease risk. It leverages advanced machine learning models and data visualization techniques to offer a user-friendly and interactive experience.

7.2 technologies used

- Python: The core logic and machine learning models are implemented using Python, ensuring robust performance and ease of integration with various data processing and machine learning libraries.
- HTML and CSS: Custom HTML and CSS were used to enhance the user interface and improve the overall layout and aesthetics of the app.
- Streamlit: The app is built and deployed using Streamlit, a powerful framework for creating and sharing data applications quickly. Streamlit simplifies the process of turning data scripts into interactive web applications.

7.3 Deployment

The app is deployed using Streamlit's sharing platform, making it accessible online at AI-Powered Heart Disease Assessment. This deployment method ensures that the app is easily accessible to users without requiring complex setup or installation processes.

7.4 Features

- User Input: Users can enter their health information, including age, BMI, physical activity levels, smoking status, and medical history.
- Risk Assessment: The app analyzes the input data using trained machine learning models to provide a personalized heart disease risk score.
- Recommendations: Based on the risk assessment, the app offers actionable recommendations to help users mitigate their heart disease risk.
- Visualization: The app includes various visualizations to help users understand their risk factors and the impact of different health parameters on their overall heart disease risk.

7.5 Benefits

- Accessibility: The app's deployment on Streamlit Share makes it easily accessible from any device with internet connectivity.
- Interactivity: Users can interact with the app in real-time, entering their data and receiving immediate feedback on their heart disease risk.
- Interpretability: By using SHAP (SHapley Additive exPlanations) values, the app provides insights into the feature importance and model predictions, making the results more transparent and understandable.

The AI-powered heart disease assessment app demonstrates the effective use of modern web technologies and machine learning to create a practical tool for health risk assessment. It combines Python's powerful data processing capabilities with the interactivity and accessibility provided by Streamlit, offering users a valuable resource for understanding and managing their heart disease risk.

8. Conclusion and Recommendation

Conclusion

The AI-Powered Heart Disease Risk Assessment App demonstrates the effective integration of advanced machine learning techniques and user-friendly web technologies to provide a practical tool for cardiovascular health assessment. The app leverages data from the Behavioral Risk Factor Surveillance System (BRFSS) 2022, ensuring that the analysis is based on a large, representative, and up-to-date dataset. By focusing on key risk factors such as age, BMI, physical activity, smoking status, and medical history, the app provides users with personalized risk scores and actionable recommendations to mitigate their heart disease risk. The use of SHAP values for model interpretation further enhances the transparency and trustworthiness of the app's predictions.

Recommendations

- Enhanced Hyperparameter Tuning: While initial hyperparameter tuning has been conducted, further experiments and refinements are planned to identify the optimal settings for the best model. Additional time will be dedicated to exploring a wider range of hyperparameters and employing advanced tuning techniques to enhance model performance.
- Classification Threshold Tuning: To achieve a better balance between false positives and true positives, we will focus on tuning the classification thresholds. This will help in optimizing the trade-off between sensitivity and specificity, ensuring more accurate predictions.
- Feature Selection Improvements: Further efforts will be made to refine the feature selection process. By carefully analyzing and selecting the most relevant features, we aim to improve the overall performance of the model. This includes experimenting with different feature selection techniques to enhance the predictive power of our app.

References

- Akiba, T., Sano, S., Yanase, T., Ohta, T., & Koyama, M. (2019). Optuna: A Next-generation Hyperparameter Optimization Framework. arXiv preprint arXiv:1907.10902. Retrieved from https://arxiv.org/abs/1907.10902
- Brownlee, J. (2019). Information Gain and Mutual Information for Machine Learning. Retrieved from MachineLearningMastery.com
- Datacamp. (2023). An Introduction to SHAP Values and Machine Learning Interpretability. Retrieved from https://www.datacamp.com/community/tutorials/introduction-shap-values-machine-learning
- Lundberg, S. M., & Lee, S.-I. (2017). A Unified Approach to Interpreting Model Predictions. arXiv preprint arXiv:1705.07874. Retrieved from https://arxiv.org/abs/1705.07874
- pandas.pydata.org. (2023). Categorical data — pandas 2.2.2 documentation. Retrieved from https://pandas.pydata.org/pandas-docs/stable/user_guide/categorical.html
- Prokhorenkova, L., Gusev, G., Vorobev, A., Dorogush, A. V., & Gulin, A. (2018). CatBoost: unbiased boosting with categorical features. Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS 2018), 6638–6648. Retrieved from https://proceedings.neurips.cc/paper_files/paper/2018/file/14491b756b3a51daac41c24863285549-Paper.pdf
- Saar-Tsechansky, M., & Provost, F. (2007). Handling missing values when applying classification models. Journal of Machine Learning Research, 8, 1625-1657. Retrieved from https://jmlr.csail.mit.edu/papers/volume8/saar-tsechansky07a/saar-tsechansky07a.pdf

- Uyar, A., Polat, H., & Deveci, M. (2022). Heart disease prediction using distinct artificial intelligence techniques: performance analysis and comparison. Iran Journal of Computer Science. Retrieved from https://link.springer.com/article/10.1007/s42044-022-00090-7

Appendix:

Data Wrangling and Pre-processing Code

Exploratory Data Analysis Code

Modeling Code

App Code

Data Wrangling: AI-Powered Heart Disease Risk Assessment

- **Name:** Aktham Almomani
- **Course:** Probability and Statistics for Artificial Intelligence (MS-AAI-500-02) / University Of San Diego
- **Semester:** Summer 2024
- **Group:** 8

Contents

- Column 6: Have_Personal_Health_Care_Provider
- Column 7: Could_Not_Afford_To_See_Doctor
- Column 8: Length_of_time_since_last_routine_checkup
- Column 9: Ever_Diagnosed_with_Heart_Attack
- Column 10: Ever_Diagnosed_with_a_Stroke
- Column 11: Ever_told_you_had_a_depressive_disorder
- Column 12: Ever_told_you_have_kidney_disease
- Column 13: Ever_told_you_had_diabetes
- Column 14: Computed_body_mass_index_categories
- Column 15: Difficulty_Walking_or_Climbing_Stairs
- Column 16: Computed_Physical_Health_Status
- Column 17: Computed_Mental_Health_Status
- Column 18: Computed_Asthma_Status
- Column 19: Exercise_in_Past_30_Days
- Column 20: Computed_Smoking_Status
- Column 21: Binge_Drinking_Calculated_Variable
- Column 22: How_Much_Time_Do_You_Sleep
- Column 23: Computed_number_of_drinks_of_alcohol_beverages_per_week

- Dropping unnecessary columns
- Review final structure of the cleaned dataframe
- Saving the cleaned dataframe

Introduction

Contents

In this notebook, I have undertaken a series of data wrangling steps to prepare our dataset for analysis. **Data wrangling** is a crucial step in the data science process, involving the transformation and mapping of raw data into a more usable format. Here's a summary of the key steps taken in this notebook:

- **Dealing with Missing Data:** Identified and imputed missing values in critical columns, such as the gender column, ensuring the dataset's completeness.
- **Data Mapping:** Transformed categorical variables into more meaningful representations, making the data easier to analyze and interpret.
- **Data Cleaning:** Removed or corrected inconsistent and erroneous entries to improve data quality.
- **Feature Engineering:** Created new features that may enhance the predictive power of our models. These steps are essential for building a reliable and robust model for heart disease prediction.

Dataset

Contents

- The Behavioral Risk Factor Surveillance System (BRFSS) is the nation's premier system of health-related telephone surveys that collect state data about U.S. residents regarding their health-related risk behaviors, chronic health conditions, and use of preventive services. Established in 1984 with 15 states, BRFSS now collects data in all 50 states as well as the District of Columbia and three U.S. territories. CDC BRFSS completes more than 400,000 adult interviews each year, making it the largest continuously conducted health survey system in the world.
- The dataset was sourced from Kaggle (Behavioral Risk Factor Surveillance System (BRFSS) 2022) and it was originally downloaded from the CDC BRFSS 2022 website.
- To get more understanding regarding the dataset, please go to the data_directory folder in my Github.

Setup and preliminaries

Contents

Import libraries

Contents

[1]:
```
#Let's import the necessary packages:
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import math
import scipy.stats as stats
from scipy.stats import gamma, linregress
from bs4 import BeautifulSoup
import re
from fancyimpute import KNN
import dask.dataframe as dd

# Let's run below to customize notebook display:
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)

# format floating-point numbers to 2 decimal places: we'll adjust below requirement as needed for specific answers during this assignment:
pd.set_option('float_format', '{:.2f}'.format)
```

Necessary functions

Contents

[2]:
```
def summarize_df(df):
    """
    Generate a summary DataFrame for an input DataFrame.
    Parameters:
    df (pd.DataFrame): The DataFrame to summarize.
    Returns:
    A datafram: containing the following columns:
              - 'unique_count': No. unique values in each column.
              - 'data_types': Data types of each column.
              - 'missing_counts': No. of missing (NaN) values in each column.
              - 'missing_percentage': Percentage of missing values in each column.
    """
    # No. of unique values for each column:
    unique_counts = df.nunique()
    # Data types of each column:
    data_types = df.dtypes
    # No. of missing (NaN) values in each column:
    missing_counts = df.isnull().sum()
    # Percentage of missing values in each column:
    missing_percentage = 100 * df.isnull().mean()
    # Concatenate the above metrics:
    summary_df = pd.concat([unique_counts, data_types, missing_counts, missing_percentage], axis=1)
    # Rename the columns for better readibility
    summary_df.columns = ['unique_count', 'data_types', 'missing_counts', 'missing_percentage']
    # Return summary df
    return summary_df
#-----------------------------------------------------------------------------------------------------------------#
# Function to clean and format the label
def clean_label(label):
    # Replace any non-alphabetic or non-numeric characters with nothing
    label = re.sub(r'[^a-zA-Z0-9\s]', '', label)
    # Replace spaces with underscores
    label = re.sub(r'\s+', '_', label)
    return label
#-----------------------------------------------------------------------------------------------------------------#
```

```
# Function to impute missing values based on distribution
def impute_missing(row):
    if pd.isna(row['Are_you_male_or_female_3']):
        return np.random.choice(value_counts.index, p=value_counts.values)
    else:
        return row['Are_you_male_or_female_3']


#-----------------------------------------------------------------------------------------------------------------#
def value_counts_with_percentage(df, column_name):
    # Calculate value counts
    counts = df[column_name].value_counts(dropna=False)

    # Calculate percentages
    percentages = df[column_name].value_counts(dropna=False, normalize=True) * 100

    # Combine counts and percentages into a DataFrame
    result = pd.DataFrame({
        'Count': counts,
        'Percentage': percentages
    })

    return result
```

Extracting descriptive column Names for the dataset

Contents

The Behavioral Risk Factor Surveillance System (BRFSS) dataset available on Kaggle, found here, contains a wealth of information collected through surveys. However, the column names in the dataset are represented by short labels or codes (e.g., _STATE, FMONTH, IDATE), which can be difficult to interpret without additional context.

To ensure we fully understand what each column in the dataset represents, it is crucial to replace these short codes with their corresponding descriptive names. These descriptive names provide clear insights into the type of data each column holds, making the dataset easier to understand and analyze.

Process Overview:

- **Identify the Source for Descriptive Names:** The descriptive names corresponding to these short labels are typically documented in the codebook in HTML or metadata provided by the data collection authority. In this case, the descriptive names are found in an HTML document provided by the BRFSS.
- **Parse the HTML Document:** Using web scraping techniques, such as BeautifulSoup in Python, we can parse the HTML document to extract the relevant information. Specifically, we look for tables or sections in the HTML that list the short labels alongside their descriptive names.
- **Match and Replace:** We create a mapping of short labels to their descriptive names. This mapping is then applied to our dataset to replace the short labels with more meaningful descriptive names.
- **Save the Enhanced Dataset:** The dataset with descriptive column names is saved for subsequent analysis, ensuring that all users can easily interpret the columns.

[3]:
```
# Path to the HTML file:
file_path = 'USCODE22_LLCP_102523.HTML'

# Read the HTML file:
with open(file_path, 'r', encoding='windows-1252') as file:
    html_content = file.read()

# Parse the HTML content using BeautifulSoup:
soup = BeautifulSoup(html_content, 'html.parser')

# Find all the tables that contain the required information:
tables = soup.find_all('table', class_='table')

# Initialize lists to store the extracted data:
labels = []
sas_variable_names = []
```

```
# Loop through each table to extract 'Label' and 'SAS Variable Name':
for table in tables:
    # Find all 'td' elements in the table:
    cells = table.find_all('td', class_='l m linecontent')

    # Loop through each cell to find 'Label' and 'SAS Variable Name':
    for cell in cells:
        text = cell.get_text(separator="\n")
        label = None
        sas_variable_name = None
        for line in text.split('\n'):
            if line.strip().startswith('Label:'):
                label = line.split('Label:')[1].strip()
            elif line.strip().startswith('SAS\xa0Variable\xa0Name:'):
                sas_variable_name = line.split('SAS\xa0Variable\xa0Name:')[1].strip()
        if label and sas_variable_name:
            labels.append(label)
            sas_variable_names.append(sas_variable_name)
        else:
            print("Label or SAS Variable Name not found in the text:")
            print(text)

# Create a DataFrame:
data = {'SAS Variable Name': sas_variable_names, 'Label': labels}
cols_df = pd.DataFrame(data)

# Save the DataFrame to a CSV file:
output_file_path = 'extracted_data.csv'
cols_df.to_csv(output_file_path, index=False)

print(f"Data has been successfully extracted and saved to {output_file_path}")

cols_df.head()
```

```
Data has been successfully extracted and saved to extracted_data.csv
```

[3]:

	SAS Variable Name	Label
0	_STATE	State FIPS Code
1	FMONTH	File Month
2	IDATE	Interview Date
3	IMONTH	Interview Month
4	IDAY	Interview Day

[4]:
```
#Let's run below to examin each features again missing data count & percentage, unique count, data types:
summarize_df(cols_df)
```

[4]:

	unique_count	data_types	missing_counts	missing_percentage
SAS Variable Name	324	object	0	0.00
Label	317	object	0	0.00

No Missing Data - looks like we have 324 columns

Importing dataset

Contents

[5]:
```
#First, let's load the main dataset BRFSS 2022:
df = pd.read_csv('brfss2022.csv')
```

Validating the dataset

Contents

[6]:
```
# Now, Let's Look at the top 5 rows of the df:
df.head()
```

[6]:

	_STATE	FMONTH	IDATE	IMONTH	IDAY	IYEAR	DISPCODE	SEQNO	_PSU	CTELENM1	PVTRESD1	COLGHOUS	STATERE1	CELPHON1	LADULT1
0	1.00	1.00	2032022	2	3	2022	1100.00	2022000001	2022000001.00	1.00	1.00	NaN	1.00	2.00	1.00
1	1.00	1.00	2042022	2	4	2022	1100.00	2022000002	2022000002.00	1.00	1.00	NaN	1.00	2.00	1.00
2	1.00	1.00	2022022	2	2	2022	1100.00	2022000003	2022000003.00	1.00	1.00	NaN	1.00	2.00	1.00
3	1.00	1.00	2032022	2	3	2022	1100.00	2022000004	2022000004.00	1.00	1.00	NaN	1.00	2.00	1.00
4	1.00	1.00	2022022	2	2	2022	1100.00	2022000005	2022000005.00	1.00	1.00	NaN	1.00	2.00	1.00

[7]:
```
#now, Let's Look at the shape of df:
shape = df.shape
print("Number of rows:", shape[0], "\nNumber of columns:", shape[1])
```

```
Number of rows: 445132
Number of columns: 326
```

Correcting dataset column names

Contents

To replace the SAS Variable Names in your dataset with the corresponding labels (where spaces in the labels are replaced with underscores), you can follow these steps:

- Create a mapping from the SAS Variable Names to the modified labels.
- Use this mapping to rename the columns in your dataset.

[8]:
```
# Function to clean and format the label
def clean_label(label):
    # Replace any non-alphabetic or non-numeric characters with nothing
    label = re.sub(r'[^a-zA-Z0-9\s]', '', label)
    # Replace spaces with underscores
    label = re.sub(r'\s+', '_', label)
    return label

# Create a dictionary for mapping SAS Variable Names to cleaned labels
mapping = {row['SAS Variable Name']: clean_label(row['Label']) for _, row in cols_df.iterrows()}

# Print the mapping dictionary to verify the changes
#print("Column Renaming Mapping:")
#for k, v in mapping.items():
#    print(f"{k}: {v}")
# Rename the columns in the actual data DataFrame
df.rename(columns=mapping, inplace=True)
df.head()
```

[8]:

	State_FIPS_Code	File_Month	Interview_Date	Interview_Month	Interview_Day	Interview_Year	Final_Disposition	Annual_Sequence_Number	Primary_Sampling_Unit
0	1.00	1.00	2032022	2	3	2022	1100.00	2022000001	2022000001.00
1	1.00	1.00	2042022	2	4	2022	1100.00	2022000002	2022000002.00
2	1.00	1.00	2022022	2	2	2022	1100.00	2022000003	2022000003.00
3	1.00	1.00	2032022	2	3	2022	1100.00	2022000004	2022000004.00
4	1.00	1.00	2022022	2	2	2022	1100.00	2022000005	2022000005.00

Heart Disease related features ¶

Contents

After several days of research and analysis of the dataset's features, we have identified the following key features for heart disease assessment:

- **Target Variable (Dependent Variable):**
  - Heart_disease: "Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease"
- **Demographics:**
  - Gender: Are_you_male_or_female
  - Race: Computed_race_groups_used_for_internet_prevalence_tables
  - Age: Imputed_Age_value_collapsed_above_80
- **Medical History:**
  - General_Health
  - Have_Personal_Health_Care_Provider
  - Could_Not_Afford_To_See_Doctor
  - Length_of_time_since_last_routine_checkup
  - Ever_Diagnosed_with_Heart_Attack
  - Ever_Diagnosed_with_a_Stroke
  - Ever_told_you_had_a_depressive_disorder
  - Ever_told_you_have_kidney_disease
  - Ever_told_you_had_diabetes
  - Reported_Weight_in_Pounds
  - Reported_Height_in_Feet_and_Inches
  - Computed_body_mass_index_categories
  - Difficulty_Walking_or_Climbing_Stairs
  - Computed_Physical_Health_Status
  - Computed_Mental_Health_Status
  - Computed_Asthma_Status
- **Life Style:**
  - Leisure_Time_Physical_Activity_Calculated_Variable
  - Smoked_at_Least_100_Cigarettes
  - Computed_Smoking_Status
  - Binge_Drinking_Calculated_Variable
  - Computed_number_of_drinks_of_alcohol_beverages_per_week
  - Exercise_in_Past_30_Days
  - How_Much_Time_Do_You_Sleep

Selection Heart disease related features

Contents

[9]:
```
#Here, let's seelect the main features directly related to heart disease:
df = df[["Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease", # Target Variable
         "Are_you_male_or_female", #Demographics
         "Computed_race_groups_used_for_internet_prevalence_tables",#Demographics
         "Imputed_Age_value_collapsed_above_80",#Demographics
         "General_Health", #Medical History
         "Have_Personal_Health_Care_Provider",#Medical History
         "Could_Not_Afford_To_See_Doctor",#Medical History
         "Length_of_time_since_last_routine_checkup",#Medical History
         "Ever_Diagnosed_with_Heart_Attack",#Medical History
         "Ever_Diagnosed_with_a_Stroke",#Medical History
         "Ever_told_you_had_a_depressive_disorder",#Medical History
         "Ever_told_you_have_kidney_disease",#Medical History
         "Ever_told_you_had_diabetes",#Medical History
         "Reported_Weight_in_Pounds",#Medical History
         "Reported_Height_in_Feet_and_Inches",#Medical History
         "Computed_body_mass_index_categories",#Medical History
         "Difficulty_Walking_or_Climbing_Stairs",#Medical History
         "Computed_Physical_Health_Status",#Medical History
         "Computed_Mental_Health_Status",#Medical History
         "Computed_Asthma_Status",#Medical History
         "Leisure_Time_Physical_Activity_Calculated_Variable",#Life Style
         "Smoked_at_Least_100_Cigarettes",#Life Style
         "Computed_Smoking_Status",#Life Style
         "Binge_Drinking_Calculated_Variable",#Life Style
         "Computed_number_of_drinks_of_alcohol_beverages_per_week",#Life Style
         "Exercise_in_Past_30_Days",#Life Style
         "How_Much_Time_Do_You_Sleep"#Life Style
        ]]
df.head()
```

[9]:

	Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease	Are_you_male_or_female	Are_you_male_or_female	Are_you_male_or_female	Are_you_male_or_female	Compute
0	2.00	NaN	NaN	NaN	NaN	
1	2.00	NaN	NaN	NaN	NaN	
2	2.00	NaN	2.00	NaN	NaN	
3	2.00	NaN	NaN	NaN	NaN	
4	2.00	NaN	NaN	NaN	NaN	

[10]:
```
#now, Let's Look at the shape of df after features selection:
shape = df.shape
print("Number of rows:", shape[0], "\nNumber of columns:", shape[1])
```

```
Number of rows: 445132
Number of columns: 30
```

Imputing Missing Data, Transforming Columns and Features Engineering ¶

Contents

In this step, we address missing data, map categorical values, and rename columns for improved data quality and clarity. The key actions taken are as follows:

- Replace Specific Values with NaN: Identify and replace erroneous or placeholder values with NaN to standardize missing data representation.
- Calculate Value Distribution: Determine the distribution of existing values to understand the data's baseline state.
- Impute Missing Values: Use a function to impute missing values based on the calculated distribution, ensuring the data remains representative of its original characteristics.
- Map Categorical Values: Apply a mapping dictionary to convert numeric codes into meaningful categorical labels.
- Rename Columns: Update column names to reflect their contents accurately and improve dataset readability.
- Feature Engineering: Created new features that may enhance the predictive power of our models. These steps are essential for building a reliable and robust model for heart disease prediction.

Distribution-Based Imputation

Contents To deal with missing data in this project, we'll be using **Distribution-Based Imputation**:

- **Introduction**
  - Distribution-Based: The imputation process relies on the existing distribution of the categories in the dataset.
  - Imputation: The act of filling in missing values.
- **Why This Method Works**
  - Preserves Original Distribution: By using the observed proportions to guide the imputation, the method maintains the original distribution of gender categories.
  - Random Imputation: Randomly selecting values based on the existing distribution prevents systematic biases that could arise from deterministic imputation methods.
  - Scalability: This approach can be easily scaled to larger datasets and applied to other categorical variables with missing values.
- **Advantages**
  - Bias Minimization: Ensures that the imputed values do not skew the dataset in favor of any particular category.
  - Simplicity: The method is straightforward to implement and understand.
  - Flexibility: Can be adapted to any categorical variable with missing values.

This method is particularly useful in scenarios where preserving the natural distribution of data is crucial for subsequent analysis or modeling tasks.

[11]:
```
#Let's run below to examin each features again missing data count & percentage, unique count, data types:
summarize_df(df)
```

[11]:

	unique_count	data_types	missing_counts	missing_percentage
Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease	4	float64	2	0.00
Are_you_male_or_female	2	float64	445111	100.00
Are_you_male_or_female	6	float64	401436	90.18
Are_you_male_or_female	5	float64	96053	21.58
Are_you_male_or_female	4	float64	365705	82.16
Computed_race_groups_used_for_internet_prevalence_tables	7	float64	0	0.00
Imputed_Age_value_collapsed_above_80	63	float64	0	0.00
General_Health	7	float64	3	0.00
Have_Personal_Health_Care_Provider	5	float64	2	0.00

Column 1: Are_you_male_or_female

Contents

We have 4 versions of the same column, so now let's keep the least columns with missing data `21.58`

[12]:
```
# Let's get the column names in a list:
print(df.columns)
```

```
Index(['Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease',
       'Are_you_male_or_female', 'Are_you_male_or_female',
       'Are_you_male_or_female', 'Are_you_male_or_female',
       'Computed_race_groups_used_for_internet_prevalence_tables',
       'Imputed_Age_value_collapsed_above_80', 'General_Health',
       'Have_Personal_Health_Care_Provider', 'Could_Not_Afford_To_See_Doctor',
       'Length_of_time_since_last_routine_checkup',
       'Ever_Diagnosed_with_Heart_Attack', 'Ever_Diagnosed_with_a_Stroke',
       'Ever_told_you_had_a_depressive_disorder',
       'Ever_told_you_have_kidney_disease', 'Ever_told_you_had_diabetes',
       'Reported_Weight_in_Pounds', 'Reported_Height_in_Feet_and_Inches',
       'Computed_body_mass_index_categories',
       'Difficulty_Walking_or_Climbing_Stairs',
       'Computed_Physical_Health_Status', 'Computed_Mental_Health_Status',
       'Computed_Asthma_Status',
       'Leisure_Time_Physical_Activity_Calculated_Variable',
       'Smoked_at_Least_100_Cigarettes', 'Computed_Smoking_Status',
       'Binge_Drinking_Calculated_Variable',
       'Computed_number_of_drinks_of_alcohol_beverages_per_week',
       'Exercise_in_Past_30_Days', 'How_Much_Time_Do_You_Sleep'],
      dtype='object')
```

[13]:
```
#Let's select the main features related to heart disease:
df.columns = ['Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease', # This is my target variable!!!
       'Are_you_male_or_female_1', 'Are_you_male_or_female_2',
       'Are_you_male_or_female_3', 'Are_you_male_or_female_4',
       'Computed_race_groups_used_for_internet_prevalence_tables',
       'Imputed_Age_value_collapsed_above_80', 'General_Health',
       'Have_Personal_Health_Care_Provider', 'Could_Not_Afford_To_See_Doctor',
       'Length_of_time_since_last_routine_checkup',
       'Ever_Diagnosed_with_Heart_Attack', 'Ever_Diagnosed_with_a_Stroke',
       'Ever_told_you_had_a_depressive_disorder',
       'Ever_told_you_have_kidney_disease', 'Ever_told_you_had_diabetes',
       'Reported_Weight_in_Pounds', 'Reported_Height_in_Feet_and_Inches',
       'Computed_body_mass_index_categories',
       'Difficulty_Walking_or_Climbing_Stairs',
       'Computed_Physical_Health_Status', 'Computed_Mental_Health_Status',
       'Computed_Asthma_Status',
       'Leisure_Time_Physical_Activity_Calculated_Variable',
       'Smoked_at_Least_100_Cigarettes', 'Computed_Smoking_Status',
       'Binge_Drinking_Calculated_Variable',
       'Computed_number_of_drinks_of_alcohol_beverages_per_week',
       'Exercise_in_Past_30_Days', 'How_Much_Time_Do_You_Sleep']
#Let's run below to examin each features again missing data count & percentage, unique count, data types:
summarize_df(df)
```

[13]:

	unique_count	data_types	missing_counts	missing_percentage
Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease	4	float64	2	0.00
Are_you_male_or_female_1	2	float64	445111	100.00
Are_you_male_or_female_2	6	float64	401436	90.18
Are_you_male_or_female_3	5	float64	96053	21.58
Are_you_male_or_female_4	4	float64	365705	82.16
Computed_race_groups_used_for_internet_prevalence_tables	7	float64	0	0.00
Imputed_Age_value_collapsed_above_80	63	float64	0	0.00
General_Health	7	float64	3	0.00
Have_Personal_Health_Care_Provider	5	float64	2	0.00

Alright, as we can see above, now, let's drop 'Are_you_male_or_female_1', 'Are_you_male_or_female_2' and 'Are_you_male_or_female_4'

[14]:
```
#Let's drop the unnecessary columns:
columns_to_drop = ['Are_you_male_or_female_1', 'Are_you_male_or_female_2', 'Are_you_male_or_female_4']
df = df.drop(columns=columns_to_drop)

#Let's run below to examin each features again missing data count & percentage, unique count, data types:
summarize_df(df)
```

[14]:

	unique_count	data_types	missing_counts	missing_percentage
Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease	4	float64	2	0.00
Are_you_male_or_female_3	5	float64	96053	21.58
Computed_race_groups_used_for_internet_prevalence_tables	7	float64	0	0.00
Imputed_Age_value_collapsed_above_80	63	float64	0	0.00
General_Health	7	float64	3	0.00
Have_Personal_Health_Care_Provider	5	float64	2	0.00
Could_Not_Afford_To_See_Doctor	4	float64	4	0.00
Length_of_time_since_last_routine_checkup	7	float64	3	0.00
Ever_Diagnosed_with_Heart_Attack	4	float64	4	0.00

[15]:
```
# view columns count:
df.Are_you_male_or_female_3.value_counts(dropna=False)
```

[15]:
```
2.00    174948
1.00    173639
NaN      96053
3.00       328
9.00       113
7.00        51
Name: Are_you_male_or_female_3, dtype: int64
```

Are_you_male_or_female_3:

- 2: Femal
- 1: Male
- 3: Nonbinary
- 7: Don't know/Not Sure
- 9: Refused

So based on above, let's change 7 and 9 to nan

[16]:
```
# Replace 7 and 9 with NaN
df['Are_you_male_or_female_3'].replace([7, 9], np.nan, inplace=True)
df.Are_you_male_or_female_3.value_counts(dropna=False)
```

[16]:
```
2.00    174948
1.00    173639
NaN      96217
3.00       328
Name: Are_you_male_or_female_3, dtype: int64
```

[17]:
```
# Calculate the distribution of existing values
value_counts = df['Are_you_male_or_female_3'].value_counts(normalize=True, dropna=True)
print("Original distribution:\n", value_counts)
```

```
Original distribution:
 2.00   0.50
1.00   0.50
3.00   0.00
Name: Are_you_male_or_female_3, dtype: float64
```

[18]:
```
# Function to impute missing values based on distribution
def impute_missing_gender(row):
    if pd.isna(row['Are_you_male_or_female_3']):
        return np.random.choice(value_counts.index, p=value_counts.values)
    else:
        return row['Are_you_male_or_female_3']

# Apply the imputation function
df['Are_you_male_or_female_3'] = df.apply(impute_missing_gender, axis=1)
```

[19]:
```
# Verify the imputation
imputed_value_counts = df['Are_you_male_or_female_3'].value_counts(dropna=False) # normalize=True
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 2.00    223217
1.00    221503
3.00       412
Name: Are_you_male_or_female_3, dtype: int64
```

Alright, as we can see above, no missing data on this column and the proportions reserved (Random imputation worked as expected).

[20]:
```
# Create a mapping dictionary:
gender_mapping = {2: 'female', 1: 'male', 3: 'nonbinary'}

# Apply the mapping to the gender column:
df['Are_you_male_or_female_3'] = df['Are_you_male_or_female_3'].map(gender_mapping)

# Rename the column:
df.rename(columns={'Are_you_male_or_female_3': 'gender'}, inplace=True)

df.head()
```

[20]:

	Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease	gender	Computed_race_groups_used_for_internet_prevalence_tables	Imputed_Age_value_collapsed_above_80	G
0	2.00	female	1.00	80.00	
1	2.00	male	1.00	80.00	
2	2.00	male	1.00	56.00	
3	2.00	female	1.00	73.00	
4	2.00	male	1.00	43.00	

[21]:
```
#Let's run below to examin each features again missing data count & percentage, unique count, data types:
summarize_df(df)
```

[21]:

	unique_count	data_types	missing_counts	missing_percentage
Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease	4	float64	2	0.00
gender	3	object	0	0.00
Computed_race_groups_used_for_internet_prevalence_tables	7	float64	0	0.00
Imputed_Age_value_collapsed_above_80	63	float64	0	0.00
General_Health	7	float64	3	0.00
Have_Personal_Health_Care_Provider	5	float64	2	0.00
Could_Not_Afford_To_See_Doctor	4	float64	4	0.00
Length_of_time_since_last_routine_checkup	7	float64	3	0.00
Ever_Diagnosed_with_Heart_Attack	4	float64	4	0.00

Column 2: Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease

Contents

[22]:
```
#view column counts:
df.Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease.value_counts(dropna=False)
```

[22]:
```
2.00    414176
1.00     26551
7.00      4044
9.00       359
NaN          2
Name: Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease, dtype: int64
```

Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease:

- 2: No
- 1: Yes
- 7: Don't know/Not Sure
- 9: Refused

Alright, so next let's change 7 and 9 to nan:

[23]:
```
# Replace 7 and 9 with NaN
df['Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease'].replace([7, 9], np.nan, inplace=True)
df.Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease.value_counts(dropna=False)
```

[23]:
```
2.00    414176
1.00     26551
NaN       4405
Name: Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease, dtype: int64
```

Alright, again, let's use **Distribution-Based Imputation** for the above missing data:

[24]:
```
# Calculate the distribution of existing values
value_counts = df['Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease'].value_counts(normalize=True, dropna=True)
print("Original distribution:\n", value_counts)
```

```
Original distribution:
 2.00   0.94
1.00   0.06
Name: Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease, dtype: float64
```

[25]:
```
# Function to impute missing values based on distribution
def impute_missing(row):
    if pd.isna(row['Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease']):
        return np.random.choice(value_counts.index, p=value_counts.values)
    else:
        return row['Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease']
```

[26]:
```
# Apply the imputation function
df['Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease'] = df.apply(impute_missing, axis=1)
```

[27]:
```
# Verify the imputation
imputed_value_counts = df['Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease'].value_counts(dropna=False) # normalize=True
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 2.00    418331
1.00     26801
Name: Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease, dtype: int64
```

[28]:
```
# Verify the imputation
imputed_value_counts = df['Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease'].value_counts(dropna=False, normalize=True) #
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 2.00   0.94
1.00   0.06
Name: Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease, dtype: float64
```

[29]:
```
# Create a mapping dictionary:
heart_disease_mapping = {2: 'no', 1: 'yes'}

# Apply the mapping to the "Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease" column:
df['Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease'] = df['Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease'].map(heart_disease_mapping)

# Rename the column:
df.rename(columns={'Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease': 'heart_disease'}, inplace=True)
```

[30]:
```
#Let's run below to examin each features again missing data count & percentage, unique count, data types:
summarize_df(df)
```

General_Health	7	float64	3	0.00
Have_Personal_Health_Care_Provider	5	float64	2	0.00
Could_Not_Afford_To_See_Doctor	4	float64	4	0.00
Length_of_time_since_last_routine_checkup	7	float64	3	0.00
Ever_Diagnosed_with_Heart_Attack	4	float64	4	0.00
Ever_Diagnosed_with_a_Stroke	4	float64	2	0.00
Ever_told_you_had_a_depressive_disorder	4	float64	7	0.00
Ever_told_you_have_kidney_disease	4	float64	2	0.00
Ever_told_you_had_diabetes	6	float64	3	0.00
Reported_Weight_in_Pounds	619	float64	15901	3.57

Column 3: Computed_race_groups_used_for_internet_prevalence_tables

Contents

[31]:
```
#view column counts:
df.Computed_race_groups_used_for_internet_prevalence_tables.value_counts(dropna=False)
```

[31]:
```
1.00    333514
7.00     42977
2.00     35876
4.00     13487
6.00      9744
3.00      7120
5.00      2414
Name: Computed_race_groups_used_for_internet_prevalence_tables, dtype: int64
```

Alright, so good news is there's no missing data in this column

Computed_race_groups_used_for_internet_prevalence_tables:

- 1: white_only_non_hispanic
- 2: black_only_non_hispanic
- 3: american_indian_or_alaskan_native_only_non_hispanic
- 4: asian_only_non_hispanic
- 5: native_hawaiian_or_other_pacific_islander_only_non_hispanic
- 6: multiracial_non_hispanic
- 7: hispanic

[32]:
```
# Create a mapping dictionary:
race_mapping = {1: 'white_only_non_hispanic',
2: 'black_only_non_hispanic',
3: 'american_indian_or_alaskan_native_only_non_hispanic',
4: 'asian_only_non_hispanic',
5: 'native_hawaiian_or_other_pacific_islander_only_non_hispanic',
6: 'multiracial_non_hispanic',
7: 'hispanic'}

# Apply the mapping to the race column:
df['Computed_race_groups_used_for_internet_prevalence_tables'] = df['Computed_race_groups_used_for_internet_prevalence_tables'].map(race_mapping)

# Rename the column:
df.rename(columns={'Computed_race_groups_used_for_internet_prevalence_tables': 'race'}, inplace=True)
```

[33]:
```
#view column counts:
df.race.value_counts(dropna=False)
```

[33]:
```
white_only_non_hispanic                                333514
hispanic                                                42977
black_only_non_hispanic                                 35876
asian_only_non_hispanic                                 13487
multiracial_non_hispanic                                 9744
american_indian_or_alaskan_native_only_non_hispanic      7120
native_hawaiian_or_other_pacific_islander_only_non_hispanic    2414
Name: race, dtype: int64
```

column 4: Imputed_Age_value_collapsed_above_80

Contents

[34]:
```
#view column counts:
df.Imputed_Age_value_collapsed_above_80.value_counts(dropna=False)
```

[34]:
```
80.00    36253
65.00    10421
70.00    10371
67.00     9652
68.00     9351
62.00     9262
72.00     9212
66.00     9191
64.00     9174
60.00     9155
69.00     9027
75.00     8920
63.00     8874
71.00     8686
73.00     8404
74.00     8267
52.00     8256
61.00     8216
```

[35]:
```
# Define bins and labels:
bins = [17, 24, 29, 34, 39, 44, 49, 54, 59, 64, 69, 74, 79, 99]
labels = [
    'Age_18_to_24', 'Age_25_to_29', 'Age_30_to_34', 'Age_35_to_39',
    'Age_40_to_44', 'Age_45_to_49', 'Age_50_to_54', 'Age_55_to_59',
    'Age_60_to_64', 'Age_65_to_69', 'Age_70_to_74', 'Age_75_to_79',
    'Age_80_or_older'
        ]
```

[36]:
```
# Categorize the age values into bins:
df['age_category'] = pd.cut(df['Imputed_Age_value_collapsed_above_80'], bins=bins, labels=labels, right=True)
df.age_category.value_counts(dropna=False)
```

[36]:
```
Age_65_to_69       47642
Age_70_to_74       44940
Age_60_to_64       44681
Age_55_to_59       38059
Age_50_to_54       37404
Age_80_or_older    36253
Age_75_to_79       32616
Age_40_to_44       30403
Age_45_to_49       29580
Age_35_to_39       28771
Age_18_to_24       26943
Age_30_to_34       25840
Age_25_to_29       22000
Name: age_category, dtype: int64
```

Column 5: General_Health

Contents

[37]:
```
#view column counts:
df.General_Health.value_counts(dropna=False)
```

[37]:
```
2.00    148444
3.00    143598
1.00     71878
4.00     60273
5.00     19741
7.00       810
9.00       385
NaN          3
Name: General_Health, dtype: int64
```

General_Health:

- 1: excellent
- 2: very_good
- 3: good
- 4: fair
- 5: poor
- 7: dont_know
- 9: refused

so for 7, 9 let's convert to nan:

[38]:
```
# Replace 7 and 9 with NaN
df['General_Health'].replace([7, 9], np.nan, inplace=True)
df.General_Health.value_counts(dropna=False)
```

[38]:
```
2.00    148444
3.00    143598
1.00     71878
4.00     60273
5.00     19741
NaN       1198
Name: General_Health, dtype: int64
```

[39]:
```
# Calculate the distribution of existing values
value_counts = df['General_Health'].value_counts(normalize=True, dropna=True)
print("Original General_Health:\n", value_counts)
```

```
Original General_Health:
 2.00   0.33
3.00   0.32
1.00   0.16
4.00   0.14
5.00   0.04
Name: General_Health, dtype: float64
```

[40]:
```
# Function to impute missing values based on distribution
def impute_missing(row):
    if pd.isna(row['General_Health']):
        return np.random.choice(value_counts.index, p=value_counts.values)
    else:
        return row['General_Health']
```

[41]:
```
# Apply the imputation function
df['General_Health'] = df.apply(impute_missing, axis=1)
```

[42]:
```
# Verify the imputation
imputed_value_counts = df['General_Health'].value_counts(dropna=False) # normalize=True
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 2.00    148843
3.00    143983
1.00     72064
4.00     60440
5.00     19802
Name: General_Health, dtype: int64
```

[43]:
```
# Create a mapping dictionary:
health_mapping = {1: 'excellent',
                  2: 'very_good',
                  3: 'good',
                  4: 'fair',
                  5: 'poor'
                 }


# Apply the mapping to the health column:
df['General_Health'] = df['General_Health'].map(health_mapping)

# Rename the column:
df.rename(columns={'General_Health': 'general_health'}, inplace=True)
```

[44]:
```
#view column counts:
df.general_health.value_counts(dropna=False)
```

[44]:
```
very_good    148843
good         143983
excellent     72064
fair          60440
poor          19802
Name: general_health, dtype: int64
```

Column 6: Have_Personal_Health_Care_Provider

Contents

[45]:
```
#view column counts:
df.Have_Personal_Health_Care_Provider.value_counts(dropna=False)
```

[45]:
```
1.00    246967
2.00    136685
3.00     57105
7.00      3270
9.00      1103
NaN          2
Name: Have_Personal_Health_Care_Provider, dtype: int64
```

Have_Personal_Health_Care_Provider:

- 1: yes_only_one
- 2: more_than_one
- 3: no
- 7: dont_know
- 9: refused

so for 7, 9 let's convert to nan:

[46]:
```
# Replace 7 and 9 with NaN
df['Have_Personal_Health_Care_Provider'].replace([7, 9], np.nan, inplace=True)
df.Have_Personal_Health_Care_Provider.value_counts(dropna=False)
```

[46]:
```
1.00    246967
2.00    136685
3.00     57105
NaN       4375
Name: Have_Personal_Health_Care_Provider, dtype: int64
```

[47]:
```
# Calculate the distribution of existing values
value_counts = df['Have_Personal_Health_Care_Provider'].value_counts(normalize=True, dropna=True)
print("Original Have_Personal_Health_Care_Provider:\n", value_counts)
```

```
Original Have_Personal_Health_Care_Provider:
 1.00   0.56
2.00   0.31
3.00   0.13
Name: Have_Personal_Health_Care_Provider, dtype: float64
```

[48]:
```
# Function to impute missing values based on distribution
def impute_missing(row):
    if pd.isna(row['Have_Personal_Health_Care_Provider']):
        return np.random.choice(value_counts.index, p=value_counts.values)
    else:
        return row['Have_Personal_Health_Care_Provider']
```

[49]:
```
# Apply the imputation function
df['Have_Personal_Health_Care_Provider'] = df.apply(impute_missing, axis=1)
```

[50]:
```
# Verify the imputation
imputed_value_counts = df['Have_Personal_Health_Care_Provider'].value_counts(dropna=False) # normalize=True
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 1.00    249393
2.00    138061
3.00     57678
Name: Have_Personal_Health_Care_Provider, dtype: int64
```

[51]:
```
# Verify the imputation
imputed_value_counts = df['Have_Personal_Health_Care_Provider'].value_counts(dropna=False,normalize=True) #
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 1.00   0.56
2.00   0.31
3.00   0.13
Name: Have_Personal_Health_Care_Provider, dtype: float64
```

[52]:
```
# Create a mapping dictionary:
porvider_mapping = {1: 'yes_only_one',
                    2: 'more_than_one',
                    3: 'no'
                   }

# Apply the mapping to the provider column:
df['Have_Personal_Health_Care_Provider'] = df['Have_Personal_Health_Care_Provider'].map(porvider_mapping)

# Rename the column:
df.rename(columns={'Have_Personal_Health_Care_Provider': 'health_care_provider'}, inplace=True)
```

Column 7: Could_Not_Afford_To_See_Doctor

Contents

[53]:
```
#view column counts:
df.Could_Not_Afford_To_See_Doctor.value_counts(dropna=False)
```

[53]:
```
2.00    406296
1.00     37227
7.00      1157
9.00       448
NaN          4
Name: Could_Not_Afford_To_See_Doctor, dtype: int64
```

Could_Not_Afford_To_See_Doctor:

- 1: yes
- 2: no
- 7: dont_know
- 9: refused

so for 7, 9 let's convert to nan:

[54]:
```
# Replace 7 and 9 with NaN
df['Could_Not_Afford_To_See_Doctor'].replace([7, 9], np.nan, inplace=True)
df.Could_Not_Afford_To_See_Doctor.value_counts(dropna=False)
```

[54]:
```
2.00    406296
1.00     37227
NaN       1609
Name: Could_Not_Afford_To_See_Doctor, dtype: int64
```

[55]:
```
# Calculate the distribution of existing values
value_counts = df['Could_Not_Afford_To_See_Doctor'].value_counts(normalize=True, dropna=True)
print("Original Could_Not_Afford_To_See_Doctor:\n", value_counts)
```

```
Original Could_Not_Afford_To_See_Doctor:
 2.00   0.92
1.00   0.08
Name: Could_Not_Afford_To_See_Doctor, dtype: float64
```

[56]:
```
# Function to impute missing values based on distribution
def impute_missing(row):
    if pd.isna(row['Could_Not_Afford_To_See_Doctor']):
        return np.random.choice(value_counts.index, p=value_counts.values)
    else:
        return row['Could_Not_Afford_To_See_Doctor']
```

[57]:
```
# Apply the imputation function
df['Could_Not_Afford_To_See_Doctor'] = df.apply(impute_missing, axis=1)
```

[58]:
```
# Verify the imputation
imputed_value_counts = df['Could_Not_Afford_To_See_Doctor'].value_counts(dropna=False) # normalize=True
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 2.00    407773
1.00     37359
Name: Could_Not_Afford_To_See_Doctor, dtype: int64
```

[59]:
```
# Verify the imputation
imputed_value_counts = df['Could_Not_Afford_To_See_Doctor'].value_counts(dropna=False,normalize=True) #
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 2.00   0.92
1.00   0.08
Name: Could_Not_Afford_To_See_Doctor, dtype: float64
```

[60]:
```
# Create a mapping dictionary:
doctor_mapping = {1: 'yes',
                  2: 'no'
                 }

# Apply the mapping to the doctor column:
df['Could_Not_Afford_To_See_Doctor'] = df['Could_Not_Afford_To_See_Doctor'].map(doctor_mapping)

# Rename the column:
df.rename(columns={'Could_Not_Afford_To_See_Doctor': 'could_not_afford_to_see_doctor'}, inplace=True)
```

Column 8: Length_of_time_since_last_routine_checkup

Contents

[61]:
```
#view column counts:
df.Length_of_time_since_last_routine_checkup.value_counts(dropna=False)
```

[61]:
```
1.00    350944
2.00     41919
3.00     24882
4.00     19079
7.00      5063
8.00      2509
9.00       733
NaN          3
Name: Length_of_time_since_last_routine_checkup, dtype: int64
```

Could_Not_Afford_To_See_Doctor:

- 1: 'past_year',
- 2: 'past_2_years',
- 3: 'past_5_years',
- 4: '5+_years_ago',
- 7: 'dont_know',
- 8: 'never',
- 9: 'refused', so for 7, 9 let's convert to nan:

[62]:
```
#Replace 7 and 9 with NaN:
df['Length_of_time_since_last_routine_checkup'].replace([7, 9], np.nan, inplace=True)
df.Length_of_time_since_last_routine_checkup.value_counts(dropna=False)
```

[62]:
```
1.00    350944
2.00     41919
3.00     24882
4.00     19079
NaN       5799
8.00      2509
Name: Length_of_time_since_last_routine_checkup, dtype: int64
```

[63]:
```
# Calculate the distribution of existing values:
value_counts = df['Length_of_time_since_last_routine_checkup'].value_counts(normalize=True, dropna=True)
print("Original Length_of_time_since_last_routine_checkup:\n", value_counts)
```

```
Original Length_of_time_since_last_routine_checkup:
 1.00   0.80
2.00   0.10
3.00   0.06
4.00   0.04
8.00   0.01
Name: Length_of_time_since_last_routine_checkup, dtype: float64
```

[64]:
```
# Function to impute missing values based on distribution:
def impute_missing(row):
    if pd.isna(row['Length_of_time_since_last_routine_checkup']):
        return np.random.choice(value_counts.index, p=value_counts.values)
    else:
        return row['Length_of_time_since_last_routine_checkup']
```

[65]:
```
# Apply the imputation function:
df['Length_of_time_since_last_routine_checkup'] = df.apply(impute_missing, axis=1)
```

[66]:
```
# Verify the imputation:
imputed_value_counts = df['Length_of_time_since_last_routine_checkup'].value_counts(dropna=False) # normalize=True
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 1.00    355585
2.00     42502
3.00     25212
4.00     19296
8.00      2537
Name: Length_of_time_since_last_routine_checkup, dtype: int64
```

[67]:
```
# Verify the imputation:
imputed_value_counts = df['Length_of_time_since_last_routine_checkup'].value_counts(dropna=False,normalize=True) #
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 1.00   0.80
2.00   0.10
3.00   0.06
4.00   0.04
8.00   0.01
Name: Length_of_time_since_last_routine_checkup, dtype: float64
```

[68]:
```
# Create a mapping dictionary:
checkup_mapping = {1: 'past_year',
                   2: 'past_2_years',
                   3: 'past_5_years',
                   4: '5+_years_ago',
                   8: 'never',
                  }

# Apply the mapping to the checkup_mapping column:
df['Length_of_time_since_last_routine_checkup'] = df['Length_of_time_since_last_routine_checkup'].map(checkup_mapping)

# Rename the column:
df.rename(columns={'Length_of_time_since_last_routine_checkup': 'length_of_time_since_last_routine_checkup'}, inplace=True)
```

[69]:
```
#view column counts:
df['length_of_time_since_last_routine_checkup'].value_counts(dropna=False,normalize=True)
```

[69]:
```
past_year      0.80
past_2_years   0.10
past_5_years   0.06
5+_years_ago   0.04
never          0.01
Name: length_of_time_since_last_routine_checkup, dtype: float64
```

Column 9: Ever_Diagnosed_with_Heart_Attack

Contents

[70]:
```
#view column counts:
df['Ever_Diagnosed_with_Heart_Attack'].value_counts(dropna=False)
```

[70]:
```
2.00    416959
1.00     25108
7.00      2731
9.00       330
NaN          4
Name: Ever_Diagnosed_with_Heart_Attack, dtype: int64
```

Ever_Diagnosed_with_Heart_Attack:

- 1: yes
- 2: no
- 7: dont_know
- 9: refused

so for 7, 9 let's convert to nan:

[71]:
```
#Replace 7 and 9 with NaN:
df['Ever_Diagnosed_with_Heart_Attack'].replace([7, 9], np.nan, inplace=True)
df.Ever_Diagnosed_with_Heart_Attack.value_counts(dropna=False)
```

[71]:
```
2.00    416959
1.00     25108
NaN       3065
Name: Ever_Diagnosed_with_Heart_Attack, dtype: int64
```

[72]:
```
# Calculate the distribution of existing values:
value_counts = df['Ever_Diagnosed_with_Heart_Attack'].value_counts(normalize=True, dropna=True)
print("Original Length_of_time_since_last_routine_checkup:\n", value_counts)
```

```
Original Length_of_time_since_last_routine_checkup:
 2.00   0.94
1.00   0.06
Name: Ever_Diagnosed_with_Heart_Attack, dtype: float64
```

[73]:
```
# Function to impute missing values based on distribution:
def impute_missing(row):
    if pd.isna(row['Ever_Diagnosed_with_Heart_Attack']):
        return np.random.choice(value_counts.index, p=value_counts.values)
    else:
        return row['Ever_Diagnosed_with_Heart_Attack']
```

[74]:
```
# Apply the imputation function:
df['Ever_Diagnosed_with_Heart_Attack'] = df.apply(impute_missing, axis=1)
```

[75]:
```
# Verify the imputation:
imputed_value_counts = df['Ever_Diagnosed_with_Heart_Attack'].value_counts(dropna=False) # normalize=True
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 2.00    419856
1.00     25276
Name: Ever_Diagnosed_with_Heart_Attack, dtype: int64
```

[76]:
```
# Verify the imputation:
imputed_value_counts = df['Ever_Diagnosed_with_Heart_Attack'].value_counts(dropna=False,normalize=True) #
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 2.00   0.94
1.00   0.06
Name: Ever_Diagnosed_with_Heart_Attack, dtype: float64
```

[77]:
```
# Create a mapping dictionary:
heart_attack_mapping = {1: 'yes',
                        2: 'no',

                       }

# Apply the mapping to the heart_attack_mapping column:
df['Ever_Diagnosed_with_Heart_Attack'] = df['Ever_Diagnosed_with_Heart_Attack'].map(heart_attack_mapping)

# Rename the column:
df.rename(columns={'Ever_Diagnosed_with_Heart_Attack': 'ever_diagnosed_with_heart_attack'}, inplace=True)
```

[78]:
```
#view column counts:
df['ever_diagnosed_with_heart_attack'].value_counts(dropna=False,normalize=True) #
```

[78]:
```
no    0.94
yes   0.06
Name: ever_diagnosed_with_heart_attack, dtype: float64
```

Column 10: Ever_Diagnosed_with_a_Stroke

Contents

[79]:
```
#view column counts:
df['Ever_Diagnosed_with_a_Stroke'].value_counts(dropna=False)
```

[79]:
```
2.00    424336
1.00     19239
7.00      1274
9.00       281
NaN          2
Name: Ever_Diagnosed_with_a_Stroke, dtype: int64
```

Ever_Diagnosed_with_Heart_Attack:

- 1: yes
- 2: no
- 7: dont_know
- 9: refused

so for 7, 9 let's convert to nan:

[80]:
```
#Replace 7 and 9 with NaN:
df['Ever_Diagnosed_with_a_Stroke'].replace([7, 9], np.nan, inplace=True)
df.Ever_Diagnosed_with_a_Stroke.value_counts(dropna=False)
```

[80]:
```
2.00    424336
1.00     19239
NaN       1557
Name: Ever_Diagnosed_with_a_Stroke, dtype: int64
```

[81]:
```
# Calculate the distribution of existing values:
value_counts = df['Ever_Diagnosed_with_a_Stroke'].value_counts(normalize=True, dropna=True)
print("Original Ever_Diagnosed_with_a_Stroke:\n", value_counts)
```

```
Original Ever_Diagnosed_with_a_Stroke:
 2.00   0.96
1.00   0.04
Name: Ever_Diagnosed_with_a_Stroke, dtype: float64
```

[82]:
```
# Function to impute missing values based on distribution:
def impute_missing(row):
    if pd.isna(row['Ever_Diagnosed_with_a_Stroke']):
        return np.random.choice(value_counts.index, p=value_counts.values)
    else:
        return row['Ever_Diagnosed_with_a_Stroke']
```

[83]:
```
# Apply the imputation function:
df['Ever_Diagnosed_with_a_Stroke'] = df.apply(impute_missing, axis=1)
```

[84]:
```
# Verify the imputation:
imputed_value_counts = df['Ever_Diagnosed_with_a_Stroke'].value_counts(dropna=False) # normalize=True
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 2.00    425821
1.00     19311
Name: Ever_Diagnosed_with_a_Stroke, dtype: int64
```

[85]:
```
# Verify the imputation:
imputed_value_counts = df['Ever_Diagnosed_with_a_Stroke'].value_counts(dropna=False,normalize=True) #
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 2.00   0.96
1.00   0.04
Name: Ever_Diagnosed_with_a_Stroke, dtype: float64
```

[86]:
```
# Create a mapping dictionary:
stroke_mapping = {1: 'yes',
                  2: 'no',

                }

# Apply the mapping to the stroke column:
df['Ever_Diagnosed_with_a_Stroke'] = df['Ever_Diagnosed_with_a_Stroke'].map(stroke_mapping)

# Rename the column:
df.rename(columns={'Ever_Diagnosed_with_a_Stroke': 'ever_diagnosed_with_a_stroke'}, inplace=True)
```

[87]:
```
#view column counts:
df['ever_diagnosed_with_a_stroke'].value_counts(dropna=False,normalize=True) #
```

[87]:
```
no    0.96
yes   0.04
Name: ever_diagnosed_with_a_stroke, dtype: float64
```

Column 11: Ever_told_you_had_a_depressive_disorder

Contents

[88]:
```
#view column counts:
value_counts_with_percentage(df, 'Ever_told_you_had_a_depressive_disorder')
```

[88]:

	Count	Percentage
2.00	350910	78.83
1.00	91410	20.54
7.00	2140	0.48
9.00	665	0.15
NaN	7	0.00

Ever_told_you_had_a_depressive_disorder:

- 1: yes
- 2: no
- 7: dont_know
- 9: refused

so for 7, 9 let's convert to nan:

[89]:
```
#Replace 7 and 9 with NaN:
df['Ever_told_you_had_a_depressive_disorder'].replace([7, 9], np.nan, inplace=True)
df.Ever_told_you_had_a_depressive_disorder.value_counts(dropna=False)
```

[89]:
```
2.00    350910
1.00     91410
NaN       2812
Name: Ever_told_you_had_a_depressive_disorder, dtype: int64
```

[90]:
```
# Calculate the distribution of existing values:
value_counts = df['Ever_told_you_had_a_depressive_disorder'].value_counts(normalize=True, dropna=True)
print("Original Ever_told_you_had_a_depressive_disorder:\n", value_counts)
```

```
Original Ever_told_you_had_a_depressive_disorder:
 2.00   0.79
1.00   0.21
Name: Ever_told_you_had_a_depressive_disorder, dtype: float64
```

[91]:
```
# Function to impute missing values based on distribution:
def impute_missing(row):
    if pd.isna(row['Ever_told_you_had_a_depressive_disorder']):
        return np.random.choice(value_counts.index, p=value_counts.values)
    else:
        return row['Ever_told_you_had_a_depressive_disorder']
```

[92]:
```
# Apply the imputation function:
df['Ever_told_you_had_a_depressive_disorder'] = df.apply(impute_missing, axis=1)
```

[93]:
```
# Verify the imputation:
imputed_value_counts = df['Ever_told_you_had_a_depressive_disorder'].value_counts(dropna=False) # normalize=True
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 2.00    353187
1.00     91945
Name: Ever_told_you_had_a_depressive_disorder, dtype: int64
```

[94]:
```
# Verify the imputation:
imputed_value_counts = df['Ever_told_you_had_a_depressive_disorder'].value_counts(dropna=False,normalize=True) #
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 2.00   0.79
1.00   0.21
Name: Ever_told_you_had_a_depressive_disorder, dtype: float64
```

[95]:
```
# Create a mapping dictionary:
depressive_disorder_mapping = {1: 'yes',
                   2: 'no',

                  }

# Apply the mapping to the depressive_disorder column:
df['Ever_told_you_had_a_depressive_disorder'] = df['Ever_told_you_had_a_depressive_disorder'].map(depressive_disorder_mapping)

# Rename the column:
df.rename(columns={'Ever_told_you_had_a_depressive_disorder': 'ever_told_you_had_a_depressive_disorder'}, inplace=True)
```

[96]:
```
#view column counts & percentage:
value_counts_with_percentage(df, 'ever_told_you_had_a_depressive_disorder')
```

[96]:

	Count	Percentage
no	353187	79.34
yes	91945	20.66

Column 12: Ever_told_you_have_kidney_disease

Contents

[97]:
```
#view column counts & percentage:
value_counts_with_percentage(df, 'Ever_told_you_have_kidney_disease')
```

[97]:

	Count	Percentage
2.00	422891	95.00
1.00	20315	4.56
7.00	1581	0.36
9.00	343	0.08
NaN	2	0.00

Ever_told_you_had_a_depressive_disorder:

- 1: yes
- 2: no
- 7: dont_know
- 9: refused

so for 7, 9 let's convert to nan:

[98]:
```
#Replace 7 and 9 with NaN:
df['Ever_told_you_have_kidney_disease'].replace([7, 9], np.nan, inplace=True)
df.Ever_told_you_have_kidney_disease.value_counts(dropna=False)
```

[98]:
```
2.00    422891
1.00     20315
NaN       1926
Name: Ever_told_you_have_kidney_disease, dtype: int64
```

[99]:
```
# Calculate the distribution of existing values:
value_counts = df['Ever_told_you_have_kidney_disease'].value_counts(normalize=True, dropna=True)
print("Original Ever_told_you_have_kidney_disease:\n", value_counts)
```

```
Original Ever_told_you_have_kidney_disease:
 2.00   0.95
1.00   0.05
Name: Ever_told_you_have_kidney_disease, dtype: float64
```

[100]:
```
# Function to impute missing values based on distribution:
def impute_missing(row):
    if pd.isna(row['Ever_told_you_have_kidney_disease']):
        return np.random.choice(value_counts.index, p=value_counts.values)
    else:
        return row['Ever_told_you_have_kidney_disease']
```

[101]:
```
# Apply the imputation function:
df['Ever_told_you_have_kidney_disease'] = df.apply(impute_missing, axis=1)
```

[102]:
```
# Verify the imputation:
imputed_value_counts = df['Ever_told_you_have_kidney_disease'].value_counts(dropna=False) # normalize=True
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 2.00    424726
1.00     20486
Name: Ever_told_you_have_kidney_disease, dtype: int64
```

[103]:
```
# Verify the imputation:
imputed_value_counts = df['Ever_told_you_have_kidney_disease'].value_counts(dropna=False,normalize=True) #
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 2.00   0.95
1.00   0.05
Name: Ever_told_you_have_kidney_disease, dtype: float64
```

[104]:
```
# Create a mapping dictionary:
kidney_mapping = {1: 'yes',
                  2: 'no',

                 }

# Apply the mapping to the kidney column:
df['Ever_told_you_have_kidney_disease'] = df['Ever_told_you_have_kidney_disease'].map(kidney_mapping)

# Rename the column:
df.rename(columns={'Ever_told_you_have_kidney_disease': 'ever_told_you_have_kidney_disease'}, inplace=True)
```

[105]:
```
#view column counts & percentage:
value_counts_with_percentage(df, 'ever_told_you_have_kidney_disease')
```

[105]:

	Count	Percentage
no	424726	95.42
yes	20406	4.58

Column 13: Ever_told_you_had_diabetes

Contents

[106]:
```
#view column counts & percentage:
value_counts_with_percentage(df, 'Ever_told_you_had_diabetes')
```

[106]:

	Count	Percentage
3.00	368722	82.83
1.00	61158	13.74
4.00	10329	2.32
2.00	3836	0.86
7.00	763	0.17
9.00	321	0.07
NaN	3	0.00

Ever_told_you_had_diabetes:

- 1: 'yes',
- 2: 'yes_during_pregnancy',
- 3: 'no',
- 4: 'no_prediabetes',
- 7: 'dont_know',
- 9: 'refused',

so for 7, 9 let's convert to nan:

[107]:
```
#Replace 7 and 9 with NaN:
df['Ever_told_you_had_diabetes'].replace([7, 9], np.nan, inplace=True)
df.Ever_told_you_had_diabetes.value_counts(dropna=False)
```

[107]:
```
3.00    368722
1.00     61158
4.00     10329
2.00      3836
NaN       1087
Name: Ever_told_you_had_diabetes, dtype: int64
```

[108]:
```
# Calculate the distribution of existing values:
value_counts = df['Ever_told_you_had_diabetes'].value_counts(normalize=True, dropna=True)
print("Original Ever_told_you_have_kidney_disease:\n", value_counts)
```

```
Original Ever_told_you_have_kidney_disease:
 3.00   0.83
1.00   0.14
4.00   0.02
2.00   0.01
Name: Ever_told_you_had_diabetes, dtype: float64
```

[109]:
```
# Function to impute missing values based on distribution:
def impute_missing(row):
    if pd.isna(row['Ever_told_you_had_diabetes']):
        return np.random.choice(value_counts.index, p=value_counts.values)
    else:
        return row['Ever_told_you_had_diabetes']
```

[110]:
```
# Apply the imputation function:
df['Ever_told_you_had_diabetes'] = df.apply(impute_missing, axis=1)
```

[111]:
```
# Verify the imputation:
imputed_value_counts = df['Ever_told_you_had_diabetes'].value_counts(dropna=False) # normalize=True
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 3.00    369615
1.00     61307
4.00     10362
2.00      3848
Name: Ever_told_you_had_diabetes, dtype: int64
```

[112]:
```
# Verify the imputation:
imputed_value_counts = df['Ever_told_you_had_diabetes'].value_counts(dropna=False,normalize=True) #
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 3.00   0.83
1.00   0.14
4.00   0.02
2.00   0.01
Name: Ever_told_you_had_diabetes, dtype: float64
```

[113]:
```
# Create a mapping dictionary:
diabetes_mapping = {1: 'yes',
                    2: 'yes_during_pregnancy',
                    3: 'no',
                    4: 'no_prediabetes',

                   }

# Apply the mapping to the diabetes column:
df['Ever_told_you_had_diabetes'] = df['Ever_told_you_had_diabetes'].map(diabetes_mapping)

# Rename the column:
df.rename(columns={'Ever_told_you_had_diabetes': 'ever_told_you_had_diabetes'}, inplace=True)
```

[114]:
```
#view column counts & percentage:
value_counts_with_percentage(df, 'ever_told_you_had_diabetes')
```

[114]:

	Count	Percentage
no	369615	83.03
yes	61307	13.77
no_prediabetes	10362	2.33
yes_during_pregnancy	3848	0.86

Column 14: Computed_body_mass_index_categories

Contents

[115]:
```
#view column counts & percentage:
value_counts_with_percentage(df, 'Computed_body_mass_index_categories')
```

[115]:

	Count	Percentage
3.00	139995	31.45
4.00	132577	29.78
2.00	116976	26.28
NaN	48806	10.96
1.00	6778	1.52

Computed_body_mass_index_categories:

- 1: 'underweight_bmi_less_than_18_5',
- 2: 'normal_weight_bmi_18_5_to_24_9',
- 3: 'overweight_bmi_25_to_29_9',
- 4: 'obese_bmi_30_or_more',

[116]:
```
# Calculate the distribution of existing values:
value_counts = df['Computed_body_mass_index_categories'].value_counts(normalize=True, dropna=True)
print("Original Computed_body_mass_index_categories:\n", value_counts)
```

```
Original Computed_body_mass_index_categories:
 3.00   0.35
4.00   0.33
2.00   0.30
1.00   0.02
Name: Computed_body_mass_index_categories, dtype: float64
```

[117]:
```
# Function to impute missing values based on distribution:
def impute_missing(row):
    if pd.isna(row['Computed_body_mass_index_categories']):
        return np.random.choice(value_counts.index, p=value_counts.values)
    else:
        return row['Computed_body_mass_index_categories']
```

[118]:
```
# Apply the imputation function:
df['Computed_body_mass_index_categories'] = df.apply(impute_missing, axis=1)
```

[119]:
```
# Verify the imputation:
imputed_value_counts = df['Computed_body_mass_index_categories'].value_counts(dropna=False) # normalize=True
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 3.00    157226
4.00    148919
2.00    131365
1.00      7622
Name: Computed_body_mass_index_categories, dtype: int64
```

[120]:
```
# Verify the imputation:
imputed_value_counts = df['Computed_body_mass_index_categories'].value_counts(dropna=False,normalize=True) #
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 3.00   0.35
4.00   0.33
2.00   0.30
1.00   0.02
Name: Computed_body_mass_index_categories, dtype: float64
```

[121]:
```
# Create a mapping dictionary:
bmi_mapping = {1: 'underweight_bmi_less_than_18_5',
                    2: 'normal_weight_bmi_18_5_to_24_9',
                    3: 'overweight_bmi_25_to_29_9',
                    4: 'obese_bmi_30_or_more',

                }

# Apply the mapping to the bmi column:
df['Computed_body_mass_index_categories'] = df['Computed_body_mass_index_categories'].map(bmi_mapping)

# Rename the column:
df.rename(columns={'Computed_body_mass_index_categories': 'BMI'}, inplace=True)
```

[122]:
```
#view column counts & percentage:
value_counts_with_percentage(df, 'BMI')
```

[122]:

	Count	Percentage
overweight_bmi_25_to_29_9	157226	35.32
obese_bmi_30_or_more	148919	33.46
normal_weight_bmi_18_5_to_24_9	131365	29.51
underweight_bmi_less_than_18_5	7622	1.71

Column 15: Difficulty_Walking_or_Climbing_Stairs

Contents

[123]:
```
#view column counts & percentage:
value_counts_with_percentage(df, 'Difficulty_Walking_or_Climbing_Stairs')
```

[123]:

	Count	Percentage
2.00	353039	79.31
1.00	68081	15.29
NaN	22155	4.98
7.00	1221	0.27
9.00	636	0.14

Difficulty_Walking_or_Climbing_Stairs:

- 1: yes
- 2: no
- 7: dont_know
- 9: refused

so for 7, 9 let's convert to nan:

[124]:
```
#Replace 7 and 9 with NaN:
df['Difficulty_Walking_or_Climbing_Stairs'].replace([7, 9], np.nan, inplace=True)
df.Difficulty_Walking_or_Climbing_Stairs.value_counts(dropna=False)
```

[124]:
```
2.00    353039
1.00     68081
NaN      24012
Name: Difficulty_Walking_or_Climbing_Stairs, dtype: int64
```

[125]:
```
# Calculate the distribution of existing values:
value_counts = df['Difficulty_Walking_or_Climbing_Stairs'].value_counts(normalize=True, dropna=True)
print("Original Difficulty_Walking_or_Climbing_Stairs:\n", value_counts)
```

```
Original Difficulty_Walking_or_Climbing_Stairs:
 2.00   0.84
1.00   0.16
Name: Difficulty_Walking_or_Climbing_Stairs, dtype: float64
```

[126]:
```
# Function to impute missing values based on distribution:
def impute_missing(row):
    if pd.isna(row['Difficulty_Walking_or_Climbing_Stairs']):
        return np.random.choice(value_counts.index, p=value_counts.values)
    else:
        return row['Difficulty_Walking_or_Climbing_Stairs']
```

[127]:
```
# Apply the imputation function:
df['Difficulty_Walking_or_Climbing_Stairs'] = df.apply(impute_missing, axis=1)
```

[128]:
```
# Verify the imputation:
imputed_value_counts = df['Difficulty_Walking_or_Climbing_Stairs'].value_counts(dropna=False) # normalize=True
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 2.00    373082
1.00     72050
Name: Difficulty_Walking_or_Climbing_Stairs, dtype: int64
```

[129]:
```
# Verify the imputation:
imputed_value_counts = df['Difficulty_Walking_or_Climbing_Stairs'].value_counts(dropna=False,normalize=True) #
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 2.00   0.84
1.00   0.16
Name: Difficulty_Walking_or_Climbing_Stairs, dtype: float64
```

[130]:
```
# Create a mapping dictionary:
climbing_mapping = {1: 'yes',
                    2: 'no',

                   }

# Apply the mapping to the climbing_mapping column:
df['Difficulty_Walking_or_Climbing_Stairs'] = df['Difficulty_Walking_or_Climbing_Stairs'].map(climbing_mapping)

# Rename the column:
df.rename(columns={'Difficulty_Walking_or_Climbing_Stairs': 'difficulty_walking_or_climbing_stairs'}, inplace=True)
```

[131]:
```
#view column counts & percentage:
value_counts_with_percentage(df, 'difficulty_walking_or_climbing_stairs')
```

[131]:

	Count	Percentage
no	373082	83.81
yes	72050	16.19

Column 16: Computed_Physical_Health_Status

Contents

[132]:
```
#view column counts & percentage:
value_counts_with_percentage(df, 'Computed_Physical_Health_Status')
```

[132]:

	Count	Percentage
1.00	267819	60.17
2.00	108312	24.33
3.00	58074	13.05
9.00	10927	2.45

Computed_Physical_Health_Status:

- 1: 'zero_days_not_good',
- 2: '1_to_13_days_not_good',
- 3: '14_plus_days_not_good',
- 9: 'dont_know'

so for 9 let's convert to nan:

[133]:
```
#Replace 7 and 9 with NaN:
df['Computed_Physical_Health_Status'].replace([9], np.nan, inplace=True)
df.Computed_Physical_Health_Status.value_counts(dropna=False)
```

[133]:
```
1.00    267819
2.00    108312
3.00     58074
NaN      10927
Name: Computed_Physical_Health_Status, dtype: int64
```

[134]:
```
# Calculate the distribution of existing values:
value_counts = df['Computed_Physical_Health_Status'].value_counts(normalize=True, dropna=True)
print("Original Computed_Physical_Health_Status:\n", value_counts)
```

```
Original Computed_Physical_Health_Status:
 1.00   0.62
2.00   0.25
3.00   0.13
Name: Computed_Physical_Health_Status, dtype: float64
```

[135]:
```
# Function to impute missing values based on distribution:
def impute_missing(row):
    if pd.isna(row['Computed_Physical_Health_Status']):
        return np.random.choice(value_counts.index, p=value_counts.values)
    else:
        return row['Computed_Physical_Health_Status']
```

[136]:
```
# Apply the imputation function:
df['Computed_Physical_Health_Status'] = df.apply(impute_missing, axis=1)
```

[137]:
```
# Verify the imputation:
imputed_value_counts = df['Computed_Physical_Health_Status'].value_counts(dropna=False) # normalize=True
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 1.00    274525
2.00    111056
3.00     59551
Name: Computed_Physical_Health_Status, dtype: int64
```

[138]:
```
# Verify the imputation:
imputed_value_counts = df['Computed_Physical_Health_Status'].value_counts(dropna=False,normalize=True) #
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 1.00   0.62
2.00   0.25
3.00   0.13
Name: Computed_Physical_Health_Status, dtype: float64
```

[139]:
```
# Create a mapping dictionary:
health_status_mapping = {1: 'zero_days_not_good',
                    2: '1_to_13_days_not_good',
                    3: '14_plus_days_not_good',

                }

# Apply the mapping to the health_status_mapping column:
df['Computed_Physical_Health_Status'] = df['Computed_Physical_Health_Status'].map(health_status_mapping)

# Rename the column:
df.rename(columns={'Computed_Physical_Health_Status': 'physical_health_status'}, inplace=True)
```

[140]:
```
#view column counts & percentage:
value_counts_with_percentage(df, 'physical_health_status')
```

[140]:

	Count	Percentage
zero_days_not_good	274525	61.67
1_to_13_days_not_good	111056	24.95
14_plus_days_not_good	59551	13.38

Column 17: Computed_Mental_Health_Status

Contents

[141]:
```
#view column counts & percentage:
value_counts_with_percentage(df, 'Computed_Mental_Health_Status')
```

[141]:

	Count	Percentage
1.00	265229	59.58
2.00	110616	24.85
3.00	60220	13.53
9.00	9067	2.04

Computed_Physical_Health_Status:

- 1: 'zero_days_not_good',
- 2: '1_to_13_days_not_good',
- 3: '14_plus_days_not_good',
- 9: 'dont_know'

so for 9 let's convert to nan:

[142]:
```
#Replace 7 and 9 with NaN:
df['Computed_Mental_Health_Status'].replace([9], np.nan, inplace=True)
df.Computed_Mental_Health_Status.value_counts(dropna=False)
```

```
1.00    265229
2.00    110616
3.00     60220
NaN       9067
Name: Computed_Mental_Health_Status, dtype: int64
```

[143]:
```
# Calculate the distribution of existing values:
value_counts = df['Computed_Mental_Health_Status'].value_counts(normalize=True, dropna=True)
print("Original Computed_Mental_Health_Status:\n", value_counts)
```

```
Original Computed_Mental_Health_Status:
 1.00   0.61
2.00   0.25
3.00   0.14
Name: Computed_Mental_Health_Status, dtype: float64
```

[144]:
```
# Function to impute missing values based on distribution:
def impute_missing(row):
    if pd.isna(row['Computed_Mental_Health_Status']):
        return np.random.choice(value_counts.index, p=value_counts.values)
    else:
        return row['Computed_Mental_Health_Status']
```

[145]:
```
# Apply the imputation function:
df['Computed_Mental_Health_Status'] = df.apply(impute_missing, axis=1)
```

[146]:
```
# Verify the imputation:
imputed_value_counts = df['Computed_Mental_Health_Status'].value_counts(dropna=False) # normalize=True
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 1.00    270792
2.00    112907
3.00     61433
Name: Computed_Mental_Health_Status, dtype: int64
```

[147]:
```
# Verify the imputation:
imputed_value_counts = df['Computed_Mental_Health_Status'].value_counts(dropna=False,normalize=True) #
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 1.00   0.61
2.00   0.25
3.00   0.14
Name: Computed_Mental_Health_Status, dtype: float64
```

[148]:
```
# Create a mapping dictionary:
m_health_status_mapping = {1: 'zero_days_not_good',
                   2: '1_to_13_days_not_good',
                   3: '14_plus_days_not_good',

                }

# Apply the mapping to the m_health_status_mapping column:
df['Computed_Mental_Health_Status'] = df['Computed_Mental_Health_Status'].map(m_health_status_mapping)

# Rename the column:
df.rename(columns={'Computed_Mental_Health_Status': 'mental_health_status'}, inplace=True)
```

[149]:
```
#view column counts & percentage:
value_counts_with_percentage(df, 'mental_health_status')
```

[149]:

	Count	Percentage
zero_days_not_good	270792	60.83
1_to_13_days_not_good	112907	25.36
14_plus_days_not_good	61433	13.80

Column 18: Computed_Asthma_Status

Contents

[150]:
```
#view column counts & percentage:
value_counts_with_percentage(df, 'Computed_Asthma_Status')
```

[150]:

	Count	Percentage
3.00	376665	84.62
1.00	45659	10.26
2.00	18948	4.26
9.00	3860	0.87

Computed_Asthma_Status:

- 1: 'current_asthma',
- 2: 'former_asthma',
- 3: 'never_asthma',
- 9: 'dont_know_refused_missing'

so for 9 let's convert to nan:

[151]:
```
#Replace 7 and 9 with NaN:
df['Computed_Asthma_Status'].replace([9], np.nan, inplace=True)
df.Computed_Asthma_Status.value_counts(dropna=False)
```

[151]:
```
3.00    376665
1.00     45659
2.00     18948
NaN       3860
Name: Computed_Asthma_Status, dtype: int64
```

[152]:
```
# Calculate the distribution of existing values:
value_counts = df['Computed_Asthma_Status'].value_counts(normalize=True, dropna=True)
print("Original Computed_Asthma_Status:\n", value_counts)
```

```
Original Computed_Asthma_Status:
 3.00   0.85
1.00   0.10
2.00   0.04
Name: Computed_Asthma_Status, dtype: float64
```

[153]:
```
# Function to impute missing values based on distribution:
def impute_missing(row):
    if pd.isna(row['Computed_Asthma_Status']):
        return np.random.choice(value_counts.index, p=value_counts.values)
    else:
        return row['Computed_Asthma_Status']
```

[154]:
```
# Apply the imputation function:
df['Computed_Asthma_Status'] = df.apply(impute_missing, axis=1)
```

[155]:
```
# Verify the imputation:
imputed_value_counts = df['Computed_Asthma_Status'].value_counts(dropna=False) # normalize=True
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 3.00    379980
1.00     46035
2.00     19117
Name: Computed_Asthma_Status, dtype: int64
```

[156]:
```
# Verify the imputation:
imputed_value_counts = df['Computed_Asthma_Status'].value_counts(dropna=False,normalize=True) #
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 3.00   0.85
1.00   0.10
2.00   0.04
Name: Computed_Asthma_Status, dtype: float64
```

[157]:
```
# Create a mapping dictionary:
Asthma_Status_mapping = {1: 'current_asthma',
                         2: 'former_asthma',
                         3: 'never_asthma',


                }

# Apply the mapping to the Asthma_Status_mapping column:
df['Computed_Asthma_Status'] = df['Computed_Asthma_Status'].map(Asthma_Status_mapping)

# Rename the column:
df.rename(columns={'Computed_Asthma_Status': 'asthma_Status'}, inplace=True)
```

[158]:
```
#view column counts & percentage:
value_counts_with_percentage(df, 'asthma_Status')
```

[158]:

	Count	Percentage
never_asthma	379980	85.36
current_asthma	46035	10.34
former_asthma	19117	4.29

Column 19: Exercise_in_Past_30_Days

Contents

[159]:
```
#view column counts & percentage:
value_counts_with_percentage(df, 'Exercise_in_Past_30_Days')
```

[159]:

	Count	Percentage
1.00	337559	75.83
2.00	106480	23.92
7.00	724	0.16
9.00	367	0.08
NaN	2	0.00

Exercise_in_Past_30_Days:

- 1: 'yes',
- 2: 'no',
- 7: 'dont_know'
- 9: 'refused_missing'

so for 7, 9 let's convert to nan:

[160]:
```
#Replace 7 and 9 with NaN:
df['Exercise_in_Past_30_Days'].replace([7, 9], np.nan, inplace=True)
df.Exercise_in_Past_30_Days.value_counts(dropna=False)
```

[160]:
```
1.00    337559
2.00    106480
NaN       1093
Name: Exercise_in_Past_30_Days, dtype: int64
```

[161]:
```
# Calculate the distribution of existing values:
value_counts = df['Exercise_in_Past_30_Days'].value_counts(normalize=True, dropna=True)
print("Original Exercise_in_Past_30_Days:\n", value_counts)
```

```
Original Exercise_in_Past_30_Days:
 1.00   0.76
2.00   0.24
Name: Exercise_in_Past_30_Days, dtype: float64
```

[162]:
```
# Function to impute missing values based on distribution:
def impute_missing(row):
    if pd.isna(row['Exercise_in_Past_30_Days']):
        return np.random.choice(value_counts.index, p=value_counts.values)
    else:
        return row['Exercise_in_Past_30_Days']
```

[163]:
```
# Apply the imputation function:
df['Exercise_in_Past_30_Days'] = df.apply(impute_missing, axis=1)
```

[164]:
```
# Verify the imputation:
imputed_value_counts = df['Exercise_in_Past_30_Days'].value_counts(dropna=False) # normalize=True
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 1.00    338376
2.00    106756
Name: Exercise_in_Past_30_Days, dtype: int64
```

[165]:
```
# Verify the imputation:
imputed_value_counts = df['Exercise_in_Past_30_Days'].value_counts(dropna=False,normalize=True) #
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 1.00   0.76
2.00   0.24
Name: Exercise_in_Past_30_Days, dtype: float64
```

[166]:
```
# Create a mapping dictionary:
exercise_Status_mapping = {1: 'yes',
                           2: 'no',

                }

# Apply the mapping to the exercise_Status_mapping column:
df['Exercise_in_Past_30_Days'] = df['Exercise_in_Past_30_Days'].map(exercise_Status_mapping)

# Rename the column:
df.rename(columns={'Exercise_in_Past_30_Days': 'exercise_status_in_past_30_Days'}, inplace=True)
```

[167]:
```
#view column counts & percentage:
value_counts_with_percentage(df, 'exercise_status_in_past_30_Days')
```

[167]:

	Count	Percentage
yes	338376	76.02
no	106756	23.98

Column 20: Computed_Smoking_Status

Contents

[168]:
```
#view column counts & percentage:
value_counts_with_percentage(df, 'Computed_Smoking_Status')
```

[168]:

	Count	Percentage
4.00	245955	55.25
3.00	113774	25.56
1.00	36003	8.09
9.00	35462	7.97
2.00	13938	3.13

Computed_Smoking_Status:

- 1: 'current_smoker_every_day',
- 2: 'current_smoker_some_days',
- 3: 'former_smoker',
- 4: 'never_smoked',
- 9: 'dont_know_refused_missing'

so for 9 let's convert to nan:

[169]:
```
#Replace 7 and 9 with NaN:
df['Computed_Smoking_Status'].replace([9], np.nan, inplace=True)
df.Computed_Smoking_Status.value_counts(dropna=False)
```

[169]:
```
4.00    245955
3.00    113774
1.00     36003
NaN      35462
2.00     13938
Name: Computed_Smoking_Status, dtype: int64
```

[170]:
```
# Calculate the distribution of existing values:
value_counts = df['Computed_Smoking_Status'].value_counts(normalize=True, dropna=True)
print("Original Computed_Smoking_Status:\n", value_counts)
```

```
Original Computed_Smoking_Status:
 4.00   0.60
3.00   0.28
1.00   0.09
2.00   0.03
Name: Computed_Smoking_Status, dtype: float64
```

[171]:
```
# Function to impute missing values based on distribution:
def impute_missing(row):
    if pd.isna(row['Computed_Smoking_Status']):
        return np.random.choice(value_counts.index, p=value_counts.values)
    else:
        return row['Computed_Smoking_Status']
```

[172]:
```
# Apply the imputation function:
df['Computed_Smoking_Status'] = df.apply(impute_missing, axis=1)
```

[173]:
```
# Verify the imputation:
imputed_value_counts = df['Computed_Smoking_Status'].value_counts(dropna=False) # normalize=True
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 4.00    267178
3.00    123743
1.00     39107
2.00     15104
Name: Computed_Smoking_Status, dtype: int64
```

[174]:
```
# Verify the imputation:
imputed_value_counts = df['Computed_Smoking_Status'].value_counts(dropna=False,normalize=True) #
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 4.00   0.60
3.00   0.28
1.00   0.09
2.00   0.03
Name: Computed_Smoking_Status, dtype: float64
```

[175]:
```
# Create a mapping dictionary:
smoking_Status_mapping = {1: 'current_smoker_every_day',
                          2: 'current_smoker_some_days',
                          3: 'former_smoker',
                          4: 'never_smoked'
                         }

# Apply the mapping to the smoking_Status_mapping column:
df['Computed_Smoking_Status'] = df['Computed_Smoking_Status'].map(smoking_Status_mapping)

# Rename the column:
df.rename(columns={'Computed_Smoking_Status': 'smoking_status'}, inplace=True)
```

[176]:
```
#view column counts & percentage:
value_counts_with_percentage(df, 'smoking_status')
```

[176]:

	Count	Percentage
never_smoked	267178	60.02
former_smoker	123743	27.80
current_smoker_every_day	39107	8.79
current_smoker_some_days	15104	3.39

Column 21: Binge_Drinking_Calculated_Variable

Contents

[177]:
```
#view column counts & percentage:
value_counts_with_percentage(df, 'Binge_Drinking_Calculated_Variable')
```

[177]:

	Count	Percentage
1.00	337114	75.73
2.00	56916	12.79
9.00	51102	11.48

Binge_Drinking_Calculated_Variable:

- 1: 'no',
- 2: 'yes',
- 9: 'dont_know_refused_missing'

so for 9 let's convert to nan:

[178]:
```
#Replace 7 and 9 with NaN:
df['Binge_Drinking_Calculated_Variable'].replace([9], np.nan, inplace=True)
df.Binge_Drinking_Calculated_Variable.value_counts(dropna=False)
```

[178]:
```
1.00    337114
2.00     56916
NaN      51102
Name: Binge_Drinking_Calculated_Variable, dtype: int64
```

[179]:
```
# Calculate the distribution of existing values:
value_counts = df['Binge_Drinking_Calculated_Variable'].value_counts(normalize=True, dropna=True)
print("Original Binge_Drinking_Calculated_Variable:\n", value_counts)
```

```
Original Binge_Drinking_Calculated_Variable:
 1.00   0.86
2.00   0.14
Name: Binge_Drinking_Calculated_Variable, dtype: float64
```

[180]:
```
# Function to impute missing values based on distribution:
def impute_missing(row):
    if pd.isna(row['Binge_Drinking_Calculated_Variable']):
        return np.random.choice(value_counts.index, p=value_counts.values)
    else:
        return row['Binge_Drinking_Calculated_Variable']
```

[181]:
```
# Apply the imputation function:
df['Binge_Drinking_Calculated_Variable'] = df.apply(impute_missing, axis=1)
```

[182]:
```
# Verify the imputation:
imputed_value_counts = df['Binge_Drinking_Calculated_Variable'].value_counts(dropna=False) # normalize=True
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 1.00    380851
2.00     64281
Name: Binge_Drinking_Calculated_Variable, dtype: int64
```

[183]:
```
# Verify the imputation:
imputed_value_counts = df['Binge_Drinking_Calculated_Variable'].value_counts(dropna=False,normalize=True) #
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 1.00   0.86
2.00   0.14
Name: Binge_Drinking_Calculated_Variable, dtype: float64
```

[184]:
```
# Create a mapping dictionary:
binge_drinking_status = {1: 'no',
                         2: 'yes'
                        }

# Apply the mapping to the binge_drinking_status column:
df['Binge_Drinking_Calculated_Variable'] = df['Binge_Drinking_Calculated_Variable'].map(binge_drinking_status)

# Rename the column:
df.rename(columns={'Binge_Drinking_Calculated_Variable': 'binge_drinking_status'}, inplace=True)
```

[185]:
```
#view column counts & percentage:
value_counts_with_percentage(df, 'binge_drinking_status')
```

[185]:

	Count	Percentage
no	380851	85.56
yes	64281	14.44

Column 22: How_Much_Time_Do_You_Sleep

Contents

[186]:
```
#view column counts & percentage:
value_counts_with_percentage(df, 'How_Much_Time_Do_You_Sleep')
```

[186]:

	Count	Percentage
7.00	132927	29.86
8.00	125442	28.18
6.00	95880	21.54
5.00	30122	6.77
9.00	21210	4.76
4.00	12433	2.79
10.00	10459	2.35
77.00	4792	1.08
3.00	3260	0.73
12.00	3004	0.67
2.00	1549	0.35
1.00	1154	0.26
11.00	686	0.15
99.00	658	0.15
16.00	329	0.07
15.00	317	0.07
14.00	295	0.07
18.00	168	0.04
13.00	165	0.04
20.00	143	0.03
24.00	52	0.01
17.00	27	0.01
22.00	19	0.00
23.00	18	0.00
19.00	16	0.00
21.00	4	0.00
NaN	3	0.00

[187]:
```
def categorize_sleep_hours(df, column_name):
    # Define the mapping dictionary for known values
    sleep_mapping = {
        77: 'dont_know',
        99: 'refused_to_answer',
        np.nan: 'missing'
    }

    # Categorize hours of sleep
    for hour in range(0, 4):
        sleep_mapping[hour] = 'very_short_sleep_0_to_3_hours'
    for hour in range(4, 6):
        sleep_mapping[hour] = 'short_sleep_4_to_5_hours'
    for hour in range(6, 9):
        sleep_mapping[hour] = 'normal_sleep_6_to_8_hours'
    for hour in range(9, 11):
        sleep_mapping[hour] = 'long_sleep_9_to_10_hours'
    for hour in range(11, 25):
        sleep_mapping[hour] = 'very_long_sleep_11_or_more_hours'

    # Map the values to their categories
    df['sleep_category'] = df[column_name].map(sleep_mapping)

    return df
```

[188]:
```
# Apply the function to categorize sleep hours
df = categorize_sleep_hours(df, 'How_Much_Time_Do_You_Sleep')
```

[189]:
```
#view column counts & percentage:
value_counts_with_percentage(df, 'sleep_category')
```

[189]:

	Count	Percentage
normal_sleep_6_to_8_hours	354249	79.58
short_sleep_4_to_5_hours	42555	9.56
long_sleep_9_to_10_hours	31669	7.11
very_short_sleep_0_to_3_hours	5963	1.34
very_long_sleep_11_or_more_hours	5243	1.18
dont_know	4792	1.08
refused_to_answer	658	0.15
missing	3	0.00

[190]:
```
#Replace 7 and 9 with NaN:
#df['sleep_category'].replace(['dont_know', 'refused_to_answer'], np.nan, inplace=True)
df['sleep_category'].replace(['missing', 'dont_know','refused_to_answer'], np.nan, inplace=True)
df.sleep_category.value_counts(dropna=False)
```

[190]:
```
normal_sleep_6_to_8_hours          354249
short_sleep_4_to_5_hours            42555
long_sleep_9_to_10_hours            31669
very_short_sleep_0_to_3_hours        5963
NaN                                  5453
very_long_sleep_11_or_more_hours     5243
```

[191]:
```
# Calculate the distribution of existing values:
value_counts = df['sleep_category'].value_counts(normalize=True, dropna=True)
print("Original sleep_category:\n", value_counts)
```

```
Original sleep_category:
 normal_sleep_6_to_8_hours          0.81
short_sleep_4_to_5_hours           0.10
long_sleep_9_to_10_hours           0.07
very_short_sleep_0_to_3_hours      0.01
very_long_sleep_11_or_more_hours   0.01
Name: sleep_category, dtype: float64
```

[192]:
```
# Function to impute missing values based on distribution:
def impute_missing(row):
    if pd.isna(row['sleep_category']):
        return np.random.choice(value_counts.index, p=value_counts.values)
    else:
        return row['sleep_category']
```

[193]:
```
# Apply the imputation function:
df['sleep_category'] = df.apply(impute_missing, axis=1)
```

[194]:
```
# Verify the imputation:
imputed_value_counts = df['sleep_category'].value_counts(dropna=False) # normalize=True
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 normal_sleep_6_to_8_hours          358636
short_sleep_4_to_5_hours            43092
long_sleep_9_to_10_hours            32046
very_short_sleep_0_to_3_hours        6049
very_long_sleep_11_or_more_hours     5309
Name: sleep_category, dtype: int64
```

[195]:
```
# Verify the imputation:
imputed_value_counts = df['sleep_category'].value_counts(dropna=False,normalize=True) #
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 normal_sleep_6_to_8_hours          0.81
short_sleep_4_to_5_hours           0.10
long_sleep_9_to_10_hours           0.07
very_short_sleep_0_to_3_hours      0.01
very_long_sleep_11_or_more_hours   0.01
Name: sleep_category, dtype: float64
```

[196]:
```
#view column counts & percentage:
value_counts_with_percentage(df, 'sleep_category')
```

[196]:

	Count	Percentage
normal_sleep_6_to_8_hours	358636	80.57
short_sleep_4_to_5_hours	43092	9.68
long_sleep_9_to_10_hours	32046	7.20
very_short_sleep_0_to_3_hours	6049	1.36
very_long_sleep_11_or_more_hours	5309	1.19

Column 23: Computed_number_of_drinks_of_alcohol_beverages_per_week

Contents

Column 23: Computed_number_of_drinks_of_alcohol_beverages_per_week

Contents

[197]:
```
#view column counts & percentage:
value_counts_with_percentage(df, 'Computed_number_of_drinks_of_alcohol_beverages_per_week')
```

[197]:

	Count	Percentage
0.00	188832	42.42
99900.00	49705	11.17
23.00	20646	4.64
47.00	18325	4.12
93.00	12104	2.72
200.00	10153	2.28
700.00	10143	2.28
100.00	9507	2.14
400.00	8845	1.99

[198]:
```
# Divide by 100 to get the number of drinks per week
df['drinks_per_week'] = df['Computed_number_of_drinks_of_alcohol_beverages_per_week'] / 100
```

[199]:
```
# Define the function to categorize the drink consumption
def categorize_drinks(drinks_per_week):
    #if drinks_per_week == 0:
        #return 'did_not_drink'
    if drinks_per_week == 99900 / 100:
        return 'do_not_know'
    elif 0.01 <= drinks_per_week <= 1:
        return 'very_low_consumption_0.01_to_1_drinks'
    elif 1.01 <= drinks_per_week <= 5:
        return 'low_consumption_1.01_to_5_drinks'
    elif 5.01 <= drinks_per_week <= 10:
        return 'moderate_consumption_5.01_to_10_drinks'
    elif 10.01 <= drinks_per_week <= 20:
        return 'high_consumption_10.01_to_20_drinks'
    elif drinks_per_week > 20:
        return 'very_high_consumption_more_than_20_drinks'
    else:
        return 'did_not_drink'
```

[200]:
```
# Apply the categorization function
df['drinks_category'] = df['drinks_per_week'].apply(categorize_drinks)
```

[201]:
```
#view column counts & percentage:
value_counts_with_percentage(df, 'drinks_category')
```

[201]:

	Count	Percentage
did_not_drink	188832	42.42
low_consumption_1.01_to_5_drinks	72539	16.30
very_low_consumption_0.01_to_1_drinks	68894	15.48
do_not_know	49705	11.17
moderate_consumption_5.01_to_10_drinks	33916	7.62
high_consumption_10.01_to_20_drinks	19734	4.43

[202]:
```
#Replace 7 and 9 with NaN:
df['drinks_category'].replace(['do_not_know'], np.nan, inplace=True)
df.drinks_category.value_counts(dropna=False)
```

[202]:
```
did_not_drink                          188832
low_consumption_1.01_to_5_drinks        72539
very_low_consumption_0.01_to_1_drinks   68894
NaN                                     49705
moderate_consumption_5.01_to_10_drinks  33916
high_consumption_10.01_to_20_drinks     19734
very_high_consumption_more_than_20_drinks  11512
Name: drinks_category, dtype: int64
```

[203]:
```
# Calculate the distribution of existing values:
value_counts = df['drinks_category'].value_counts(normalize=True, dropna=True)
print("Original drinks_category:\n", value_counts)
```

```
Original drinks_category:
 did_not_drink                          0.48
low_consumption_1.01_to_5_drinks        0.18
very_low_consumption_0.01_to_1_drinks   0.17
moderate_consumption_5.01_to_10_drinks  0.09
high_consumption_10.01_to_20_drinks     0.05
very_high_consumption_more_than_20_drinks  0.03
Name: drinks_category, dtype: float64
```

[204]:
```
# Function to impute missing values based on distribution:
def impute_missing(row):
    if pd.isna(row['drinks_category']):
        return np.random.choice(value_counts.index, p=value_counts.values)
    else:
        return row['drinks_category']
```

[205]:
```
# Apply the imputation function:
df['drinks_category'] = df.apply(impute_missing, axis=1)
```

[206]:
```
# Verify the imputation:
imputed_value_counts = df['drinks_category'].value_counts(dropna=False) # normalize=True
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 did_not_drink                          212603
low_consumption_1.01_to_5_drinks        81487
very_low_consumption_0.01_to_1_drinks   77630
moderate_consumption_5.01_to_10_drinks  38269
high_consumption_10.01_to_20_drinks     22197
very_high_consumption_more_than_20_drinks  12946
Name: drinks_category, dtype: int64
```

[207]:
```
# Verify the imputation:
imputed_value_counts = df['drinks_category'].value_counts(dropna=False,normalize=True) #
print("Distribution after imputation:\n", imputed_value_counts)
```

```
Distribution after imputation:
 did_not_drink                          0.48
low_consumption_1.01_to_5_drinks        0.18
very_low_consumption_0.01_to_1_drinks   0.17
moderate_consumption_5.01_to_10_drinks  0.09
high_consumption_10.01_to_20_drinks     0.05
very_high_consumption_more_than_20_drinks  0.03
Name: drinks_category, dtype: float64
```

[208]:
```
#Final check after imputation:
value_counts_with_percentage(df, 'drinks_category')
```

[208]:

	Count	Percentage
did_not_drink	212603	47.76
low_consumption_1.01_to_5_drinks	81487	18.31
very_low_consumption_0.01_to_1_drinks	77630	17.44
moderate_consumption_5.01_to_10_drinks	38269	8.60
high_consumption_10.01_to_20_drinks	22197	4.99
very_high_consumption_more_than_20_drinks	12946	2.91

Dropping unnecessary columns

Contents

[209]:
```
#Here, Let's drop the unnecessary colums:
columns_to_drop = ['Imputed_Age_value_collapsed_above_80', 'Reported_Weight_in_Pounds',
                   'Reported_Height_in_Feet_and_Inches', 'Leisure_Time_Physical_Activity_Calculated_Variable',
                  'Smoked_at_Least_100_Cigarettes', 'Computed_number_of_drinks_of_alcohol_beverages_per_week',
                  'How_Much_Time_Do_You_Sleep', 'drinks_per_week']
df = df.drop(columns=columns_to_drop)
```

Review the final structre of the cleaned dataframe

Contents

[210]:
```
#now, Let's look at the shape of df:
shape = df.shape
print("Number of rows:", shape[0], "\nNumber of columns:", shape[1])
```

```
Number of rows: 445132
Number of columns: 23
```

Review the final structre of the cleaned dataframe

Contents

[210]:
```
#now, Let's Look at the shape of df:
shape = df.shape
print("Number of rows:", shape[0], "\nNumber of columns:", shape[1])
```

```
Number of rows: 445132
Number of columns: 23
```

[211]:
```
summarize_df(df)
```

[211]:

	unique_count	data_types	missing_counts	missing_percentage
heart_disease	2	object	0	0.00
gender	3	object	0	0.00
race	7	object	0	0.00
general_health	5	object	0	0.00
health_care_provider	3	object	0	0.00
could_not_afford_to_see_doctor	2	object	0	0.00
length_of_time_since_last_routine_checkup	5	object	0	0.00
ever_diagnosed_with_heart_attack	2	object	0	0.00
ever_diagnosed_with_a_stroke	2	object	0	0.00
ever_told_you_had_a_depressive_disorder	2	object	0	0.00
ever_told_you_have_kidney_disease	2	object	0	0.00
ever_told_you_had_diabetes	4	object	0	0.00
BMI	4	object	0	0.00
difficulty_walking_or_climbing_stairs	2	object	0	0.00
physical_health_status	3	object	0	0.00
mental_health_status	3	object	0	0.00
asthma_Status	3	object	0	0.00
smoking_status	4	object	0	0.00
binge_drinking_status	2	object	0	0.00
exercise_status_in_past_30_Days	2	object	0	0.00
age_category	13	category	0	0.00
sleep_category	5	object	0	0.00
drinks_category	6	object	0	0.00

Awesome, there's no missing data. **So, as we can see above. we cleaned the data, removed missing data and still maintained the size of the dataset "rows"**

Saving the clean dataframe

Contents

[212]:
```
output_file_path = "./brfss2022_data_wrangling_output.csv"

df.to_csv(output_file_path, index=False)
```

Exploratory Data Analysis: AI-Powered Heart Disease Risk Assessment

- **Name:** Aktham Almomani
- **Course:** Probability and Statistics for Artificial Intelligence (MS-AAI-500-02) / University Of San Diego
- **Semester:** Summer 2024
- **Group:** 8

Contents

Introduction

Contents

Welcome to the Exploratory Data Analysis (EDA) notebook for our heart disease prediction project. This notebook serves as a critical step in our data science workflow, aimed at uncovering insights and patterns within our dataset that will guide our predictive modeling efforts.

In this notebook, we will:

- Validate the Dataset: Ensure the data is clean, consistent, and ready for analysis.
- Explore Feature Distributions: Analyze the distribution of various features in relation to heart disease.
- Convert Categorical Data: Transform categorical features into numeric format using CatBoost encoding for better analysis and modeling.
- Analyze Correlations: Examine both linear and non-linear relationships between features and the target variable (heart disease) using Pearson correlation and mutual information.
- Feature Selection: Identify and select key features that have the most predictive power for heart disease.

These steps will help us understand the data better, reveal important relationships, and prepare the data for building robust predictive models.

Dataset

Contents

The dataset used in this Exploratory Data Analysis (EDA) notebook is the result of a comprehensive data wrangling process. Data wrangling is a crucial step in the data science workflow, involving the transformation and preparation of raw data into a more usable format. The main tasks performed during data wrangling included:

- Dealing with missing data
- Data mapping
- Data cleaning
- Feature engineering

These steps ensured that the dataset is well-prepared for analysis and modeling, enabling us to build reliable and robust models for heart disease prediction.

Setup and preliminaries

Contents

Import libraries

Contents

[1]:
```
#Let's import the necessary packages:
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import math
import scipy.stats as stats
from scipy.stats import gamma, linregress
from bs4 import BeautifulSoup
import re
import category_encoders as ce
from sklearn.feature_selection import mutual_info_classif
from sklearn.model_selection import train_test_split

# Let's run below to customize notebook display:
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)

# format floating-point numbers to 2 decimal places: we'll adjust below requirement as needed for specific answers during this assignment:
#pd.set_option('float_format', '{:.2f}'.format)
```

Necessary functions

Contents

[2]:
```
def summarize_df(df):
    """
    Generate a summary DataFrame for an input DataFrame.
    Parameters:
    df (pd.DataFrame): The DataFrame to summarize.
    Returns:
    A dataframe containing the following columns:
              - 'unique_count': No. unique values in each column.
              - 'data_types': Data types of each column.
              - 'missing_counts': No. of missing (NaN) values in each column.
              - 'missing_percentage': Percentage of missing values in each column.
    """
    # No. of unique values for each column:
    unique_counts = df.nunique()
    # Data types of each column:
    data_types = df.dtypes
    # No. of missing (NaN) values in each column:
    missing_counts = df.isnull().sum()
    # Percentage of missing values in each column:
    missing_percentage = 100 * df.isnull().mean()
    # Concatenate the above metrics:
    summary_df = pd.concat([unique_counts, data_types, missing_counts, missing_percentage], axis=1)
    # Rename the columns for better readibility
    summary_df.columns = ['unique_count', 'data_types', 'missing_counts', 'missing_percentage']
    # Return summary df
    return summary_df
```

```python
def clean_label(label):
    # Replace any non-alphabetic or non-numeric characters with nothing
    label = re.sub(r'[^a-zA-Z0-9\s]', '', label)
    # Replace spaces with underscores
    label = re.sub(r'\s+', '_', label)
    return label

#-----------------------------------------------------------------------------------------------------------------#
def value_counts_with_percentage(df, column_name):
    # Calculate value counts
    counts = df[column_name].value_counts(dropna=False)

    # Calculate percentages
    percentages = df[column_name].value_counts(dropna=False, normalize=True) * 100

    # Combine counts and percentages into a DataFrame
    result = pd.DataFrame({
        'Count': counts,
        'Percentage': percentages
    })

    return result
#-----------------------------------------------------------------------------------------------------------------#

def plot_horizontal_stacked_bar(df, categorical_cols, target):
    """
    Plots horizontal stacked bar charts for categorical variables against the target variable.
    """
    for col in categorical_cols:
        # Create a crosstab
        crosstab = pd.crosstab(df[col], df[target])

        # Determine if there are six or more categories
        many_categories = len(crosstab) >= 6

        # Plot
        fig, ax = plt.subplots(figsize=(16, 6))  # Increase the width of the figure
        crosstab.plot(kind='barh', stacked=True, color=['green', 'red'], ax=ax)
        ax.set_title(f'{col} distribution by {target}')
        ax.set_xlabel('Count')
        ax.set_ylabel(col)
        ax.grid(True, axis='x')
        ax.set_axisbelow(True)  # Grid lines behind bars

        # Add labels outside the bars
        for i in range(len(crosstab)):
            total = sum(crosstab.iloc[i])
            if col == target:
                label = f"{crosstab.iloc[i].sum() / 1000:.1f}K"
                ax.text(total + 5000, i, label, ha='left', va='center', color='black')
            else:
                label_no = f"No({crosstab.iloc[i, 0] / 1000:.1f}K)"
                label_yes = f"yes({crosstab.iloc[i, 1] / 1000:.1f}K)"
                if many_categories:
                    # Labels next to each other
                    ax.text(total + 5000, i, f'{label_no}, {label_yes}', ha='left', va='center', color='black')
                else:
                    # Labels on top of each other, centered
                    ax.text(total + 5000, i + 0.15, f'{label_no}', ha='left', va='center', color='black')
                    ax.text(total + 5000, i - 0.15, f'{label_yes}', ha='left', va='center', color='black')

        # Adjust the limits to ensure labels fit
        ax.set_xlim(right=ax.get_xlim()[1] + 100000)

        # Move the legend outside of the plot area
        ax.legend(title=target, loc='center left', bbox_to_anchor=(1, 0.5))

        # Ensure labels and plot area fit within the figure
        plt.tight_layout(rect=[0, 0, 0.85, 1])
        plt.show()
```

Importing dataset

Contents

[3]:
```
#First, let's load the cleaned dataset "Data Wrangling output dataset":
df = pd.read_csv('brfss2022_data_wrangling_output.csv')
```

Validating the dataset

Contents

[4]:
```
# Now, Let's Look at the top 5 rows of the df:
df.head()
```

[4]:

	heart_disease	gender	race	general_health	health_care_provider	could_not_afford_to_see_doctor	length_of_time_since_last_routine_checkup	ever_di
0	no	female	white_only_non_hispanic	very_good	yes_only_one	no	past_year	
1	no	male	white_only_non_hispanic	excellent	more_than_one	no	never	
2	no	male	white_only_non_hispanic	very_good	yes_only_one	no	past_year	
3	no	female	white_only_non_hispanic	excellent	yes_only_one	no	past_year	
4	no	male	white_only_non_hispanic	fair	more_than_one	no	past_year	

[5]:
```
#now, Let's Look at the shape of df:
shape = df.shape
print("Number of rows:", shape[0], "\nNumber of columns:", shape[1])
```

```
Number of rows: 445132
Number of columns: 23
```

Heart Disease related features

Contents

After several days of research and analysis of the dataset's features, we have identified the following key features for heart disease assessment:

- **Target Variable (Dependent Variable):**
  - Heart_disease: "Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease"
- **Demographics:**
  - Gender
  - Race
  - Age_category
- **Medical History:**
  - General_Health
  - Have_Personal_Health_Care_Provider
  - Could_Not_Afford_To_See_Doctor
  - Length_of_time_since_last_routine_checkup
  - Ever_Diagnosed_with_Heart_Attack
  - Ever_Diagnosed_with_a_Stroke
  - Ever_told_you_had_a_depressive_disorder
  - Ever_told_you_have_kidney_disease
  - Ever_told_you_had_diabetes
  - BMI
  - Difficulty_Walking_or_Climbing_Stairs

- Physical_Health_Status
- Mental_Health_Status
- Asthma_Status

- **Life Style:**
  - Smoking_status
  - Binge_Drinking_status
  - Drinks_category
  - Exercise_in_Past_30_Days
  - Sleep_category

[6]:
```
#Let's run below to examin each features again missing data count & percentage, unique count, data types:
summarize_df(df)
```

[6]:

	unique_count	data_types	missing_counts	missing_percentage
heart_disease	2	object	0	0.0
gender	3	object	0	0.0
race	7	object	0	0.0
general_health	5	object	0	0.0
health_care_provider	3	object	0	0.0
could_not_afford_to_see_doctor	2	object	0	0.0
length_of_time_since_last_routine_checkup	5	object	0	0.0
ever_diagnosed_with_heart_attack	2	object	0	0.0
ever_diagnosed_with_a_stroke	2	object	0	0.0
ever_told_you_had_a_depressive_disorder	2	object	0	0.0
ever_told_you_have_kidney_disease	2	object	0	0.0
ever_told_you_had_diabetes	4	object	0	0.0
BMI	4	object	0	0.0
difficulty_walking_or_climbing_stairs	2	object	0	0.0
physical_health_status	3	object	0	0.0
mental_health_status	3	object	0	0.0
asthma_Status	3	object	0	0.0
smoking_status	4	object	0	0.0
binge_drinking_status	2	object	0	0.0
exercise_status_in_past_30_Days	2	object	0	0.0
age_category	13	object	0	0.0
sleep_category	5	object	0	0.0
drinks_category	6	object	0	0.0

Converting features data type

Contents

In pandas, the object data type is used for text or mixed data. When a column contains categorical data, it's often beneficial to explicitly convert it to the category data type. Here are some reasons why:

Benefits of Converting to Categorical Type:

- Memory Efficiency: Categorical data types are more memory efficient. Instead of storing each unique string separately, pandas stores the categories and uses integer codes to represent the values.
- Performance Improvement: Operations on categorical data can be faster since pandas can make use of the underlying integer codes.
- Explicit Semantics: Converting to category makes the data's categorical nature explicit, improving code readability and reducing the risk of treating categorical data as continuous.

[7]:
```
# Convert columns to categorical
categorical_columns = df.columns  # assuming all columns need to be categorical
df[categorical_columns] = df[categorical_columns].astype('category')

summarize_df(df)
```

[7]:

	unique_count	data_types	missing_counts	missing_percentage
heart_disease	2	category	0	0.0
gender	3	category	0	0.0
race	7	category	0	0.0
general_health	5	category	0	0.0
health_care_provider	3	category	0	0.0
could_not_afford_to_see_doctor	2	category	0	0.0
length_of_time_since_last_routine_checkup	5	category	0	0.0
ever_diagnosed_with_heart_attack	2	category	0	0.0
ever_diagnosed_with_a_stroke	2	category	0	0.0
ever_told_you_had_a_depressive_disorder	2	category	0	0.0
ever_told_you_have_kidney_disease	2	category	0	0.0
ever_told_you_had_diabetes	4	category	0	0.0
BMI	4	category	0	0.0
difficulty_walking_or_climbing_stairs	2	category	0	0.0
physical_health_status	3	category	0	0.0
mental_health_status	3	category	0	0.0
asthma_Status	3	category	0	0.0
smoking_status	4	category	0	0.0
binge_drinking_status	2	category	0	0.0
exercise_status_in_past_30_Days	2	category	0	0.0
age_category	13	category	0	0.0
sleep_category	5	category	0	0.0
drinks_category	6	category	0	0.0

Alright, now all our features are categorical, let's move to the next step

Analyzing categorical feature distributions against a target variable

Contents

In data analysis, understanding the distribution of categorical features in relation to a target variable is crucial for gaining insights into the data. One effective way to achieve this is by using horizontal stacked bar charts. These visualizations allow us to see how different categories of a feature are distributed across the levels of the target variable, providing a clear view of relationships and patterns within the data.

[8]:
```python
def plot_horizontal_stacked_bar(df, categorical_cols, target):
    """
    Plots horizontal stacked bar charts for categorical variables against the target variable.
    """
    for col in categorical_cols:
        # Create a crosstab
        crosstab = pd.crosstab(df[col], df[target])

        # Determine if there are six or more categories
        many_categories = len(crosstab) >= 6

        # Plot
        fig, ax = plt.subplots(figsize=(16, 6))  # Increase the width of the figure
        crosstab.plot(kind='barh', stacked=True, color=['green', 'red'], ax=ax)
        ax.set_title(f'{col} distribution by {target}')
        ax.set_xlabel('Count')
        ax.set_ylabel(col)
        ax.grid(True, axis='x')
        ax.set_axisbelow(True)  # Grid lines behind bars

        # Add labels outside the bars
        for i in range(len(crosstab)):
            total = sum(crosstab.iloc[i])
            if col == target:
                label = f"{crosstab.iloc[i].sum() / 1000:.1f}K"
                ax.text(total + 5000, i, label, ha='left', va='center', color='black')
            else:
                label_no = f"No({crosstab.iloc[i, 0] / 1000:.1f}K)"
                label_yes = f"yes({crosstab.iloc[i, 1] / 1000:.1f}K)"
                if many_categories:
                    # Labels next to each other
                    ax.text(total + 5000, i, f'{label_no}, {label_yes}', ha='left', va='center', color='black')
                else:
                    # Labels on top of each other, centered
                    ax.text(total + 5000, i + 0.15, f'{label_no}', ha='left', va='center', color='black')
                    ax.text(total + 5000, i - 0.15, f'{label_yes}', ha='left', va='center', color='black')

        # Adjust the limits to ensure labels fit
        ax.set_xlim(right=ax.get_xlim()[1] + 100000)

        # Move the legend outside of the plot area
        ax.legend(title=target, loc='center left', bbox_to_anchor=(1, 0.5))

        # Ensure labels and plot area fit within the figure
        plt.tight_layout(rect=[0, 0, 0.85, 1])
        plt.show()

# Define the list of categorical columns
categorical_cols = df.select_dtypes(include=['category', 'object']).columns.tolist()

# Define the target variable
target = 'heart_disease'

# Call the function
plot_horizontal_stacked_bar(df, categorical_cols, target)
```

Correlation: Heart Disease vs all features

Contents

In this analysis, we aim to understand the relationships between heart disease and various other features in our dataset. Specifically, we will be focusing on three main tasks:

- Encoding Categorical Variables: Converting all categorical features into numerical values using CatBoost encoding. This step ensures that we can effectively use these features in our analysis. Process: convert all categorical features to numerical values using the `CatBoost encoder` from the `category_encoders` package. This encoding method handles categorical variables effectively, preserving their informational content.
- Calculating Mutual Information: Assessing the predictive power of each feature with respect to heart disease by calculating the mutual information. Mutual information measures the dependency between the features and the target variable. Process: we calculate the mutual information for each feature with respect to heart disease. `Mutual information` provides a measure of the dependency between variables, allowing us to identify which features have the most predictive power.
- Calculating Pearson Correlation: Generating a heatmap to visualize the Pearson correlation coefficients between heart disease and all other features. This helps us understand the linear relationships in the dataset, although it is less informative for a binary target. Process: we calculate the `Pearson correlation` coefficients between heart disease and all other features. We visualize these correlations using a heatmap, which helps us easily identify strong linear relationships in the dataset.

[9]:
```
df.head()
```

[9]:

	heart_disease	gender	race	general_health	health_care_provider	could_not_afford_to_see_doctor	length_of_time_since_last_routine_checkup	ever_di
0	no	female	white_only_non_hispanic	very_good	yes_only_one	no	past_year	
1	no	male	white_only_non_hispanic	excellent	more_than_one	no	never	
2	no	male	white_only_non_hispanic	very_good	yes_only_one	no	past_year	
3	no	female	white_only_non_hispanic	excellent	yes_only_one	no	past_year	
4	no	male	white_only_non_hispanic	fair	more_than_one	no	past_year	

[10]:
```
df.columns
```

[10]:
```
Index(['heart_disease', 'gender', 'race', 'general_health',
       'health_care_provider', 'could_not_afford_to_see_doctor',
       'length_of_time_since_last_routine_checkup',
       'ever_diagnosed_with_heart_attack', 'ever_diagnosed_with_a_stroke',
       'ever_told_you_had_a_depressive_disorder',
       'ever_told_you_have_kidney_disease', 'ever_told_you_had_diabetes',
       'BMI', 'difficulty_walking_or_climbing_stairs',
       'physical_health_status', 'mental_health_status', 'asthma_Status',
       'smoking_status', 'binge_drinking_status',
       'exercise_status_in_past_30_Days', 'age_category', 'sleep_category',
       'drinks_category'],
      dtype='object')
```

Features Selection

Contents

[11]:
```
# Define the target variable:
target = 'heart_disease'

# Convert the target variable to numerical values:
df[target] = df[target].apply(lambda x: 1 if x == 'yes' else 0).astype('int')
```

[12]:
```
df.head()
```

[12]:

	heart_disease	gender	race	general_health	health_care_provider	could_not_afford_to_see_doctor	length_of_time_since_last_routine_checkup	ever_di
0	0	female	white_only_non_hispanic	very_good	yes_only_one	no	past_year	
1	0	male	white_only_non_hispanic	excellent	more_than_one	no	never	
2	0	male	white_only_non_hispanic	very_good	yes_only_one	no	past_year	
3	0	female	white_only_non_hispanic	excellent	yes_only_one	no	past_year	
4	0	male	white_only_non_hispanic	fair	more_than_one	no	past_year	

[13]:
```
#Let's define slect our features:
features = [ 'gender', 'race', 'general_health',
           'health_care_provider', 'could_not_afford_to_see_doctor',
           'length_of_time_since_last_routine_checkup',
           'ever_diagnosed_with_heart_attack', 'ever_diagnosed_with_a_stroke',
           'ever_told_you_had_a_depressive_disorder',
           'ever_told_you_have_kidney_disease', 'ever_told_you_had_diabetes',
           'BMI', 'difficulty_walking_or_climbing_stairs',
           'physical_health_status', 'mental_health_status', 'asthma_Status',
           'smoking_status', 'binge_drinking_status',
           'exercise_status_in_past_30_Days', 'age_category', 'sleep_category',
           'drinks_category']

# Separate the features and target
X = df[features]
y = df['heart_disease']
```

Categorical Encoding with Catboost

Contents

Many machine learning algorithms require data to be numeric. Therefore, before training a model or calculating the correlation (Pearson) or mutual information (prediction power), we need to convert categorical data into numeric form. Various categorical encoding methods are available, and CatBoost is one of them. CatBoost is a target-based categorical encoder. It is a supervised encoder that encodes categorical columns according to the target value, supporting both binomial and continuous targets.

Target encoding is a popular technique used for categorical encoding. It replaces a categorical feature with average value of target corresponding to that category in training dataset combined with the target probability over the entire dataset. But this introduces a target leakage since the target is used to predict the target. Such models tend to be overfitted and don't generalize well in unseen circumstances.

A CatBoost encoder is similar to target encoding, but also involves an ordering principle in order to overcome this problem of target leakage. It uses the principle similar to the time series data validation. The values of target statistic rely on the observed history, i.e. target probability for the current feature is calculated only from the rows (observations) before it.

[14]:
```
# Initialize the CatBoost encoder
cbe_encoder = ce.CatBoostEncoder()

# Fit and transform the dataset
cbe_encoder.fit(X,y)

# Replace the original categorical columns with encoded columns:
X_cbe = cbe_encoder.transform(X)

#train, test, split
X_train, X_test, y_train, y_test = train_test_split(X_cbe,
                                                    y,
                                                    test_size=0.20,
                                                    random_state=1981)
```

Mutual Information - Prediction Power

Contents

Mutual Information (MI) is a measure of the mutual dependence between two variables. It quantifies the amount of information obtained about one variable through another variable. Unlike correlation, which only captures linear relationships, mutual information can capture both linear and non-linear relationships between variables, making it a powerful tool for feature selection in machine learning.

Mutual Information key advanatages:

- Captures Non-Linear Relationships: Unlike traditional correlation measures (e.g., Pearson), mutual information can capture complex, non-linear relationships between features and the target variable. This is particularly useful in real-world datasets where relationships are rarely purely linear.
- Independence Detection: A mutual information score of zero indicates that two variables are completely independent. Non-zero mutual information indicates some level of dependency.
- Predictive Power: Higher mutual information scores suggest that a feature contains more information about the target variable, indicating higher predictive power. This helps in identifying the most relevant features for building robust predictive models.

[15]:
```
# Calculate mutual information
MI_score = mutual_info_classif(X_train,
                               y_train,
                               random_state=1981)
```

[16]:
```
# Restructure the mutual information values:
MI_score = pd.Series(MI_score, index=X_train.columns)
MI_score = MI_score.sort_values(ascending=True)
```

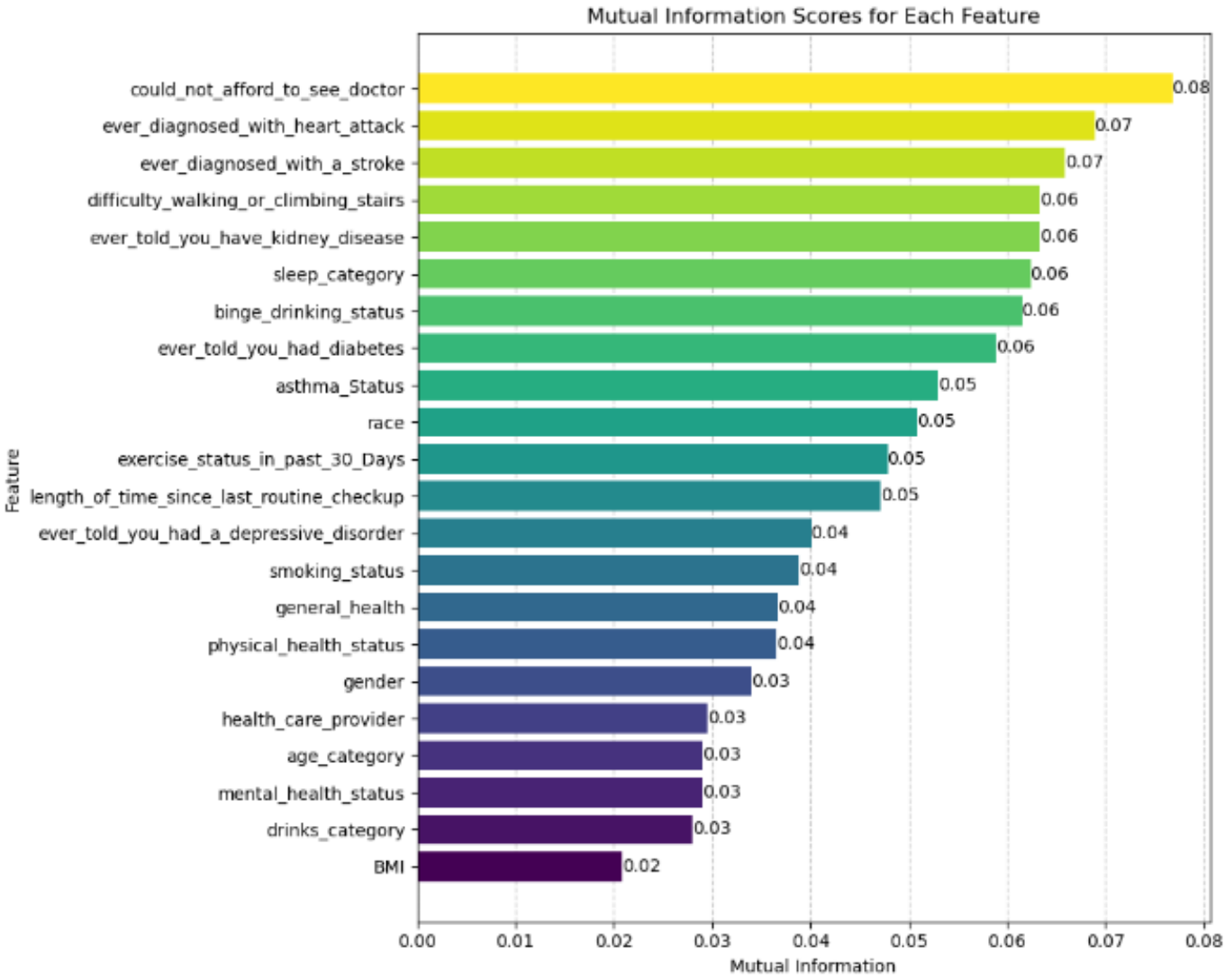
[17]:
```
# Visualize mutual information scores as horizontal bar charts
plt.figure(figsize=(10, 8))
colors = plt.cm.viridis(np.linspace(0, 1, len(MI_score)))
bars = plt.barh(MI_score.index, MI_score.values, color=colors)

# Add grid lines behind the bars
plt.gca().set_axisbelow(True)
plt.grid(axis='x', linestyle='--', alpha=0.7)

# Add titles and labels
plt.title('Mutual Information Scores for Each Feature')
plt.xlabel('Mutual Information')
plt.ylabel('Feature')

# Add data labels
for bar in bars:
    plt.text(bar.get_width(), bar.get_y() + bar.get_height()/2,
             f'{bar.get_width():.2f}',
             va='center', ha='left', color='black')

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



Interpretation of Mutual Information Scores

Contents

Mutual information measures the dependency between variables, capturing both linear and non-linear relationships. In this context, the mutual information score indicates how much information about the target variable heart_disease is provided by each feature. Higher scores imply stronger relationships between the feature and the target variable.

Top Features with High Mutual Information Scores

- could_not_afford_to_see_doctor (0.08):

This feature has the highest mutual information score, suggesting that the inability to afford to see a doctor is the most informative feature for predicting heart disease. This indicates a significant association between financial barriers to healthcare and the likelihood of having heart disease.

- ever_diagnosed_with_heart_attack (0.07):

Being previously diagnosed with a heart attack is highly informative for predicting heart disease. This is expected, as a history of heart attacks is a strong indicator of ongoing heart health issues.

- ever_diagnosed_with_a_stroke (0.07):

- ever_diagnosed_with_heart_attack (0.07):

Being previously diagnosed with a heart attack is highly informative for predicting heart disease. This is expected, as a history of heart attacks is a strong indicator of ongoing heart health issues.

- ever_diagnosed_with_a_stroke (0.07):

Similarly, a history of strokes is another strong predictor of heart disease, suggesting that individuals with a history of stroke are more likely to have heart disease.

- difficulty_walking_or_climbing_stairs (0.06):

Difficulty in physical activities like walking or climbing stairs is closely related to heart disease, likely reflecting the physical limitations caused by heart health issues.

- ever_told_you_have_kidney_disease (0.06):

Kidney disease is significantly associated with heart disease, which aligns with the known comorbidities between cardiovascular and renal health issues.

- sleep_category (0.06):

Sleep patterns or quality of sleep also show a notable relationship with heart disease, potentially indicating that poor sleep may be a risk factor.

- binge_drinking_status (0.06):

Binge drinking is another informative feature, suggesting a relationship between alcohol consumption behaviors and heart disease risk.

- ever_told_you_had_diabetes (0.06):

Diabetes is a well-known risk factor for heart disease, and its high mutual information score reflects this strong association.

- asthma_Status (0.05):

The presence of asthma also provides information about the likelihood of having heart disease, possibly due to shared risk factors or comorbid conditions.

Features with Moderate Mutual Information Scores

- race (0.05), exercise_status_in_past_30_Days (0.05), length_of_time_since_last_routine_checkup (0.05):

These features have moderate mutual information scores, indicating that they contribute useful but less substantial information about heart disease risk.

- ever_told_you_had_a_depressive_disorder (0.04), smoking_status (0.04), general_health (0.04), physical_health_status (0.04):

Mental health, smoking status, and general physical health also show meaningful associations with heart disease, reflecting the multifaceted nature of heart disease risk factors.

Features with Lower Mutual Information Scores

- gender (0.03), health_care_provider (0.03), age_category (0.03), mental_health_status (0.03), drinks_category (0.03), BMI (0.02): These features have lower mutual information scores, suggesting that while they do provide some information, their individual contributions to predicting heart disease are relatively smaller compared to the top features.

Summary

The mutual information scores provide a quantitative measure of how much each feature contributes to predicting heart disease. Features like financial barriers to healthcare, history of heart attacks or strokes, and physical limitations have the highest scores, highlighting their strong associations with heart disease. Understanding these relationships can help in prioritizing features for further analysis and in developing predictive models for heart disease.

Pearson Correlation

Contents

Pearson correlation, also known as Pearson's r, is a measure of the linear relationship between two continuous variables. It quantifies the degree to which a pair of variables are linearly related. The Pearson correlation coefficient can take values between -1 and 1, where:

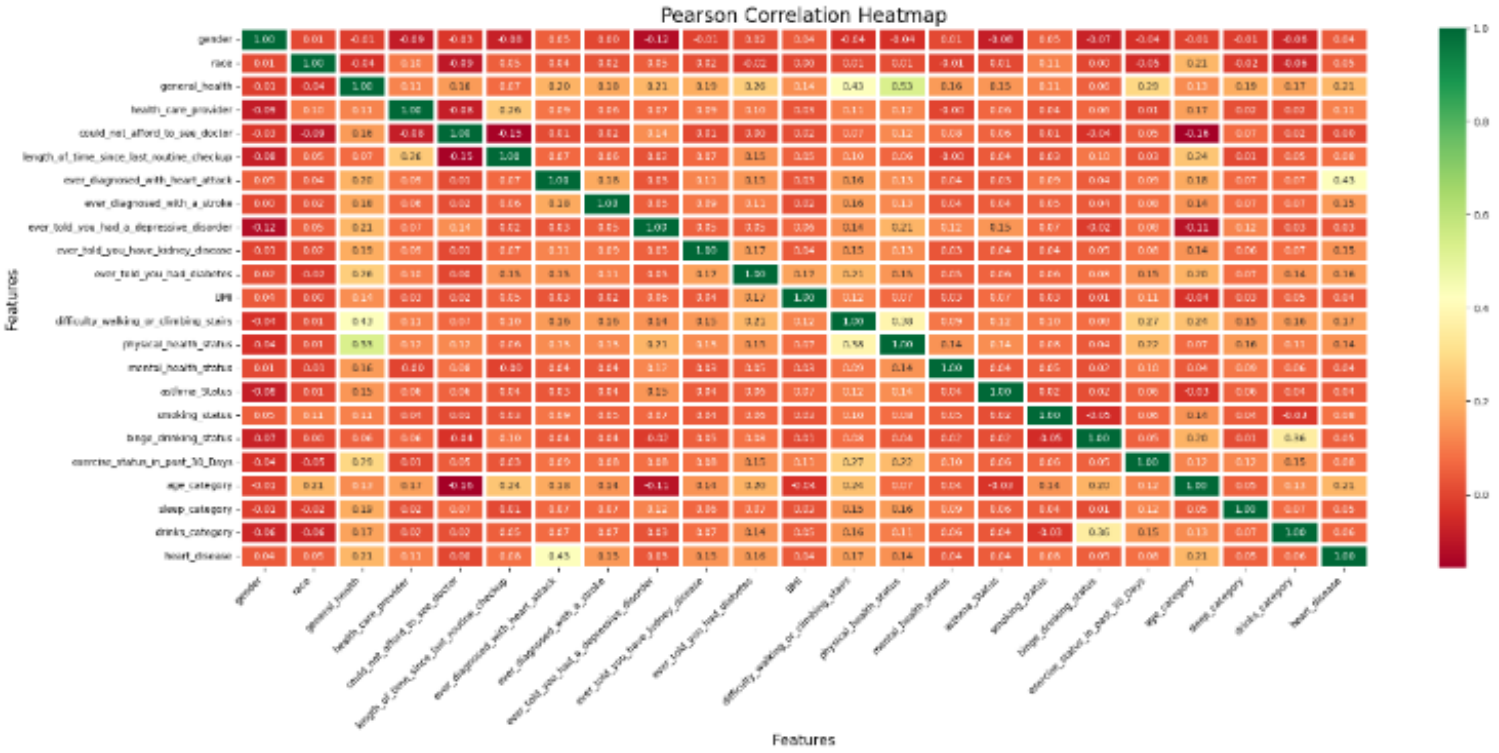
- +1 indicates a perfect positive linear relationship.
- -1 indicates a perfect negative linear relationship.
- 0 indicates no linear relationship.

[18]:
```
# Combine X_cbe (encoded features) and y (target) into a new DataFrame
encoded_df = X_cbe.copy()
encoded_df['heart_disease'] = y

corr = encoded_df.corr()
```

[19]:
```
# Now, let's build a heatmap:

plt.figure(figsize=(26, 10))
sns.heatmap(corr, linewidths=4, annot=True, fmt=".2f", cmap="RdYlGn")
plt.title('Pearson Correlation Heatmap', fontsize=20)
plt.xlabel('Features', fontsize=15)
plt.ylabel('Features', fontsize=15)
plt.xticks(rotation=45, ha='right')
plt.yticks(rotation=0)
#plt.tight_layout()
plt.show()
```



Collinearity Interpretation

Contents

The heatmap displays the Pearson correlation coefficients between various features, focusing on identifying independent features that exhibit high collinearity with each other. Here are the key observations regarding collinearity:

High Collinearity

- General Health and Physical Health Status (0.53): There is a strong positive correlation between general health and physical health status. This suggests that these two features are closely related, with better general health strongly associated with better physical health status.
- Difficulty Walking or Climbing Stairs and Physical Health Status (0.43): A strong positive correlation indicates that difficulty in physical activities is a significant

High Collinearity

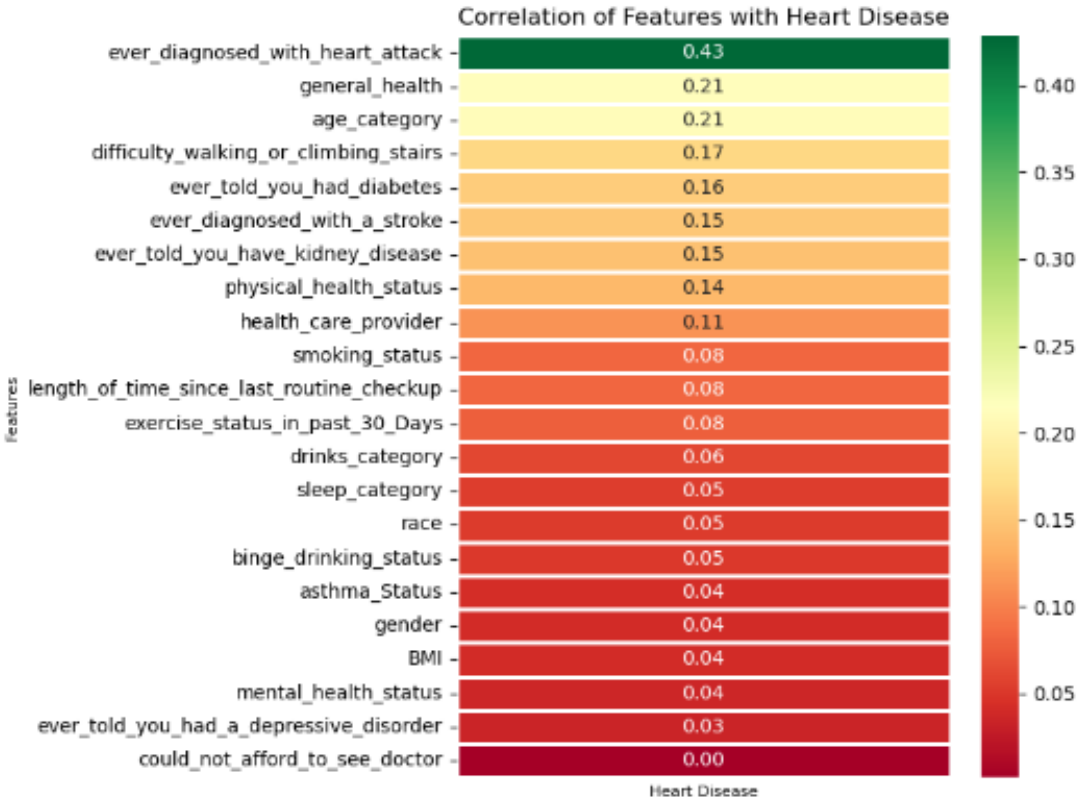
- General Health and Physical Health Status (0.53): There is a strong positive correlation between general health and physical health status. This suggests that these two features are closely related, with better general health strongly associated with better physical health status.
- Difficulty Walking or Climbing Stairs and Physical Health Status (0.43): A strong positive correlation indicates that difficulty in physical activities is a significant component of overall physical health.
- General Health and Difficulty Walking or Climbing Stairs (0.29): Moderate collinearity suggests that general health is significantly influenced by the ability to perform physical activities.
- Ever Told You Had Diabetes and General Health (0.26): Moderate collinearity suggests that diabetes status is an important factor in overall general health.
- Ever Told You Had Diabetes and Physical Health Status (0.26): Moderate collinearity indicates that diabetes has a considerable impact on physical health status.
- Length of Time Since Last Routine Checkup and General Health (0.26): Moderate collinearity suggests that routine checkups are related to overall health monitoring.
- General Health and Exercise Status in Past 30 Days (0.29): * Moderate collinearity suggests that exercise is an important factor in maintaining general health.

Moderate Collinearity

- Ever Diagnosed with Heart Attack and Ever Diagnosed with a Stroke (0.18): There is moderate collinearity, indicating that these conditions often occur together.
- Ever Told You Had Diabetes and Ever Diagnosed with Heart Attack (0.20): Moderate collinearity indicates a common comorbidity with heart disease.
- Ever Diagnosed with a Stroke and Ever Diagnosed with Heart Attack (0.18): Moderate collinearity indicating a common risk factor.
- Ever Told You Have Kidney Disease and Ever Told You Had Diabetes (0.19): Moderate collinearity suggests common comorbidities.
- Ever Diagnosed with Heart Attack and Ever Told You Had Kidney Disease (0.15): Moderate collinearity indicates a link between kidney disease and heart disease.
- Health Care Provider and General Health (0.26): Moderate collinearity indicates a link between having a healthcare provider and general health.
- Health Care Provider and Could Not Afford to See Doctor (0.16): Moderate collinearity suggests financial barriers to healthcare impact general health.
- Smoking Status and Binge Drinking Status (0.11): Weak collinearity suggests some lifestyle factors are related.

[20]:
```
# Extract correlations with the target variable:
correlation_with_target = corr[['heart_disease']].drop(index='heart_disease').sort_values(by='heart_disease', ascending=False)

# Visualize the correlation with the target variable using a heatmap:
plt.figure(figsize=(8, 6))
sns.heatmap(correlation_with_target, annot=True, fmt=".2f", cmap="RdYlGn", cbar=True, linewidths=2)
plt.title('Correlation of Features with Heart Disease', fontsize=12)
plt.xlabel('Heart Disease', fontsize=8)
plt.ylabel('Features', fontsize=8)
plt.xticks(rotation=45, ha='right')
plt.xticks(ticks=[], labels=[], rotation=45, ha='right')
plt.yticks(rotation=0)
plt.tight_layout()
plt.show()
```



Target Variable Interpretation

Contents

The heatmap displays the Pearson correlation coefficients between various features and the target variable heart_disease. The values range from `-1` to `1`, where values closer to `1` indicate a strong positive linear correlation, values closer to `-1` indicate a strong negative linear correlation, and values around `0` indicate no linear correlation. Here are the key observations:

Strongest Positive Linear Correlations

- Ever Diagnosed with Heart Attack (0.43): Interpretation: This feature has the highest positive linear correlation with heart disease, indicating that individuals who have had a heart attack are significantly more likely to have heart disease. This is a strong relationship and aligns with medical knowledge that a history of heart attacks is a major risk factor for heart disease.

Moderate Positive Linear Correlations

- General Health (0.21): Interpretation: There is a moderate positive linear correlation between general health and heart disease. Poorer general health is associated with a higher likelihood of heart disease.
- Age Category (0.21): Interpretation: Age category has a moderate positive linear correlation with heart disease, suggesting that older individuals are more likely to have heart disease.
- Difficulty Walking or Climbing Stairs (0.17): Interpretation: This feature indicates that individuals who report difficulty with physical activities are more likely to have heart disease.
- Ever Told You Had Diabetes (0.16): Interpretation: There is a moderate positive linear correlation, indicating that individuals with diabetes are more likely to have heart disease.
- Ever Diagnosed with a Stroke (0.15): Interpretation: This feature has a moderate positive linear correlation with heart disease, suggesting that individuals who have had a stroke are also at higher risk for heart disease.
- Ever Told You Have Kidney Disease (0.15): Interpretation: There is a moderate positive linear correlation, indicating a link between kidney disease and heart disease.
- Physical Health Status (0.14): Interpretation: Individuals reporting poor physical health status are more likely to have heart disease.

Weak Positive Linear Correlations

- Health Care Provider (0.11): Interpretation: A weak positive linear correlation, suggesting some association between having a healthcare provider and heart disease, potentially due to increased diagnosis rates.
- Smoking Status (0.08): Interpretation: There is a weak positive linear correlation between smoking and heart disease, reflecting the known risk of smoking for cardiovascular conditions.
- Length of Time Since Last Routine Checkup (0.08): Interpretation: A weak positive linear correlation suggests that longer intervals between checkups are slightly associated with heart disease.
- Exercise Status in Past 30 Days (0.08): Interpretation: A weak positive linear correlation indicating that less frequent exercise might be associated with heart disease.
- Drinks Category (0.06): Interpretation: There is a weak positive linear correlation between drinking status and heart disease.
- Sleep Category (0.05): Interpretation: There is a weak positive linear correlation between sleep category and heart disease, suggesting that poor sleep might be associated with heart disease.
- Race (0.05): Interpretation: A weak positive linear correlation indicating a slight association between race and heart disease.
- Binge Drinking Status (0.05): Interpretation: A weak positive linear correlation between binge drinking and heart disease.
- Asthma Status (0.04): Interpretation: A weak positive linear correlation between asthma status and heart disease.
- Gender (0.04): Interpretation: A weak positive linear correlation between gender and heart disease.
- BMI (0.04): Interpretation: A weak positive linear correlation between BMI and heart disease.
- Mental Health Status (0.04): Interpretation: A weak positive linear correlation between mental health status and heart disease.
- Ever Told You Had a Depressive Disorder (0.03): Interpretation: A weak positive linear correlation suggesting a slight association between depression and heart disease.
- Could Not Afford to See Doctor (0.00): Interpretation: This feature has a negligible linear correlation with heart disease, indicating that financial barriers to healthcare do not have a significant direct linear correlation with heart disease in this dataset.

Summary The heatmap reveals that the strongest predictor of heart disease is a history of heart attacks, followed by general health, age category, and difficulty with physical activities. Other features show weaker linear correlations, suggesting they have a less direct but still notable relationship with heart disease. These insights can be useful for identifying risk factors and guiding further analysis and modeling efforts.

Comparison Between Pearson Correlation and Mutual Information

Contents

Comparison

- Linear Relationships: Pearson correlation is effective for identifying linear relationships. "Ever Diagnosed with Heart Attack" shows the highest linear correlation with heart disease, indicating a direct linear dependency.
- Non-Linear Relationships: Mutual information captures both linear and non-linear dependencies. "Could Not Afford to See Doctor" shows the highest score, suggesting that financial barriers, although not strongly linearly correlated, have a significant impact on heart disease prediction.
- Overlap: Some features like "Ever Diagnosed with Heart Attack" and "General Health" are significant in both Pearson correlation and mutual information, indicating their strong overall predictive power.
- Unique Insights: Mutual information highlights features like "Could Not Afford to See Doctor" and "Sleep Category" that Pearson correlation does not emphasize, suggesting these features have non-linear relationships with heart disease.

Conclusion

- Pearson correlation is useful for identifying strong linear relationships but might miss non-linear dependencies.
- Mutual information provides a more comprehensive view by capturing both linear and non-linear relationships, highlighting features that might not be evident with Pearson correlation alone.
- Using both methods provides a holistic understanding of the relationships between independent features and the target variable, aiding in better feature selection for predictive modeling.

Modeling: AI-Powered Heart Disease Risk Assessment

- **Name:** Aktham Almomani
- **Course:** Probability and Statistics for Artificial Intelligence (MS-AAI-500-02) / University Of San Diego
- **Semester:** Summer 2024
- **Group:** 8

Contents

Introduction

Contents

In this notebook, we will be fitting and evaluating multiple machine learning models to classify heart disease:

- Logistic Regression
- Random Forest
- XGBoost
- LightGBM
- Balanced Bagging
- Easy Ensemble
- Balanced Random Forest
- Balanced Bagging (LightGBM): Balanced Bagging as a Wrapper and LightGBM as a base estimator
- Easy Ensemble (LightGBM): Easy Ensemble as a Wrapper and LightGBM as a base estimator

Our goal is to accurately predict heart disease risk using these models. We will employ hyperparameter tuning with `Optuna` to optimize each model's performance. Additionally, we will leverage the `BalancedRandomForestClassifier`, `BalancedBaggingClassifier` and `EasyEnsembleClassifier` from the `imbalanced-learn library` to address class imbalance. These classifiers use bootstrapped sampling to balance the dataset, ensuring robust classification of minority classes. By focusing on underrepresented data, it enhances model performance, making it particularly suitable for imbalanced datasets like heart disease prediction.

Through this comprehensive approach, we aim to develop a reliable and effective model for heart disease risk assessment, contributing to better health outcomes.

Dataset

Contents

The dataset used in this Modeling notebook is the result of a comprehensive data wrangling process. Data wrangling is a crucial step in the data science workflow, involving the transformation and preparation of raw data into a more usable format. The main tasks performed during data wrangling included:

- Dealing with missing data
- Data mapping
- Data cleaning
- Feature engineering

These steps ensured that the dataset is well-prepared for analysis and modeling, enabling us to build reliable and robust models for heart disease prediction.

Setup and preliminaries

Contents

Import libraries

Contents

[26]:
```
#Let's import the necessary packages:
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import math
import scipy.stats as stats
from scipy.stats import gamma, linregress
from bs4 import BeautifulSoup
import re

# Machine Learning models
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
import category_encoders as ce
from sklearn.feature_selection import mutual_info_classif
from sklearn.model_selection import train_test_split

# Model evaluation
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix, roc_auc_score, roc_curve
from sklearn.metrics import accuracy_score, classification_report
import shap

# Hyperparameter tuning
import optuna

# Handling class imbalance
from imblearn.ensemble import BalancedRandomForestClassifier, BalancedBaggingClassifier, EasyEnsembleClassifier
from imblearn.metrics import classification_report_imbalanced

# Let's run below to customize notebook display:
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)

# format floating-point numbers to 2 decimal places: we'll adjust below requirement as needed for specific answers during this assignment:
#pd.set_option('float_format', '{:.2f}'.format)

import warnings
warnings.filterwarnings('ignore', category=FutureWarning, module='xgboost')
warnings.filterwarnings('ignore', category=FutureWarning, module='imblearn')
```

Necessary functions

Contents

[2]:
```python
def summarize_df(df):
    """
    Generate a summary DataFrame for an input DataFrame.
    Parameters:
    df (pd.DataFrame): The DataFrame to summarize.
    Returns:
    A datafram: containing the following columns:
              - 'unique_count': No. unique values in each column.
              - 'data_types': Data types of each column.
              - 'missing_counts': No. of missing (NaN) values in each column.
              - 'missing_percentage': Percentage of missing values in each column.
    """
    # No. of unique values for each column:
    unique_counts = df.nunique()
    # Data types of each column:
    data_types = df.dtypes
    # No. of missing (NaN) values in each column:
    missing_counts = df.isnull().sum()
    # Percentage of missing values in each column:
    missing_percentage = 100 * df.isnull().mean()
    # Concatenate the above metrics:
    summary_df = pd.concat([unique_counts, data_types, missing_counts, missing_percentage], axis=1)
    # Rename the columns for better readibility
    summary_df.columns = ['unique_count', 'data_types', 'missing_counts', 'missing_percentage']
    # Return summary df
    return summary_df
#-----------------------------------------------------------------------------------------------------------------#
# Function to clean and format the label
def clean_label(label):
    # Replace any non-alphabetic or non-numeric characters with nothing
    label = re.sub(r'[^a-zA-Z0-9\s]', '', label)
    # Replace spaces with underscores
    label = re.sub(r'\s+', '_', label)
    return label

#-----------------------------------------------------------------------------------------------------------------#
def value_counts_with_percentage(df, column_name):
    # Calculate value counts
    counts = df[column_name].value_counts(dropna=False)

    # Calculate percentages
    percentages = df[column_name].value_counts(dropna=False, normalize=True) * 100

    # Combine counts and percentages into a DataFrame
    result = pd.DataFrame({
        'Count': counts,
        'Percentage': percentages
    })

    return result
```

```
#-----------------------------------------------------------------------------------------------------------------#

def plot_heart_disease_distribution(df, target):
    """
    Plots a horizontal stacked bar chart for the target variable 'Heart Disease'.
    """
    # Create a crosstab
    crosstab = pd.crosstab(df[target], df[target])

    # Plot
    fig, ax = plt.subplots(figsize=(16, 6))  # Increase the width of the figure
    crosstab.plot(kind='barh', stacked=True, color=['green', 'red'], ax=ax)
    ax.set_title(f'{target} Distribution')
    ax.set_xlabel('Count')
    ax.set_ylabel(target)
    ax.grid(True, axis='x')
    ax.set_axisbelow(True)  # Grid lines behind bars

    # Add labels outside the bars
    for i in range(len(crosstab)):
        total = sum(crosstab.iloc[i])
        label_no = f"No({crosstab.iloc[i, 0] / 1000:.1f}K)"
        label_yes = f"Yes({crosstab.iloc[i, 1] / 1000:.1f}K)"
        ax.text(total + 5000, i, f'{label_no}, {label_yes}', ha='left', va='center', color='black')

    # Adjust the limits to ensure labels fit
    ax.set_xlim(right=ax.get_xlim()[1] + 100000)

    # Move the legend outside of the plot area
    ax.legend(title=target, loc='center left', bbox_to_anchor=(1, 0.5))

    # Ensure labels and plot area fit within the figure
    plt.tight_layout(rect=[0, 0, 0.85, 1])
    plt.show()
```

Importing dataset

Contents

[3]:
```
#First, let's load the cleaned dataset "Data Wrangling output dataset":
df = pd.read_csv('brfss2022_data_wrangling_output.csv')
```

Validating The Dataset

Contents

[4]:
```
# Now, let's Look at the top 5 rows of the df:
df.head()
```

[4]:

	heart_disease	gender	race	general_health	health_care_provider	could_not_afford_to_see_doctor	length_of_time_since_last_routine_checkup	ever_di
0	no	female	white_only_non_hispanic	very_good	yes_only_one	no	past_year	
1	no	male	white_only_non_hispanic	excellent	more_than_one	no	never	
2	no	male	white_only_non_hispanic	very_good	yes_only_one	no	past_year	
3	no	female	white_only_non_hispanic	excellent	yes_only_one	no	past_year	
4	no	male	white_only_non_hispanic	fair	more_than_one	no	past_year	

[5]:
```
#now, Let's look at the shape of df:
shape = df.shape
print("Number of rows:", shape[0], "\nNumber of columns:", shape[1])
```

```
Number of rows: 445132
Number of columns: 23
```

Heart Disease related features

Contents

After several days of research and analysis of the dataset's features, we have identified the following key features for heart disease assessment:

- **Target Variable (Dependent Variable):**
  - Heart_disease: "Ever_Diagnosed_with_Angina_or_Coronary_Heart_Disease"
- **Demographics:**
  - Gender
  - Race
  - Age_category
- **Medical History:**
  - General_Health
  - Have_Personal_Health_Care_Provider
  - Could_Not_Afford_To_See_Doctor
  - Length_of_time_since_last_routine_checkup
  - Ever_Diagnosed_with_Heart_Attack
  - Ever_Diagnosed_with_a_Stroke
  - Ever_told_you_had_a_depressive_disorder
  - Ever_told_you_have_kidney_disease
  - Ever_told_you_had_diabetes
  - BMI
  - Difficulty_Walking_or_Climbing_Stairs
  - Physical_Health_Status
  - Mental_Health_Status
  - Asthma_Status
- **Life Style:**
  - Smoking_status
  - Binge_Drinking_status
  - Drinks_category
  - Exercise_in_Past_30_Days
  - Sleep_category

[6]:
```
#Let's run below to examin each features again missing data count & percentage, unique count, data types:
summarize_df(df)
```

[6]:

	unique_count	data_types	missing_counts	missing_percentage
heart_disease	2	object	0	0.0
gender	3	object	0	0.0
race	7	object	0	0.0
general_health	5	object	0	0.0
health_care_provider	3	object	0	0.0
could_not_afford_to_see_doctor	2	object	0	0.0
length_of_time_since_last_routine_checkup	5	object	0	0.0
ever_diagnosed_with_heart_attack	2	object	0	0.0
ever_diagnosed_with_a_stroke	2	object	0	0.0
ever_told_you_had_a_depressive_disorder	2	object	0	0.0
ever_told_you_have_kidney_disease	2	object	0	0.0
ever_told_you_had_diabetes	4	object	0	0.0
BMI	4	object	0	0.0
difficulty_walking_or_climbing_stairs	2	object	0	0.0
physical_health_status	3	object	0	0.0
mental_health_status	3	object	0	0.0
asthma_Status	3	object	0	0.0
smoking_status	4	object	0	0.0
binge_drinking_status	2	object	0	0.0
exercise_status_in_past_30_Days	2	object	0	0.0
age_category	13	object	0	0.0
sleep_category	5	object	0	0.0
drinks_category	6	object	0	0.0

Converting Features Data Type

Contents

In pandas, the object data type is used for text or mixed data. When a column contains categorical data, it's often beneficial to explicitly convert it to the category data type. Here are some reasons why:

Benefits of Converting to Categorical Type:

- Memory Efficiency: Categorical data types are more memory efficient. Instead of storing each unique string separately, pandas stores the categories and uses integer codes to represent the values.
- Performance Improvement: Operations on categorical data can be faster since pandas can make use of the underlying integer codes.
- Explicit Semantics: Converting to category makes the data's categorical nature explicit, improving code readability and reducing the risk of treating categorical data as continuous.

[7]:
```
# Convert columns to categorical
categorical_columns = df.columns  # assuming all columns need to be categorical
df[categorical_columns] = df[categorical_columns].astype('category')

summarize_df(df)
```

[7]:

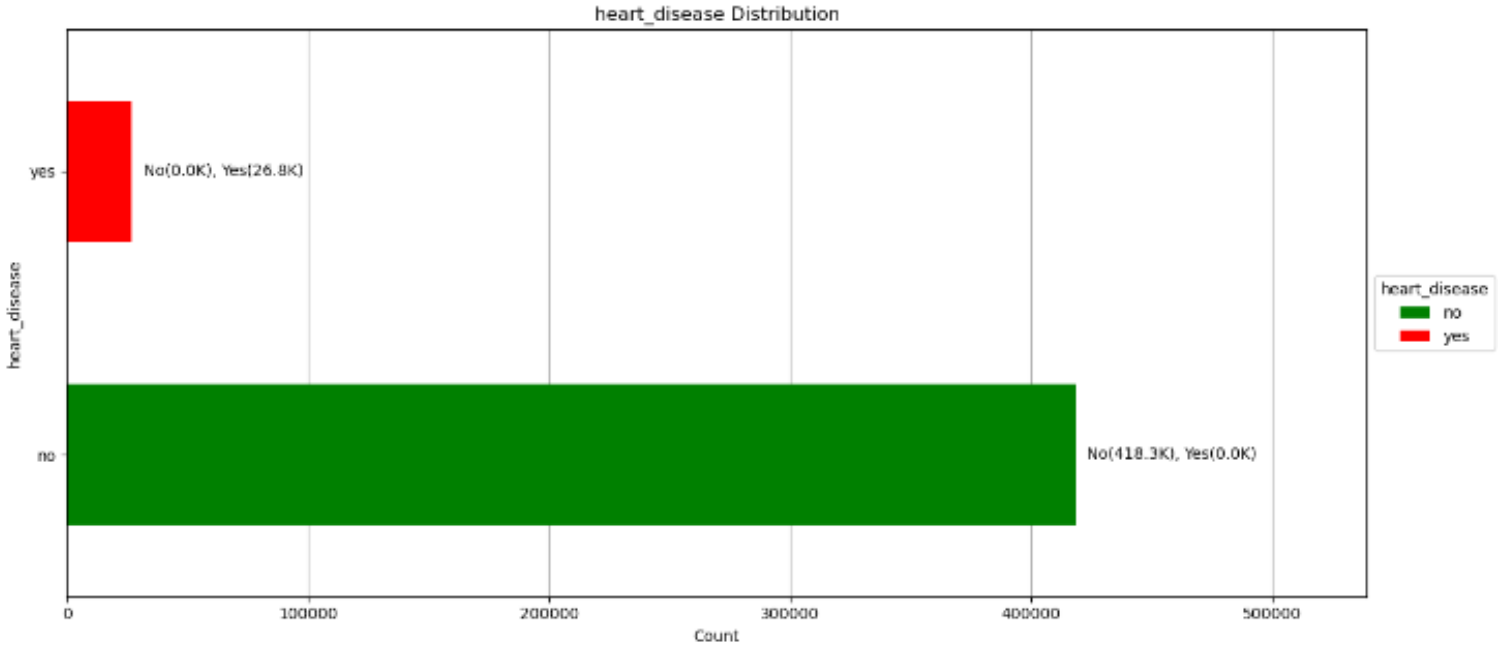
	unique_count	data_types	missing_counts	missing_percentage
heart_disease	2	category	0	0.0
gender	3	category	0	0.0
race	7	category	0	0.0
general_health	5	category	0	0.0
health_care_provider	3	category	0	0.0
could_not_afford_to_see_doctor	2	category	0	0.0
length_of_time_since_last_routine_checkup	5	category	0	0.0
ever_diagnosed_with_heart_attack	2	category	0	0.0
ever_diagnosed_with_a_stroke	2	category	0	0.0
ever_told_you_had_a_depressive_disorder	2	category	0	0.0
ever_told_you_have_kidney_disease	2	category	0	0.0
ever_told_you_had_diabetes	4	category	0	0.0
BMI	4	category	0	0.0
difficulty_walking_or_climbing_stairs	2	category	0	0.0
physical_health_status	3	category	0	0.0
mental_health_status	3	category	0	0.0
asthma_Status	3	category	0	0.0
smoking_status	4	category	0	0.0
binge_drinking_status	2	category	0	0.0
exercise_status_in_past_30_Days	2	category	0	0.0
age_category	13	category	0	0.0
sleep_category	5	category	0	0.0
drinks_category	6	category	0	0.0

Alright, now all our features are categorical, let's move to the next step

Heart Disease: Target Variable

Contents

[8]:
```
# ALright, now, Let's Look at heart disease distribution:
plot_heart_disease_distribution(df, 'heart_disease')
```



Distribution Analysis

- There is a significant imbalance between the two categories.
- A large majority of individuals do not have heart disease `418.3K`, while a much smaller number have heart disease `26.8K`.
- This imbalance can be visually observed in the chart, where the green bar is substantially longer than the red bar.

Imbalance Issue

- Model Bias: When training a classification model on this imbalanced dataset, the model might become biased towards predicting the majority class (No heart disease) more frequently because it is seen more often in the training data.
- Performance Metrics: Common performance metrics like accuracy can be misleading in imbalanced datasets. For instance, a model that always predicts "No heart disease" will have high accuracy because the majority class is well represented. However, this model would fail to correctly identify individuals with heart disease, which is critical for healthcare applications.
- Recall and Precision: Metrics such as recall (sensitivity) and precision are more informative in this context. Recall measures the ability to identify true positive cases (heart disease), while precision measures the accuracy of positive predictions. In an imbalanced dataset, a model might have low recall for the minority class (heart disease) even if it has high accuracy overall.

Strategy to Address Imbalance The `BalancedRandomForestClassifier` or `BalancedBaggingClassifier` or `EasyEnsembleClassifier` from the `imbalanced-learn` library effectively handles class imbalance by using bootstrapped sampling to balance the dataset, ensuring robust classification of minority classes. It enhances model performance by focusing on underrepresented data, making it ideal for imbalanced datasets like heart disease prediction.

Features Selection

Contents

[9]:
```
# Define the target variable:
target = 'heart_disease'

# Convert the target variable to numerical values:
df[target] = df[target].apply(lambda x: 1 if x == 'yes' else 0).astype('int')
```

[10]:
```
df.head()
```

[10]:

	heart_disease	gender	race	general_health	health_care_provider	could_not_afford_to_see_doctor	length_of_time_since_last_routine_checkup	ever_di
0	0	female	white_only_non_hispanic	very_good	yes_only_one	no	past_year	
1	0	male	white_only_non_hispanic	excellent	more_than_one	no	never	
2	0	male	white_only_non_hispanic	very_good	yes_only_one	no	past_year	
3	0	female	white_only_non_hispanic	excellent	yes_only_one	no	past_year	
4	0	male	white_only_non_hispanic	fair	more_than_one	no	past_year	

[11]:
```
#Let's define slect our features:
features = [ 'gender', 'race', 'general_health',
            'health_care_provider', 'could_not_afford_to_see_doctor',
            'length_of_time_since_last_routine_checkup',
            'ever_diagnosed_with_heart_attack', 'ever_diagnosed_with_a_stroke',
            'ever_told_you_had_a_depressive_disorder',
            'ever_told_you_have_kidney_disease', 'ever_told_you_had_diabetes',
            'BMI', 'difficulty_walking_or_climbing_stairs',
            'physical_health_status', 'mental_health_status', 'asthma_Status',
            'smoking_status', 'binge_drinking_status',
            'exercise_status_in_past_30_Days', 'age_category', 'sleep_category',
            'drinks_category']

# Separate the features and target
X = df[features]
y = df['heart_disease']
```

Categorical Encoding with Catboost

Contents

Many machine learning algorithms require data to be numeric. Therefore, before training a model or calculating the correlation (Pearson) or mutual information (prediction power), we need to convert categorical data into numeric form. Various categorical encoding methods are available, and CatBoost is one of them. CatBoost is a target-based categorical encoder. It is a supervised encoder that encodes categorical columns according to the target value, supporting both binomial and continuous targets.

Target encoding is a popular technique used for categorical encoding. It replaces a categorical feature with average value of target corresponding to that category in training dataset combined with the target probability over the entire dataset. But this introduces a target leakage since the target is used to predict the target. Such models tend to be overfitted and don't generalize well in unseen circumstances.

A CatBoost encoder is similar to target encoding, but also involves an ordering principle in order to overcome this problem of target leakage. It uses the principle similar to the time series data validation. The values of target statistic rely on the observed history, i.e, target probability for the current feature is calculated only from the rows (observations) before it.

[12]:
```
# Initialize the CatBoost encoder:
cbe_encoder = ce.CatBoostEncoder()

# Fit and transform the dataset:
cbe_encoder.fit(X,y)

# Replace the original categorical columns with encoded columns:
X_cbe = cbe_encoder.transform(X)
```

Split data into training and testing sets

Contents

[13]:
```
#train, test, split
X_train, X_test, y_train, y_test = train_test_split(X_cbe,
                                                    y,
                                                    test_size=0.20,
                                                    random_state=1981)
```

Scale features for Logistic Regression

Contents

[14]:
```
#Scale features for Logistic Regression:
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Baseline Modeling

Contents

Here, we will fit the the following models listed below and compare their performance at the class-specific level:

- Logistic Regression
- Random Forest
- XGBoost
- LightGBM
- Balanced Bagging
- Easy Ensemble
- Balanced Random Forest
- Balanced Bagging (LightGBM): Balanced Bagging as a Wrapper and LightGBM as a base estimator
- Easy Ensemble (LightGBM): Easy Ensemble as a Wrapper and LightGBM as a base estimator

[15]:
```
# Initialize models
models = {
    'Logistic Regression': LogisticRegression(random_state=1981),
    'Random Forest': RandomForestClassifier(n_jobs=-1, random_state=1981),  # Use parallel processing
    'LightGBM': LGBMClassifier(n_jobs=-1, random_state=1981),
    'XGBoost': XGBClassifier(n_jobs=-1, random_state=1981, use_label_encoder=False, eval_metric='logloss'),
    'Balanced Bagging': BalancedBaggingClassifier(random_state=1981),
    'Easy Ensemble': EasyEnsembleClassifier(random_state=1981),
    'Balanced Random Forest': BalancedRandomForestClassifier(random_state=1981),
    'Balanced Bagging (LightGBM)': BalancedBaggingClassifier(estimator=LGBMClassifier(n_jobs=-1, random_state=1981), random_state=1981),
    'Easy Ensemble (LightGBM)': EasyEnsembleClassifier(estimator=LGBMClassifier(n_jobs=-1, random_state=1981), random_state=1981)
}
```

Class-specific level Metrics Comparison

Contents

[16]:
```
# Create a DataFrame to store metrics
metrics_df = pd.DataFrame()

# Train and evaluate models
for name, model in models.items():
    if name == 'Logistic Regression':
        model.fit(X_train_scaled, y_train)
        y_pred = model.predict(X_test_scaled)
        y_pred_proba = model.predict_proba(X_test_scaled)[:, 1]
    else:
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
        y_pred_proba = model.predict_proba(X_test)[:, 1]

    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average=None)
    recall = recall_score(y_test, y_pred, average=None)
    f1 = f1_score(y_test, y_pred, average=None)
    roc_auc = roc_auc_score(y_test, y_pred_proba)

    # Append metrics to DataFrame using pd.concat
    model_metrics = pd.DataFrame({
        'Model': [name] * len(precision),
        'Class': list(range(len(precision))),
        'Accuracy': [accuracy] * len(precision),
        'Precision': precision,
        'Recall': recall,
        'F1 Score': f1,
        'ROC AUC': [roc_auc] * len(precision)
    })
    metrics_df = pd.concat([metrics_df, model_metrics], ignore_index=True)

metrics_df
```

[16]:

	Model	Class	Accuracy	Precision	Recall	F1 Score	ROC AUC
0	Logistic Regression	0	0.942826	0.952530	0.988374	0.970121	0.881984
1	Logistic Regression	1	0.942826	0.573123	0.240642	0.338961	0.881984
2	Random Forest	0	0.939535	0.951774	0.985551	0.968368	0.841807
3	Random Forest	1	0.939535	0.508143	0.230131	0.316791	0.841807
4	LightGBM	0	0.943624	0.952778	0.988984	0.970543	0.884863
5	LightGBM	1	0.943624	0.589938	0.244330	0.345547	0.884863
6	XGBoost	0	0.942546	0.952736	0.987824	0.969962	0.881696
7	XGBoost	1	0.942546	0.565700	0.244514	0.341445	0.881696
8	Balanced Bagging	0	0.839273	0.975463	0.850234	0.908554	0.847909
9	Balanced Bagging	1	0.839273	0.224994	0.670293	0.336902	0.847909
10	Easy Ensemble	0	0.815045	0.982147	0.817915	0.892539	0.880714
11	Easy Ensemble	1	0.815045	0.215431	0.770791	0.336744	0.880714
12	Balanced Random Forest	0	0.781999	0.983462	0.780991	0.870610	0.872684
13	Balanced Random Forest	1	0.781999	0.191076	0.797529	0.308290	0.872684
14	Balanced Bagging (LightGBM)	0	0.799746	0.984274	0.799531	0.882336	0.885894
15	Balanced Bagging (LightGBM)	1	0.799746	0.206251	0.803061	0.328209	0.885894
16	Easy Ensemble (LightGBM)	0	0.793546	0.984505	0.792630	0.878209	0.885778
17	Easy Ensemble (LightGBM)	1	0.793546	0.201685	0.807671	0.322771	0.885778

Class-specific level Metrics Summary

Contents

- **High Recall, Low Precision and F1 Score:** The majority of models show poor recall for class 1 (patients with heart disease), except for Balanced Bagging, Easy Ensemble, Balanced Random Forest, and when these models are combined with LightGBM. This indicates that most models struggle to identify positive cases (patients with heart disease), resulting in a significant number of false negatives (patients incorrectly identified as not having heart disease).
- **Balanced Bagging and Easy Ensemble:**
  - Balanced Bagging and Easy Ensemble models, along with Balanced Random Forest, are designed to handle class imbalance by balancing the classes during training.
  - Performance:
    - They achieve higher recall for class 1, meaning they capture most of the actual positive cases.
    - The trade-off is typically lower precision, leading to a lower F1 score.
- **Medical Context Implication:** In a medical context, high recall is crucial as it is important to identify as many true positive cases as possible, even at the cost of some false positives. Missing a true positive (false negative) could be more critical than having a false positive.
- **Using LightGBM as Base Estimator:**
  - Performance with LightGBM:
    - When using LightGBM as the base estimator in Balanced Bagging and Easy Ensemble, the results show improved recall for class 1.
    - These models also have slightly better ROC AUC scores (0.885894 and 0.885778, respectively), indicating a good balance between sensitivity and specificity.
    - LightGBM is a powerful gradient boosting framework known for its efficiency and performance, which helps in achieving better overall performance metrics.
  - Improvement:
    - When using Easy Ensemble as a wrapper and LightGBM as a base estimator, the Recall for class 1 (heart disease patients) improves significantly from 24.4% (in standalone LightGBM) to 80.7%.
    - The ROC AUC improves from 88.4% to 88.6% for class 1, showing a better balance between correctly identifying true positives and minimizing false positives.
- **Practical Implications:** Heart Disease Classification Task:
  - Identifying patients with heart disease (true positives) is critical.
  - High recall is generally more desirable, even at the cost of having more false positives.
  - High Recall ensures most patients with heart disease are identified, which is crucial for early intervention and treatment.
  - False Positives, while not ideal, can be managed through follow-up testing and further medical evaluation.
- **Conclusion:** Balanced Bagging, Easy Ensemble, and Balanced Random Forest models, particularly with LightGBM as the base estimator, provide a good balance between identifying true positives and maintaining a reasonable rate of false positives. For a medical application such as heart disease prediction, these approaches ensure that most cases of heart disease are identified, enabling timely medical intervention, which is crucial for patient care.

Model Selection

Contents

- Based on above metrics, the **Easy Ensemble (LightGBM) model** is the best choise for the being the final model:
  - Accuracy: `0.793546`
  - Precision: `0.201685`
  - Recall: `0.807671`
  - F1 Score: `0.322771`
  - ROC AUC: `0.885778`
- **Why Easy Ensemble (LightGBM)?**
  - High Recall `(0.807671)` : This ensures we capture the majority of the true positive cases (patients with heart disease).
  - Good ROC AUC `(0.885778)` : Indicates good overall model performance in distinguishing between classes.
  - Balanced Handling of Class Imbalance: Easy Ensemble effectively manages class imbalance, which is often a challenge in medical datasets.

Hyperparameter Tuning using Optuna

Contents

[17]:
```
import time
start = time.time()
def objective(trial):
    # Define the parameter search space:
    params = {
        'n_estimators': trial.suggest_categorical('n_estimators', [10, 50, 100, 500, 1000, 5000]),
        'learning_rate': trial.suggest_categorical('learning_rate', [0.001, 0.01, 0.05, 0.1, 0.3, 0.5, 0.7, 1]),
        'boosting_type': trial.suggest_categorical('boosting_type', ['gbdt', 'dart', 'goss']),
        'num_leaves': trial.suggest_int('num_leaves', 2, 256),
        'max_depth': trial.suggest_int('max_depth', 3, 30),
        'min_child_samples': trial.suggest_int('min_child_samples', 1, 100),
        'subsample': trial.suggest_float('subsample', 0.1, 1.0),
        'colsample_bytree': trial.suggest_categorical('colsample_bytree', [0.6, 0.7, 0.8, 0.9]),
        'reg_alpha': trial.suggest_categorical('reg_alpha', [0, 0.2, 0.4, 0.6, 0.8, 1]),
        'reg_lambda': trial.suggest_categorical('reg_lambda', [0, 0.2, 0.4, 0.6, 0.8, 1])
    }

    # Initialize the model with trial parameters:
    model = EasyEnsembleClassifier(
                                   estimator=LGBMClassifier(**params, n_jobs=-1, random_state=1981),
                                   random_state=1981
    )

    # Perform cross-validation and return the recall score:
    score = cross_val_score(model, X_train, y_train, scoring='recall', cv=3).mean()
    return score

# Create a study and optimize:
study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=5)
end = time.time()
duration = end-start
print('\nParameters Optimization took %0.2f seconds (%0.1f minutes)'%(duration, duration/60))
# Print the best hyperparameters:
print("\nBest hyperparameters:\n ", study.best_params)
```

```
[I 2024-06-22 14:21:08,697] A new study created in memory with name: no-name-47b1e9ab-8e71-4d35-89d4-a0eb934d423f
[I 2024-06-22 14:21:35,465] Trial 0 finished with value: 0.8134530826082887 and parameters: {'n_estimators': 10, 'learning_rate': 0.1, 'boosting_type':
'gbdt', 'num_leaves': 104, 'max_depth': 10, 'min_child_samples': 24, 'subsample': 0.8437808863271848, 'colsample_bytree': 0.8, 'reg_alpha': 0, 'reg_lamb
da': 0.6}. Best is trial 0 with value: 0.8134530826082887.
[I 2024-06-22 14:33:21,173] Trial 1 finished with value: 0.8133595284872298 and parameters: {'n_estimators': 1000, 'learning_rate': 0.001, 'boosting_typ
e': 'goss', 'num_leaves': 214, 'max_depth': 22, 'min_child_samples': 94, 'subsample': 0.5513388475010556, 'colsample_bytree': 0.6, 'reg_alpha': 0.2, 're
g_lambda': 0}. Best is trial 0 with value: 0.8134530826082887.
[I 2024-06-22 14:34:05,403] Trial 2 finished with value: 0.808026943586865 and parameters: {'n_estimators': 50, 'learning_rate': 0.05, 'boosting_type':
'dart', 'num_leaves': 174, 'max_depth': 22, 'min_child_samples': 28, 'subsample': 0.7781802141796027, 'colsample_bytree': 0.9, 'reg_alpha': 0, 'reg_lamb
da': 0}. Best is trial 0 with value: 0.8134530826082887.
[I 2024-06-22 14:34:14,704] Trial 3 finished with value: 0.8085882683132192 and parameters: {'n_estimators': 10, 'learning_rate': 0.7, 'boosting_type':
'gbdt', 'num_leaves': 6, 'max_depth': 4, 'min_child_samples': 69, 'subsample': 0.3324602203886745, 'colsample_bytree': 0.8, 'reg_alpha': 0, 'reg_lambd
a': 0.4}. Best is trial 0 with value: 0.8134530826082887.
[I 2024-06-22 14:35:37,205] Trial 4 finished with value: 0.8098044718869866 and parameters: {'n_estimators': 100, 'learning_rate': 0.1, 'boosting_type':
'dart', 'num_leaves': 105, 'max_depth': 19, 'min_child_samples': 42, 'subsample': 0.4600041876904618, 'colsample_bytree': 0.8, 'reg_alpha': 0, 'reg_lamb
da': 0.6}. Best is trial 0 with value: 0.8134530826082887.

Parameters Optimization took 868.51 seconds (14.5 minutes)

Best hyperparameters:
  {'n_estimators': 10, 'learning_rate': 0.1, 'boosting_type': 'gbdt', 'num_leaves': 104, 'max_depth': 10, 'min_child_samples': 24, 'subsample': 0.843780
8863271848, 'colsample_bytree': 0.8, 'reg_alpha': 0, 'reg_lambda': 0.6}
```

Fitting Best Model (Tuned)

Contents

[21]:
```
best_params = study.best_params

# Initialize the model with the best parameters:
best_model = EasyEnsembleClassifier(
    estimator=LGBMClassifier(**best_params, n_jobs=-1, random_state=1981),
    random_state=1981
)

# Fit the model:
best_model.fit(X_train, y_train)
y_pred = best_model.predict(X_test)
y_pred_proba = best_model.predict_proba(X_test)[:, 1]

accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average=None)
recall = recall_score(y_test, y_pred, average=None)
f1 = f1_score(y_test, y_pred, average=None)
roc_auc = roc_auc_score(y_test, y_pred_proba)

# Create a DataFrame to store metrics
metrics_df = pd.DataFrame()

# Append metrics to DataFrame using pd.concat
model_metrics = pd.DataFrame({
    'Model': ['Easy Ensemble (LightGBM)'] * len(precision),
    'Class': list(range(len(precision))),
    'Accuracy': [accuracy] * len(precision),
    'Precision': precision,
    'Recall': recall,
    'F1 Score': f1,
    'ROC AUC': [roc_auc] * len(precision)
})
metrics_df = pd.concat([metrics_df, model_metrics], ignore_index=True)
metrics_df
```

[21]:

	Model	Class	Accuracy	Precision	Recall	F1 Score	ROC AUC
0	Easy Ensemble (LightGBM)	0	0.786862	0.984788	0.785166	0.873720	0.883942
1	Easy Ensemble (LightGBM)	1	0.786862	0.197094	0.813019	0.317274	0.883942

Tuned Best Model Metrics Summary:

- Class 0:
  - The model has a high precision `(0.984788)` for class 0, indicating that when it predicts class 0, it is correct `98.48%` of the time.
  - The recall for class 0 is also reasonably high `(0.785166)`, meaning it correctly identifies `78.52%` of all actual class 0 instances.
  - The F1 score `(0.873720)` shows a good balance between precision and recall.
  - The ROC AUC for class 0 is `0.785166`, indicating good discriminative ability.
- Class 1:
  - The precision for class 1 is low `(0.197094)`, meaning many of the predicted class 1 instances are actually class 0.
  - However, the recall for class 1 is high `(0.813019)`, indicating the model is good at identifying actual class 1 instances.
  - The F1 score for class 1 is relatively low `(0.317274)`, suggesting a trade-off between precision and recall.
  - The ROC AUC for class 1 is the same as for class 0 `(0.883942)`, indicating overall good model performance in distinguishing between classes.

Comparison to Separate and Combined Models:

- LightGBM Alone:
  - LightGBM typically has strong performance due to its gradient boosting capabilities. It may achieve high accuracy and good precision/recall balances for both classes.
  - However, LightGBM alone might struggle with class imbalance, often resulting in lower recall for minority classes (class 1: Recall `24.4%`).
- EasyEnsemble Alone:
  - EasyEnsemble without LightGBM as the base estimator focuses on balancing the data using under-sampling and creating multiple models.
  - This approach improves recall for minority classes but might not achieve the high precision that LightGBM offers.
  - The combined approach of using EasyEnsemble with LightGBM leverages the strengths of both techniques, enhancing both precision and recall, particularly for the minority class.
- Combined (Tuned):
  - When tuned, EasyEnsemble with LightGBM as the base estimator provides a balanced approach to handle class imbalance.
  - The combined method shows improved recall for class 1 `from` 24.4% to 81.3% `while maintaining a good precision for class 0` (0.984788)`.
  - This combination also ensures that the model has a robust overall performance as indicated by the ROC AUC of `0.883942`.

Conclusion:

Using EasyEnsemble with LightGBM as the base estimator, especially when hyperparameters are tuned, offers a comprehensive solution to handling class imbalance. It ensures high precision and recall for class 0 and significantly improves recall for class 1, although precision for class 1 remains a challenge. This combined approach outperforms using LightGBM or EasyEnsemble separately by effectively leveraging the strengths of both methods.

Tuned Best Model Features Importance Using SHAP

Contents

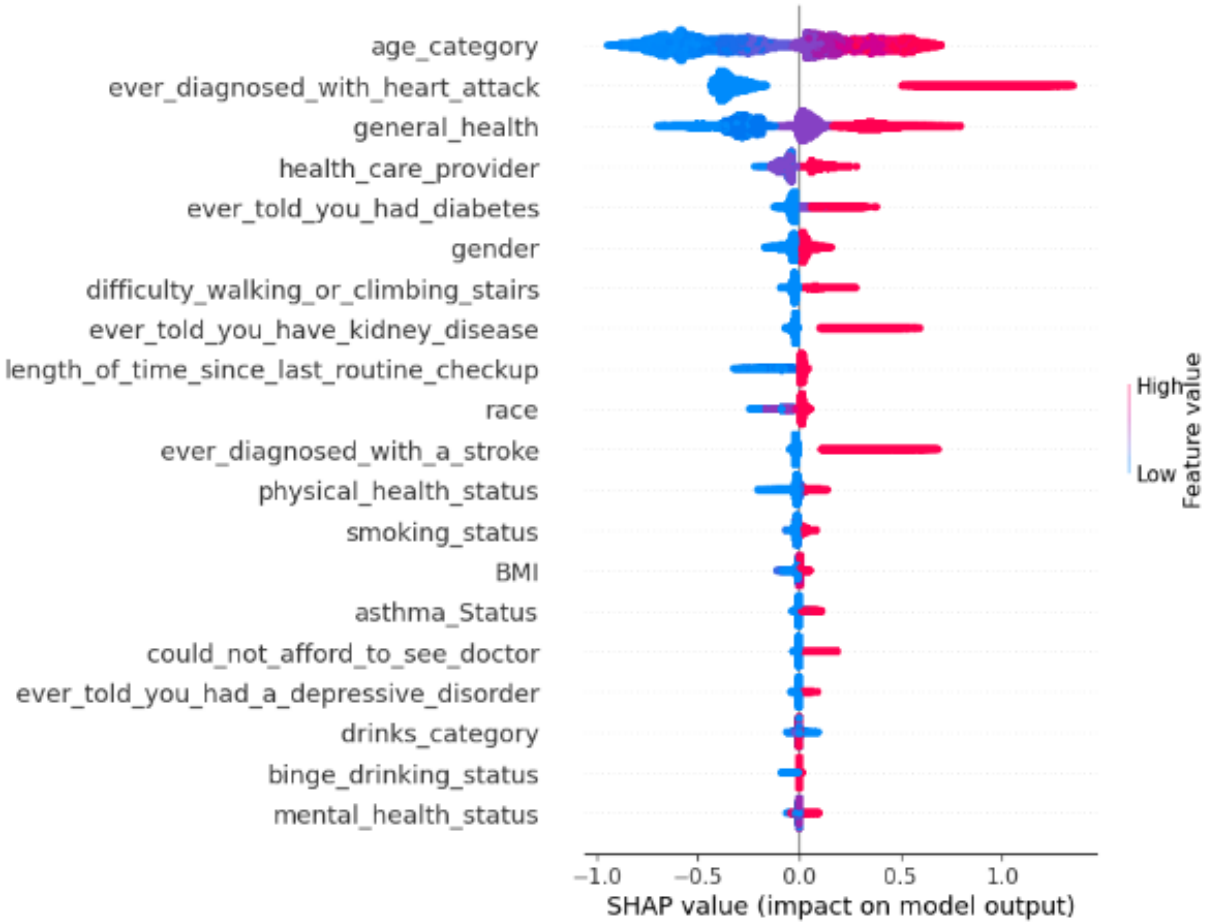
[22]:
```
# Access one of the base estimators for SHAP analysis
lgbm_model = best_model.estimators_[0].steps[-1][1]

# Create a SHAP explainer for the LGBMClassifier
explainer = shap.TreeExplainer(lgbm_model)

# Calculate SHAP values for the test set
shap_values = explainer.shap_values(X_test)

# Plot summary plot for feature importance, heart disease (class 1):
shap.summary_plot(shap_values[1], X_test, plot_size=(6,8), show=False)
```

```
LightGBM binary classifier with TreeExplainer shap values output has changed to a list of ndarray
No data for colormapping provided via 'c'. Parameters 'vmin', 'vmax' will be ignored
```



Summary of SHAP Summary Plot for Class 1 (Heart Disease Patients):

Above SHAP summary plot shows the impact of each feature on the model's output for predicting heart disease (class 1). Each dot represents a SHAP value for a feature, with the color indicating the feature's value (red for high and blue for low). Here's a detailed interpretation:

- **High Positive Impact:**
  - Age Category: Higher age (represented by red dots) increases the likelihood of heart disease. Age is a significant risk factor, with older individuals being more prone to heart disease.
  - Ever Diagnosed with Heart Attack: A history of heart attacks (red dots) greatly increases the likelihood of heart disease. This past medical history is a strong indicator of recurring or persistent heart issues.
- **Moderate Positive Impact:**
  - General Health: Poor general health (red dots) increases the likelihood of heart disease. Individuals with overall poor health are at higher risk.
  - Health Care Provider: Frequent visits to a healthcare provider (red dots) indicate a higher likelihood of heart disease, possibly due to ongoing health issues necessitating regular check-ups.
  - Ever Told You Had Diabetes: Being told by a healthcare provider that you have diabetes (red dots) increases the likelihood of heart disease. Diabetes is a well-known risk factor for cardiovascular diseases.
  - Gender: Certain gender-related factors (likely male, indicated by red dots) increase the likelihood of heart disease. Men generally have a higher risk of heart disease at a younger age compared to women.
  - Difficulty Walking or Climbing Stairs: Difficulty in these activities (red dots) indicates a higher risk of heart disease, possibly due to underlying cardiovascular issues.
  - Ever Told You Have Kidney Disease: A history of kidney disease (red dots) increases the likelihood of heart disease. Kidney disease can be associated with cardiovascular complications.
  - Length of Time Since Last Routine Checkup: A longer time since the last checkup (red dots) increases the likelihood of heart disease. Regular check-ups can help manage and prevent health issues.
  - Race: Certain racial factors (red dots) increase the likelihood of heart disease, highlighting the role of demographic and genetic factors.
  - Ever Diagnosed with a Stroke: A history of stroke (red dots) increases the likelihood of heart disease, as both share common risk factors.
  - Physical Health Status: Poor physical health (red dots) increases the likelihood of heart disease, reflecting the impact of overall physical well-being on heart health.
  - Smoking Status: Being a smoker (red dots) significantly increases the likelihood of heart disease. Smoking is a major risk factor for cardiovascular diseases.
  - BMI: Higher BMI (red dots) increases the likelihood of heart disease. Obesity is closely linked to cardiovascular risk.
  - Asthma Status: Higher severity of asthma (red dots) increases the risk, possibly due to the overall impact of chronic respiratory conditions on health.
  - Could Not Afford to See Doctor: Financial barriers to healthcare (red dots) increase the likelihood of heart disease, likely due to untreated health conditions.
  - Ever Told You Had a Depressive Disorder: A history of depressive disorder (red dots) increases the likelihood of heart disease, indicating a connection between mental and cardiovascular health.
  - Drinks Category: Higher alcohol consumption (red dots) is associated with an increased risk of heart disease. Excessive drinking can negatively impact heart health.
  - Binge Drinking Status: Higher frequency of binge drinking (red dots) increases the likelihood of heart disease, highlighting the adverse effects of excessive alcohol intake.
  - Mental Health Status: Poor mental health (red dots) increases the likelihood of heart disease, emphasizing the importance of mental well-being for heart health.
- **Mixed Impact:**
  - Race: Certain racial factors have a mixed influence but can increase the likelihood of heart disease, showing the importance of considering demographic variables in health risk assessments.
  - Asthma Status: While generally having a moderate positive impact, higher severity of asthma (red dots) increases the risk of heart disease. This indicates that severe respiratory issues can contribute to cardiovascular risk.
- **Conclusion:** The SHAP summary plot illustrates that various factors such as age, history of heart attacks, general health, and diabetes significantly impact the likelihood of heart disease. The analysis emphasizes the importance of regular health check-ups, managing chronic conditions, and addressing both physical and mental health to mitigate the risk of heart disease.

Saving Final Model

Contents

Here, we'll be saving our best model: Easy Ensemble (LightGBM) and CATBoost models.

[23]:
```
import pickle
import gzip, pickletools
```

[24]:
```
# Pickling the best model:

pickle_out = open("best_model.pkl", "wb")
pickle.dump(best_model, pickle_out)
pickle_out.close()
```

[25]:
```
pickle_out = open("cbe_encoder.pkl", "wb")
pickle.dump(cbe_encoder, pickle_out)
pickle_out.close()
```

AI-Powered Heart Disease Risk Assessment App Code:

```
import streamlit as st
import pandas as pd
import numpy as np
import pickle as pkl
from PIL import Image
import io
from lightgbm import LGBMClassifier
import category_encoders as ce
from imblearn.ensemble import EasyEnsembleClassifier


# Load the pickled model and encoder
with open('best_model.pkl', 'rb') as model_file:
    model = pkl.load(model_file)

with open('cbe_encoder.pkl', 'rb') as encoder_file:
    encoder = pkl.load(encoder_file)

# Load the dataset for reference
#data = pd.read_csv('brfss2022_data_wrangling_output.csv')
data = pd.read_csv('brfss2022_data_wrangling_output.zip', compression='zip')

# Convert the target variable to numerical values
data['heart_disease'] = data['heart_disease'].apply(lambda x: 1 if x ==
'yes' else 0).astype('int')

icon = Image.open("heart_disease.jpg")

# Now, let's change page config by changing layout to 'Wide, give the app a
new name and then change the icon:
st.set_page_config(layout='wide', page_title='AI-Powered Heart Disease
Assessment', page_icon=icon)

row0_0, row0_1, row0_2, row0_3= st.columns((0.08,6,3,0.17))

with row0_1:
    st.title("AI-Powered Heart Disease Assessment App")
    st.write(
    """
    Unmatched Accuracy with Cutting-Edge Machine Learning Models
    """
```

```
    )
st.write('---')

row0_a0, row0_b1, row0_d3= st.columns((0.08,6,0.17))
with row0_b1:
    st.write(
    """
      ## **Introduction**
    """)

with row0_b1:
    st.write(
    """
    The AI-Powered Heart Disease Risk Assessment App provides users with tailored risk scores and actionable recommendations to help mitigate their heart disease risk. Using advanced AI and modeling techniques,
    this app offers easy-to-understand assessments and preventive measures to make safeguarding your cardiovascular health straightforward and accessible.

    """)
st.write('---')

row1_a0, row2_b1, row3_d3= st.columns((0.08,6,0.17))
with row2_b1:
    st.write(
    """
      ## **How It Works:**
    """)

with row2_b1:
    st.write(
    """
    * **User Input:** Enter your health information, such as age, BMI, physical activity levels, smoking status, and medical history (e.g., heart attacks, strokes, diabetes).
    * **Data Analysis:** The app analyzes your input using advanced AI models specifically designed for heart disease risk prediction.
    * **Risk Assessment:** Receive a personalized risk score indicating your potential for heart disease.
    * **Recommendations:** Get actionable advice to mitigate your risk, including lifestyle modification suggestions.

    """)
```

```
row1_0, row1_1, row1_2, row1_3, row1_5= st.columns((0.08,3,3,3,0.17))
with row1_1:
    st.write(
    """
    #### **Demographics**
    """
    )
row2_0, row2_1, row2_2, row2_3, row2_5= st.columns((0.08,3,3,3,0.17))


gender = row2_1.selectbox("What is your gender?", ["female", "male",
"nonbinary"], index=1)
race = row2_2.selectbox("What is your race/ethnicity?", [
    "white_only_non_hispanic", "black_only_non_hispanic",
"asian_only_non_hispanic",
    "american_indian_or_alaskan_native_only_non_hispanic",
"multiracial_non_hispanic",
    "hispanic",
"native_hawaiian_or_other_pacific_islander_only_non_hispanic"
], index=0)
age_category = row2_3.selectbox("What is your age group?", [
        "Age_18_to_24", "Age_25_to_29", "Age_30_to_34","Age_35_to_39",
        "Age_40_to_44", "Age_45_to_49", "Age_50_to_54", "Age_55_to_59",
        "Age_60_to_64", "Age_65_to_69","Age_70_to_74","Age_75_to_79",
        "Age_80_or_older"
               ], index=12)

row3_0, row3_1, row3_2, row3_3, row3_5= st.columns((0.08,3,3,3,0.17))
with row3_1:
    st.write(
    """
    #### **Medical History**
    """
    )
row4_0, row4_1, row4_2, row4_3, row4_5= st.columns((0.08,3,3,3,0.17))

# row4_1
general_health = row4_1.selectbox("How would you rate your overall health?",
["excellent", "very_good", "good", "fair", "poor"], index=4)
heart_attack = row4_1.selectbox("Have you ever been diagnosed with a heart
attack?", ["yes", "no"], help="A heart attack occurs when blood flow to part
of the heart is blocked!", index=0)
kidney_disease = row4_1.selectbox("Has a doctor ever told you that you have
kidney disease?", ["yes", "no"], index=0)
```

```
asthma = row4_1.selectbox("Have you ever been diagnosed with asthma?",
["never_asthma", "current_asthma", "former_asthma"], index=1)
could_not_afford_to_see_doctor = row4_1.selectbox("Have you ever been unable
to see a doctor when needed due to cost?", ["yes", "no"], index=0)
#row4_2
health_care_provider = row4_2.selectbox("Do you have a primary health care
provider?", ["yes_only_one", "more_than_one", "no"], index=2)
stroke = row4_2.selectbox("Have you ever been diagnosed with a stroke?",
["yes", "no"], help="A stroke happens when blood supply to part of the brain
is interrupted!", index=0)
diabetes = row4_2.selectbox("Have you ever been diagnosed with diabetes?",
["yes", "no", "no_prediabetes", "yes_during_pregnancy"], key="diabetes",
index=0)
bmi = row4_2.selectbox("What is your body mass index (BMI)?", [
        "underweight_bmi_less_than_18_5", "normal_weight_bmi_18_5_to_24_9",
"overweight_bmi_25_to_29_9",
        "obese_bmi_30_or_more"

], help="BMI is a measure of body fat based on height and weight. Please use
the BMI calculator at
https://www.nhlbi.nih.gov/health/educational/lose_wt/BMI/bmicalc.htm",
index=3)
length_of_time_since_last_routine_checkup = row4_2.selectbox("How long has
it been since your last routine checkup?", ["past_year", "past_2_years",
"past_5_years", "5+_years_ago", "never"], index=4)
#row4_3
depressive_disorder = row4_3.selectbox("Has a doctor ever told you that you
have a depressive disorder?", ["yes", "no"], help="A depressive disorder is
a medical condition characterized by persistent feelings of sadness, loss of
interest, and other emotional and physical symptoms!", index=0)
physical_health = row4_3.selectbox("How many days in the past 30 days was
your physical health not good?", ["zero_days_not_good",
"1_to_13_days_not_good", "14_plus_days_not_good"], key="physical_health",
index=2)
mental_health = row4_3.selectbox("How many days in the past 30 days was your
mental health not good?", ["zero_days_not_good", "1_to_13_days_not_good",
"14_plus_days_not_good"], key="mental_health", index=2)
walking = row4_3.selectbox("Do you have difficulty walking or climbing
stairs?", ["yes", "no"], key="walking", index=0)

row5_0, row5_1, row5_2, row5_3, row5_5= st.columns((0.08,3,3,3,0.17))
with row5_1:
    st.write(
    """
    #### **Lifestyle**
```

```
    """
    )

row6_0, row6_1, row6_2, row6_3, row6_5= st.columns((0.08,3,3,3,0.17))
smoking_status = row6_1.selectbox("What is your smoking status?", [
    "never_smoked", "former_smoker", "current_smoker_some_days",
"current_smoker_every_day"], key="smoking_status", index=3)
sleep_category = row6_1.selectbox("How many hours of sleep do you get on a
typical night?", [
    "very_short_sleep_0_to_3_hours", "short_sleep_4_to_5_hours",
"normal_sleep_6_to_8_hours",
    "long_sleep_9_to_10_hours", "very_long_sleep_11_or_more_hours"],
key="sleep_category", index=0)
drinks_category = row6_2.selectbox("How many alcoholic drinks do you consume
in a typical week?", [
    "did_not_drink", "very_low_consumption_0.01_to_1_drinks",
"low_consumption_1.01_to_5_drinks",
    "moderate_consumption_5.01_to_10_drinks",
"high_consumption_10.01_to_20_drinks",
"very_high_consumption_more_than_20_drinks"], key="drinks_category",
index=5)
binge_drinking_status = row6_2.selectbox("Have you engaged in binge drinking
in the past 30 days?", ["yes", "no"], help="Binge drinking is consuming 5 or
more drinks for men, or 4 or more drinks for women, in about 2 hours!",
index=0)
exercise_status = row6_3.selectbox("Have you exercised in the past 30
days?", ["yes", "no"], key="exercise_status", index=0)
with row6_1:
    st.write(
    """
    #### **Learn More**
    [![](https://img.shields.io/badge/GitHub%20-Features%20Information-
informational)](https://github.com/akthammomani/AI_powered_heart_disease_ris
k_assessment_app/tree/main/Notebooks/Exploratory_Data_Analysis/)
    """)

# Collect input data
input_data = {
    'gender': gender,
    'race': race,
    'general_health': general_health,
    'health_care_provider': health_care_provider,
    'could_not_afford_to_see_doctor': could_not_afford_to_see_doctor,
    'length_of_time_since_last_routine_checkup':
length_of_time_since_last_routine_checkup,
```

```
    'ever_diagnosed_with_heart_attack': heart_attack,
    'ever_diagnosed_with_a_stroke': stroke,
    'ever_told_you_had_a_depressive_disorder': depressive_disorder,
    'ever_told_you_have_kidney_disease': kidney_disease,
    'ever_told_you_had_diabetes': diabetes,
    'BMI': bmi,
    'difficulty_walking_or_climbing_stairs': walking,
    'physical_health_status': physical_health,
    'mental_health_status': mental_health,
    'asthma_Status': asthma,
    'smoking_status': smoking_status,
    'binge_drinking_status': binge_drinking_status,
    'exercise_status_in_past_30_Days': exercise_status,
    'age_category': age_category,
    'sleep_category': sleep_category,
    'drinks_category': drinks_category
}

# Define a function to make predictions
def predict_heart_disease_risk(input_data, model, encoder):
    input_df = pd.DataFrame([input_data])
    input_encoded = encoder.transform(input_df, y=None,
override_return_df=False)
    prediction = model.predict_proba(input_encoded)[:, 1][0] * 100
    return prediction

st.write('---')
row8_0, row8_1, row8_5= st.columns((0.08,12,0.17))

with row8_1:
    st.write(
    """
    #### **AI Heart Disease Risk Assessment**
    """
    )

# This button will control the heart of the APP which is the Machine
Learning part :)
btn1 = row8_1.button('Get Your Heart disease Risk Assessment')

if btn1:
    try:

        risk = predict_heart_disease_risk(input_data, model, encoder)
        with row8_1:
```

```
            st.write(f"Predicted Heart Disease Risk: {risk:.2f}%")

            # Provide recommendations based on risk and user inputs
            recommendations = []

            if risk > 75:
                recommendations.append("Your risk of heart disease is very
high. Here are some recommendations to reduce your risk:")
            elif risk > 50:
                recommendations.append("Your risk of heart disease is high.
Here are some recommendations to reduce your risk:")
            elif risk > 25:
                recommendations.append("Your risk of heart disease is
moderate. Here are some recommendations to reduce your risk:")
            else:
                recommendations.append("Your risk of heart disease is low.
Keep up the good work and continue to maintain a healthy lifestyle.")

            if heart_attack == "yes":
                recommendations.append("- Regularly visit your cardiologist
and adhere to prescribed medications.")
                recommendations.append("- Monitor any new or worsening
symptoms and seek immediate medical attention if needed.")
            if stroke == "yes":
                recommendations.append("- Follow your neurologist's
recommendations and take prescribed medications consistently.")
                recommendations.append("- Engage in approved physical
therapy or exercises to regain strength and mobility.")
            if general_health in ["fair", "poor"]:
                recommendations.append("- Focus on improving your overall
health through a balanced diet and regular check-ups.")
            if bmi in ["overweight_bmi_25_to_29_9", "obese_bmi_30_or_more"]:
                recommendations.append("- Maintain a healthy weight through
diet and exercise.")
            if smoking_status != "never_smoked":
                recommendations.append("- Quit smoking to significantly
reduce your risk of heart disease.")
            if exercise_status == "no":
                recommendations.append("- Engage in regular physical
activity to improve your heart health.")
            if binge_drinking_status == "yes" or drinks_category in
["high_consumption_10.01_to_20_drinks",
"very_high_consumption_more_than_20_drinks"]:
                recommendations.append("- Limit alcohol consumption to lower
your risk.")
```

```
            if sleep_category in ["short_sleep_4_to_5_hours",
"very_short_sleep_0_to_3_hours"]:
                recommendations.append("- Consider aiming for 7-9 hours of
quality sleep each night. Adequate sleep is crucial for maintaining heart
health.")
            for recommendation in recommendations:
                st.write(recommendation)
    except Exception as e:
        row8_1.error(e)

with row8_1:
    st.write(
    """
    #### **Learn More**
    [![](https://img.shields.io/badge/GitHub%20-Machine%20Learning%20Models-
informational)](https://github.com/akthammomani/AI_powered_heart_disease_ris
k_assessment_app/tree/main/Notebooks/Modeling/)
    """)

with row8_1:
    st.write(
    """
    ###### ***Disclaimer***
    *This app is not a replacement for professional medical advice,
diagnosis, or treatment. Always consult your doctor or a qualified
healthcare provider with any questions you may have regarding your health.*
    """)
st.write('---')
null9_0,row9_1 ,row9_2= st.columns((0.02,5,0.05))
with row9_1.expander("Leave Us a Comment or Question"):
    contact_form = """
                    <form
action=https://formsubmit.co/aktham.momani81@gmail.com method="POST">
                        <input type="hidden" name="_captcha" value="false">
                        <input type="text" name="name" placeholder="Your
name" required>
                        <input type="email" name="email" placeholder="Your
email" required>
                        <textarea name="message" placeholder="Your message
here"></textarea>
                        <button type="submit">Send</button>
                    </form>
                    """
    st.markdown(contact_form, unsafe_allow_html=True)
```

```
    # Use Local CSS File
    def local_css(file_name):
        with open(file_name) as f:
            st.markdown(f"<style>{f.read()}</style>",
unsafe_allow_html=True)

    local_css("style.css")


null10_0,row10_1,row10_2= st.columns((0.04,7, 0.4))
with row10_1:
    st.write(
    """
    ### **Contacts**
    [![](https://img.shields.io/badge/GitHub-Follow-
informational)](https://github.com/akthammomani)
    [![](https://img.shields.io/badge/Linkedin-Connect-
informational)](https://www.linkedin.com/in/akthammomani/)
    [![](https://img.shields.io/badge/Open-Issue-
informational)](https://github.com/akthammomani/AI_powered_heart_disease_ris
k_assessment_app/issues)
    [![MAIL Badge](https://img.shields.io/badge/-aktham.momani81@gmail.com-
c14438?style=flat-
square&logo=Gmail&logoColor=white&link=mailto:aktham.momani81@gmail.com)](ma
ilto:aktham.momani81@gmail.com)

    ###### © Aktham Momani, 2024
    """
)
```