# CIND-820 Capstone Project: An ML Tool to Detect Heart Disease


- Robert M. Pineau
- 941-049-371
- Supervisor: Dr. Ceni Babaoglu


### Install Required Modules:

```
!pip install matplotlib
!pip install graphviz
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/publ
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-packages (3.2
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /usr/local/li
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: numpy>=1.11 in /usr/local/lib/python3.7/dist-packages (fr
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dist-packages (f
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-packages (from
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/publ
Requirement already satisfied: graphviz in /usr/local/lib/python3.7/dist-packages (0.10
```

### Load Required Libraries:

```
import sys
from google.colab import drive
import math
from statistics import mean, stdev
import pandas as pd
import numpy as np
from scipy import stats
import plotly
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn import tree
from sklearn.naive_bayes import MultinomialNB
from sklearn.naive_bayes import GaussianNB
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
```

```
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier

from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn import metrics
from sklearn.metrics import accuracy_score
from sklearn.metrics import f1_score

from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RandomizedSearchCV, GridSearchCV

import graphviz
```

### Obtain Data-Set from Google Drive:

```
# Mounting google colab, this will prompt first time each session.
drive.mount('/content/drive',force_remount=True)
dataset_file = "/content/drive/My Drive/Colab Notebooks/heart_statlog_cleveland_hungary_fi
df=pd.read_csv(dataset_file,sep=',')
df.name = "Original Data-Set"
print(df.name)
df.head(3)
```

```
Mounted at /content/drive
Original Data-Set
```

| | age | sex | chest pain type | resting bp s | cholesterol | fasting blood sugar | resting ecg | max heart rate | exercise angina | oldpeak | sl |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 40 | 1 | 2 | 140 | 289 | 0 | 0 | 172 | 0 | 0.0 | |
| 1 | 49 | 0 | 3 | 160 | 180 | 0 | 0 | 156 | 0 | 1.0 | |

### Clean up Column Names:

```
#Rename the columns to be nicer, no spaces.
df=df.rename(columns={"age": "Age", "sex": "Sex", "chest pain type": "ChestPainType", "res
df=df.rename(columns={"cholesterol":"Cholesterol","fasting blood sugar": "FastingBloodSuga
df=df.rename(columns={"chest pain type": "ChestPainType", "resting bp s": "RestingBP_s", "
df=df.rename(columns={"resting ecg": "RestingECG", "max heart rate": "MaxHeartRate", "exer
df=df.rename(columns={"oldpeak":"OldPeak", "ST slope": "ST_Slope", "target": "Target"})
df.name = "Original Data-Set"
df.head(3)
```

| | Age | Sex | ChestPainType | RestingBP_s | Cholesterol | FastingBloodSugar | RestingECG | Ma |
|---|---|---|---|---|---|---|---|---|
| **0** | 40 | 1 | 2 | 140 | 289 | 0 | 0 | |
| **1** | 49 | 0 | 3 | 160 | 180 | 0 | 0 | |

### Datatypes and Quantities:

```
#Check data types.
print(df.dtypes)
```

```
Age                    int64
Sex                    int64
ChestPainType          int64
RestingBP_s            int64
Cholesterol            int64
FastingBloodSugar      int64
RestingECG             int64
MaxHeartRate           int64
ExerciseAngina         int64
OldPeak              float64
ST_Slope               int64
Target                 int64
dtype: object
```

```
#Datatypes, counts, etc.
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1190 entries, 0 to 1189
Data columns (total 12 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   Age                1190 non-null   int64  
 1   Sex                1190 non-null   int64  
 2   ChestPainType      1190 non-null   int64  
 3   RestingBP_s        1190 non-null   int64  
 4   Cholesterol        1190 non-null   int64  
 5   FastingBloodSugar  1190 non-null   int64  
 6   RestingECG         1190 non-null   int64  
 7   MaxHeartRate       1190 non-null   int64  
 8   ExerciseAngina     1190 non-null   int64  
 9   OldPeak            1190 non-null   float64
 10  ST_Slope           1190 non-null   int64  
 11  Target             1190 non-null   int64  
dtypes: float64(1), int64(11)
memory usage: 111.7 KB
```

### Check for Missing and NULL entries:

```
#Check for NULL or missing entries. (none)
print(df.isna().any())
print("\n\n")
print(df.isnull().any())
```

```
Age                 False
Sex                 False
ChestPainType       False
RestingBP_s         False
Cholesterol         False
FastingBloodSugar   False
RestingECG          False
MaxHeartRate        False
ExerciseAngina      False
OldPeak             False
ST_Slope            False
Target              False
dtype: bool



Age                 False
Sex                 False
ChestPainType       False
RestingBP_s         False
Cholesterol         False
FastingBloodSugar   False
RestingECG          False
MaxHeartRate        False
ExerciseAngina      False
OldPeak             False
ST_Slope            False
Target              False
dtype: bool
```

### Check for Duplicate Entries:

```
#Look for rows that are 100% identical to each other.
#
dup_count = sum(df.duplicated())
print(f"There are {dup_count} duplicate rows in this dataset.\n")

#Droping any duplicate entries.
df = df.drop_duplicates(ignore_index = True)
df.name = "Original Data-Set"

df.info()
```

```
There are 0 duplicate rows in this dataset.

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 917 entries, 0 to 916
Data columns (total 12 columns):
```

```
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Age                917 non-null    int64
 1   Sex                917 non-null    int64
 2   ChestPainType      917 non-null    int64
 3   RestingBP_s        917 non-null    int64
 4   Cholesterol        917 non-null    int64
 5   FastingBloodSugar  917 non-null    int64
 6   RestingECG         917 non-null    int64
 7   MaxHeartRate       917 non-null    int64
 8   ExerciseAngina     917 non-null    int64
 9   OldPeak            917 non-null    float64
 10  ST_Slope           917 non-null    int64
 11  Target             917 non-null    int64
dtypes: float64(1), int64(11)
memory usage: 86.1 KB
```

### Check for Out-Of-Bound Entries for Nominal and Binary attributes:

```
#Check for out of bound entries(outliers) for nominal and binary attributes (including the
#Since all nominal and binary attributes have a valid contiguous integer range, ie 0-1, or
#we only need to look for those outside the range.
valid_values = {'Sex': [0,1], 'ChestPainType': [1,2,3,4], 'FastingBloodSugar': [0,1], 'Res

for col in ('Sex', 'ChestPainType', 'FastingBloodSugar', 'RestingECG', 'ExerciseAngina', '
  valid = np.array(valid_values[col])
  max_valid = valid.max()
  min_valid = valid.min()
  print(f"For attribute '{col}': Valid MAX: {max_valid}, Valid MIN: {min_valid}")
  these_outliers = df[((df[col] < min_valid) | (df[col] > max_valid))]

  if (these_outliers.shape[0] > 1):
    print(f"For attribute '{col}': There are {these_outliers.shape[0]} outliers:\n")
    print(these_outliers)
    print("\n\n")
  elif (these_outliers.shape[0] == 1):
    print(f"For attribute '{col}': There is {these_outliers.shape[0]} outlier:\n")
    print(these_outliers)
    print("\n\n")
  else:
    print(f"For attribute '{col}': There are no outliers.\n")
```

```
For attribute 'Sex': Valid MAX: 1, Valid MIN: 0
For attribute 'Sex': There are no outliers.

For attribute 'ChestPainType': Valid MAX: 4, Valid MIN: 1
For attribute 'ChestPainType': There are no outliers.

For attribute 'FastingBloodSugar': Valid MAX: 1, Valid MIN: 0
For attribute 'FastingBloodSugar': There are no outliers.

For attribute 'RestingECG': Valid MAX: 2, Valid MIN: 0
```

```
For attribute 'RestingECG': There are no outliers.

For attribute 'ExerciseAngina': Valid MAX: 1, Valid MIN: 0
For attribute 'ExerciseAngina': There are no outliers.

For attribute 'ST_Slope': Valid MAX: 3, Valid MIN: 1
For attribute 'ST_Slope': There are no outliers.

For attribute 'Target': Valid MAX: 1, Valid MIN: 0
For attribute 'Target': There are no outliers.
```

### Remove Out-Of-Bound Entry:

```
#From above, there is a problem with one entry regarding the ST_Slope attribute, it is zer
#
#The documentation at https://ieee-dataport.org/open-access/heart-disease-dataset-comprehe
#Shows a range of 0-2, but in the definition of the mapped nominal values it shows:
#
# -- Value 1: upsloping
# -- Value 2: flat
# -- Value 3: downsloping
#
print(df['ST_Slope'].value_counts().sort_index())
print("\n")
#
#
#Since there is only one entry out of range, making the assumption that the correct range
#
#Will simply drop this entry.
df = df[df['ST_Slope'] != 0]
df.name = "Original Data-Set"

df.info()
```

```
1    395
2    459
3     63
Name: ST_Slope, dtype: int64


<class 'pandas.core.frame.DataFrame'>
Int64Index: 917 entries, 0 to 916
Data columns (total 12 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Age                917 non-null    int64
 1   Sex                917 non-null    int64
 2   ChestPainType      917 non-null    int64
 3   RestingBP_s        917 non-null    int64
 4   Cholesterol        917 non-null    int64
 5   FastingBloodSugar  917 non-null    int64
```

```
 6   RestingECG        917 non-null   int64
 7   MaxHeartRate      917 non-null   int64
 8   ExerciseAngina    917 non-null   int64
 9   OldPeak           917 non-null   float64
 10  ST_Slope          917 non-null   int64
 11  Target            917 non-null   int64
dtypes: float64(1), int64(11)
memory usage: 93.1 KB
```

### Basic Statistics for All Attributes:

```
#Basic Statistics of the dataset.(Measures of Center/Central Tendency, and Measures of Var
df.describe().T
```

| | count | mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|---|
| **Age** | 917.0 | 53.495093 | 9.425601 | 28.0 | 47.0 | 54.0 | 60.0 | 77.0 |
| **Sex** | 917.0 | 0.789531 | 0.407864 | 0.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| **ChestPainType** | 917.0 | 3.251908 | 0.931502 | 1.0 | 3.0 | 4.0 | 4.0 | 4.0 |
| **RestingBP_s** | 917.0 | 132.377317 | 18.515114 | 0.0 | 120.0 | 130.0 | 140.0 | 200.0 |
| **Cholesterol** | 917.0 | 198.803708 | 109.443764 | 0.0 | 173.0 | 223.0 | 267.0 | 603.0 |
| **FastingBloodSugar** | 917.0 | 0.232279 | 0.422517 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 |
| **RestingECG** | 917.0 | 0.604144 | 0.806161 | 0.0 | 0.0 | 0.0 | 1.0 | 2.0 |
| **MaxHeartRate** | 917.0 | 136.814613 | 25.473732 | 60.0 | 120.0 | 138.0 | 156.0 | 202.0 |
| **ExerciseAngina** | 917.0 | 0.404580 | 0.491078 | 0.0 | 0.0 | 0.0 | 1.0 | 1.0 |
| **OldPeak** | 917.0 | 0.888332 | 1.066749 | -2.6 | 0.0 | 0.6 | 1.5 | 6.2 |
| **ST_Slope** | 917.0 | 1.637950 | 0.607270 | 1.0 | 1.0 | 2.0 | 2.0 | 3.0 |
| **Target** | 917.0 | 0.552890 | 0.497466 | 0.0 | 0.0 | 1.0 | 1.0 | 1.0 |

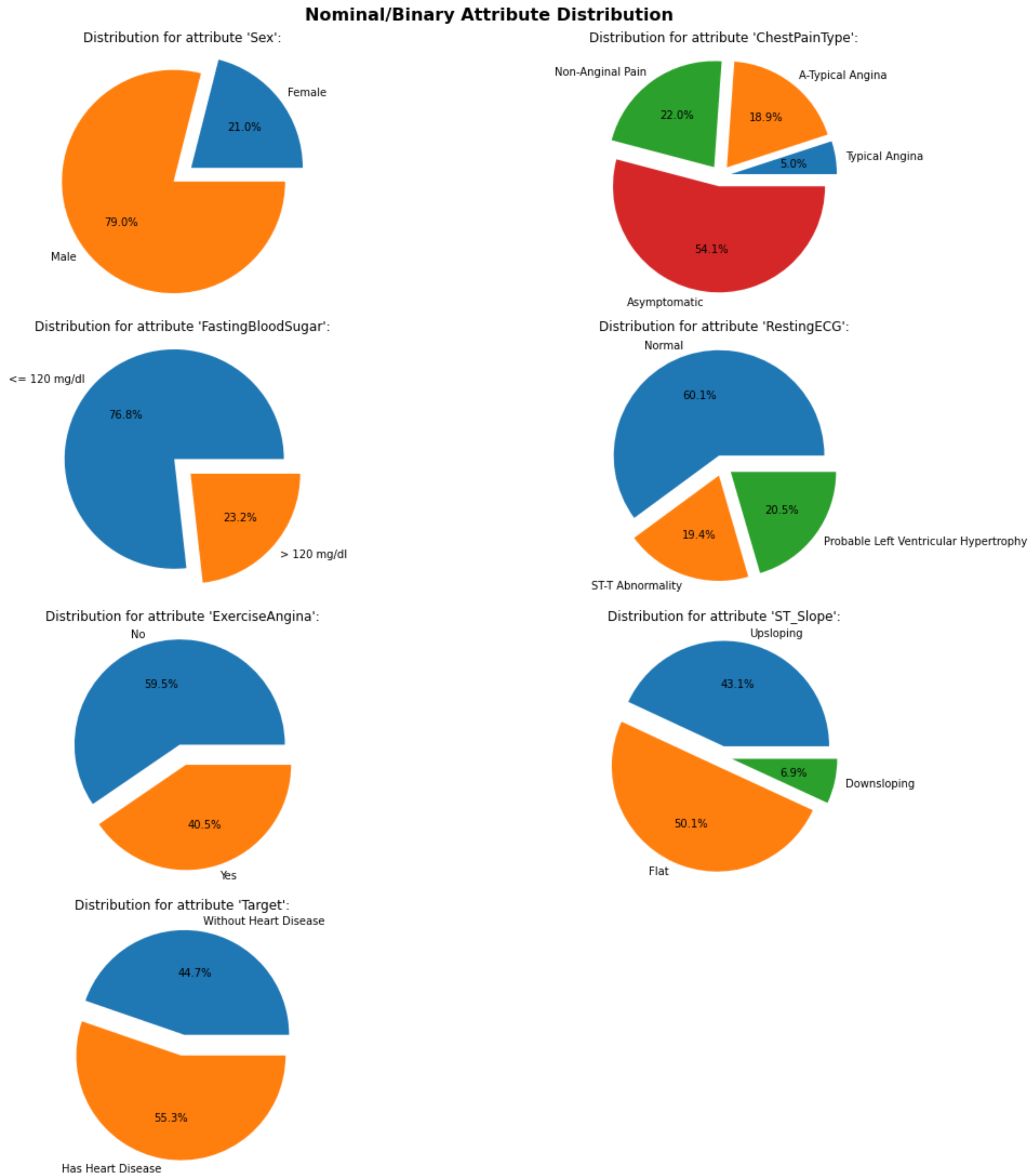
### Visualizing All Attributes:

```
#Assigning descriptive labels for all possible values for all nominal/binary attributes.
#
labels = {'Sex': ['Female', 'Male'], 'ChestPainType': ['Typical Angina', 'A-Typical Angina
          'FastingBloodSugar': ['<= 120 mg/dl', '> 120 mg/dl'], 'RestingECG': ['Normal', '
          'ExerciseAngina':['No', 'Yes'], 'ST_Slope': ['Upsloping', 'Flat', 'Downsloping']


```

```
#Creating subsets of the data for a series of interesting plots to help with visualization
```

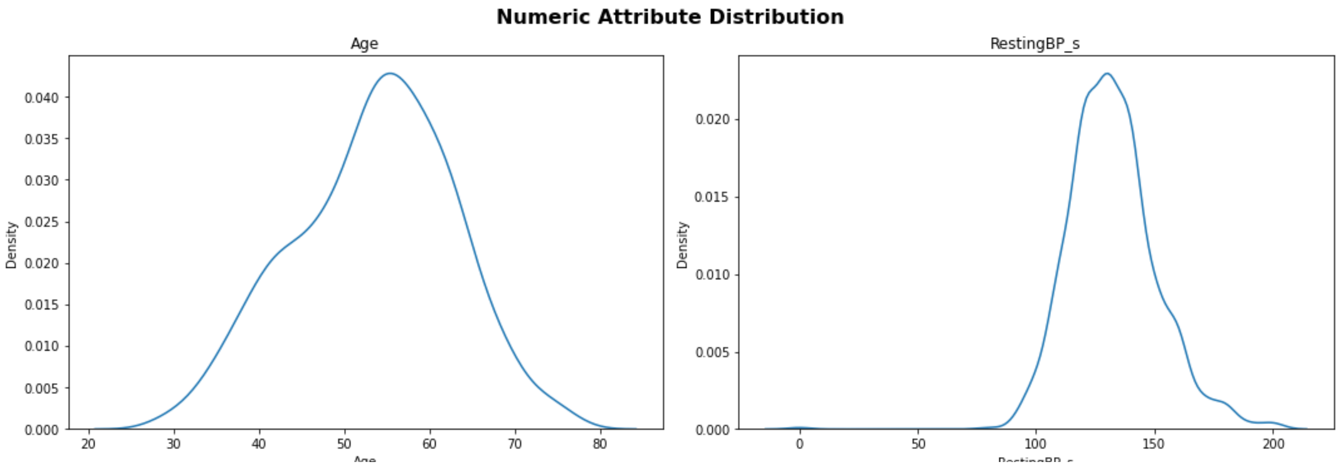
```
#
#Breaking up the dataset into two groups, those with heart disease and those without.
with_heart_disease = df[df['Target'] == 1]
no_heart_disease = df[df['Target'] == 0]

#Breaking up the dataset into four groups, by Sex, and those with heart disease and those
with_heart_disease_male = df[(df['Target'] == 1) & (df['Sex'] == 1)]
with_heart_disease_female = df[(df['Target'] == 1) & (df['Sex'] == 0)]
no_heart_disease_male = df[(df['Target'] == 0) & (df['Sex'] == 1)]
no_heart_disease_female = df[(df['Target'] == 0) & (df['Sex'] == 0)]

#For these groups remove the "Sex" column from the data.
with_heart_disease_male = with_heart_disease_male.drop("Sex",axis=1)
with_heart_disease_female = with_heart_disease_female.drop("Sex",axis=1)
no_heart_disease_male = no_heart_disease_male.drop("Sex",axis=1)
no_heart_disease_female = no_heart_disease_female.drop("Sex",axis=1)
```

```
#Visualizing the distribution of the categorical attributes, including the class variable.
ax=1
plt.figure(figsize=(15,15))

for col in ('Sex', 'ChestPainType', 'FastingBloodSugar', 'RestingECG', 'ExerciseAngina','S
  plt.subplot(4,2,ax)
  these_labels = labels[col]
  plt.title(f"Distribution for attribute '{col}':")
  plt.pie(df[col].value_counts().sort_index(),
        autopct = '%1.1f%%', labels=these_labels,
        explode=tuple([0.1] * len(these_labels)))
  plt.axis('equal')
  ax+=1

plt.suptitle('Nominal/Binary Attribute Distribution',y=1.01, size = 16, color = 'black', w
plt.tight_layout()
plt.savefig("nominal_dist.pdf",dpi=1200, bbox_inches='tight')
```



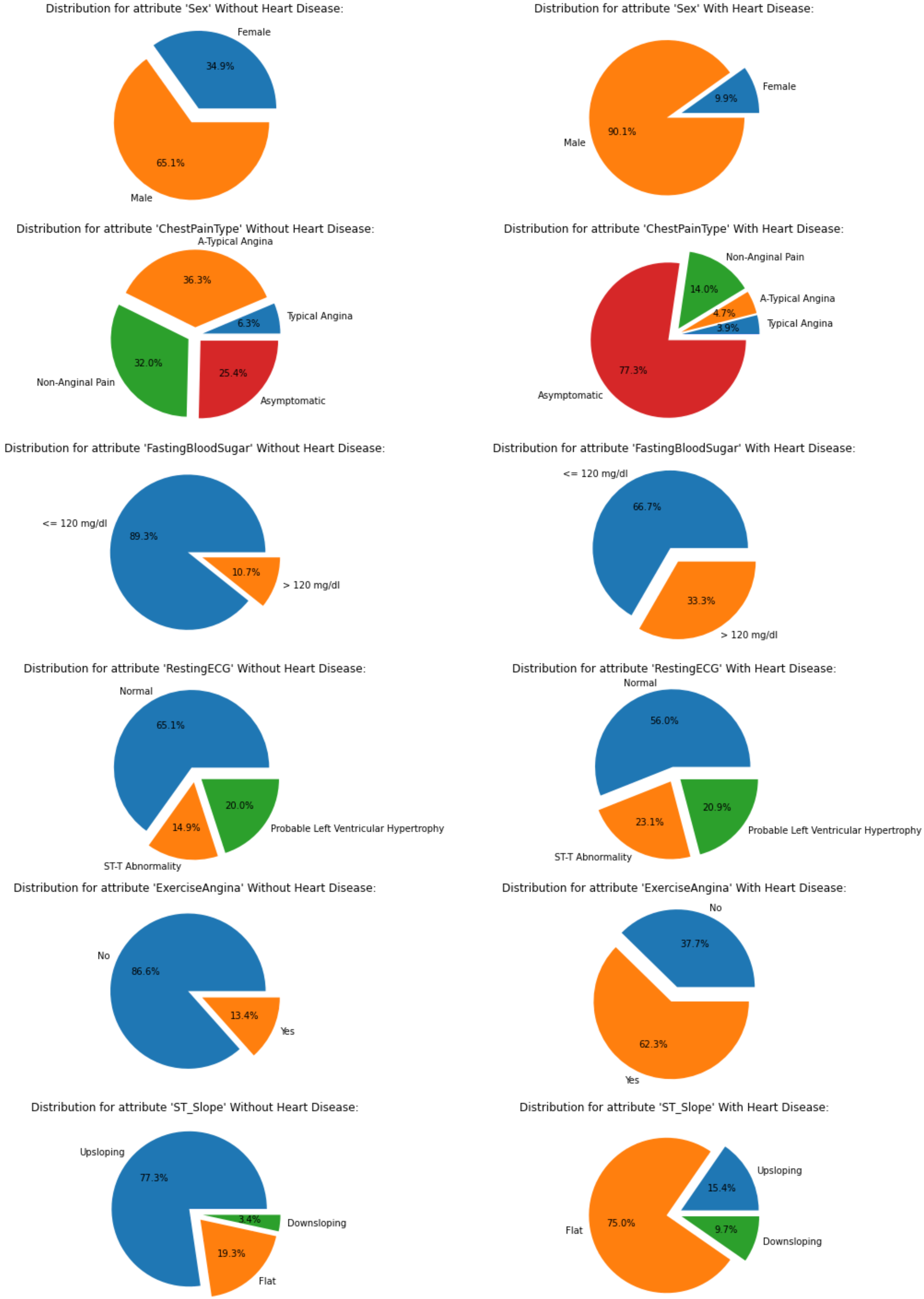

```
#Visualizing the overall distribution of the numeric attributes.
plt.figure(figsize=(15,15))

ax=1
for col in ('Age', 'RestingBP_s', 'Cholesterol', 'MaxHeartRate', 'OldPeak'):
    plt.subplot(3,2,ax)
    plt.title(col)
    sns.kdeplot(x=df[col])
```

```
    ax += 1

plt.suptitle('Numeric Attribute Distribution',y=1.01, size = 16, color = 'black', weight='
plt.tight_layout()
plt.savefig("numeric_dist.pdf",dpi=1200, bbox_inches='tight')
```

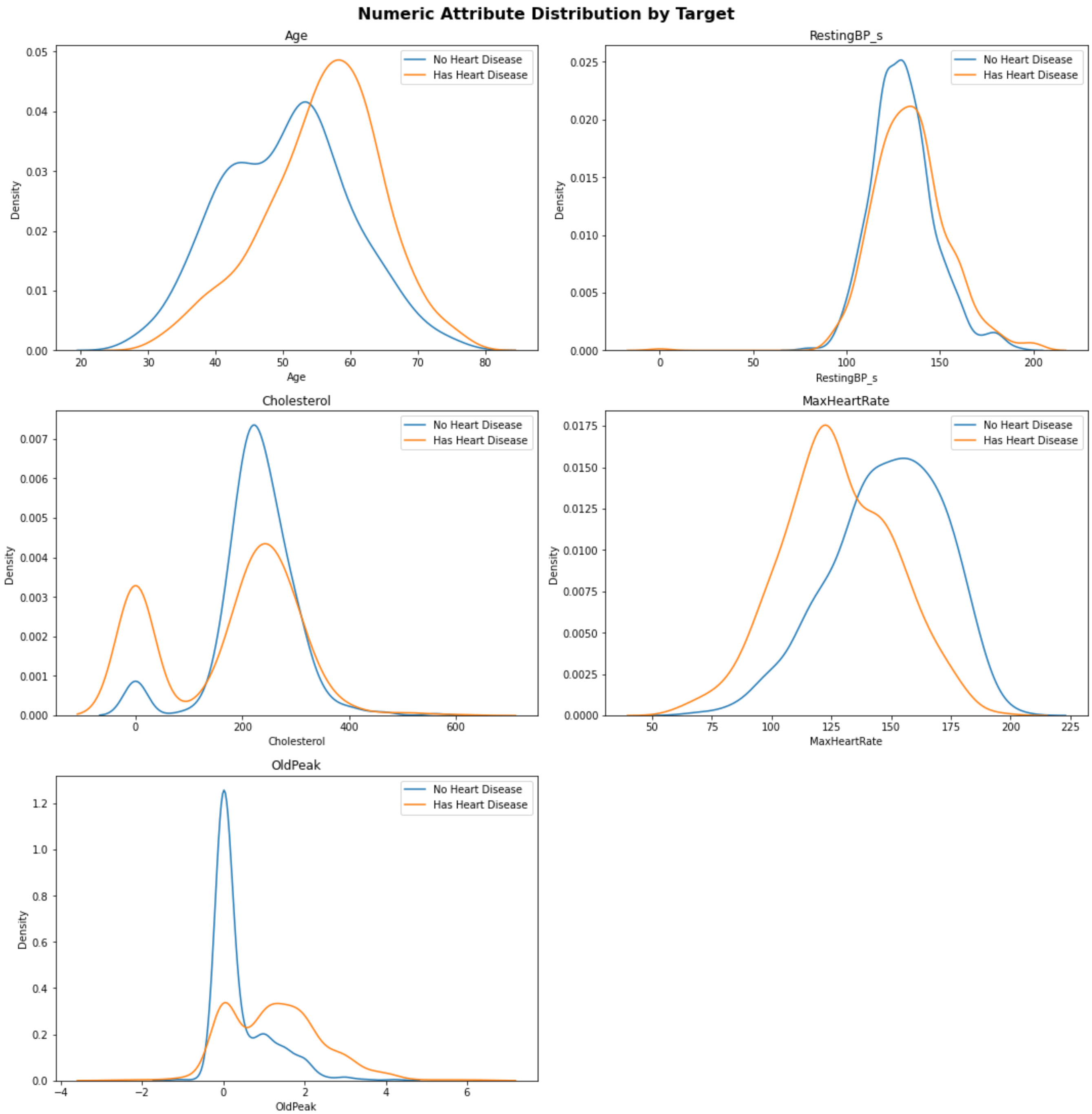



```
#Visualizing the distribution of the categorical attributes, by target
ax=1
plt.figure(figsize=(15,20))
plt.axis('equal')

for col in ('Sex','ChestPainType', 'FastingBloodSugar', 'RestingECG', 'ExerciseAngina','ST
  plt.subplot(6,2,ax)
  these_labels = labels[col]
  plt.title(f"Distribution for attribute '{col}' Without Heart Disease:")
  plt.pie(no_heart_disease[col].value_counts().sort_index(),
        autopct = '%1.1f%%', labels=these_labels,
        explode=tuple([0.1] * len(these_labels)))
  ax+=1
  plt.subplot(6,2,ax)
  these_labels = labels[col]
  plt.title(f"Distribution for attribute '{col}' With Heart Disease:")
  plt.pie(with_heart_disease[col].value_counts().sort_index(),
        autopct = '%1.1f%%', labels=these_labels,
        explode=tuple([0.1] * len(these_labels)))
  ax+=1

plt.suptitle('Nominal/Binary Attribute Distribution by Target',y=1.01, size = 16, color =
plt.tight_layout()
plt.savefig("nominal_dist_by_target.pdf",dpi=1200, bbox_inches='tight')
```

**Nominal/Binary Attribute Distribution by Target**

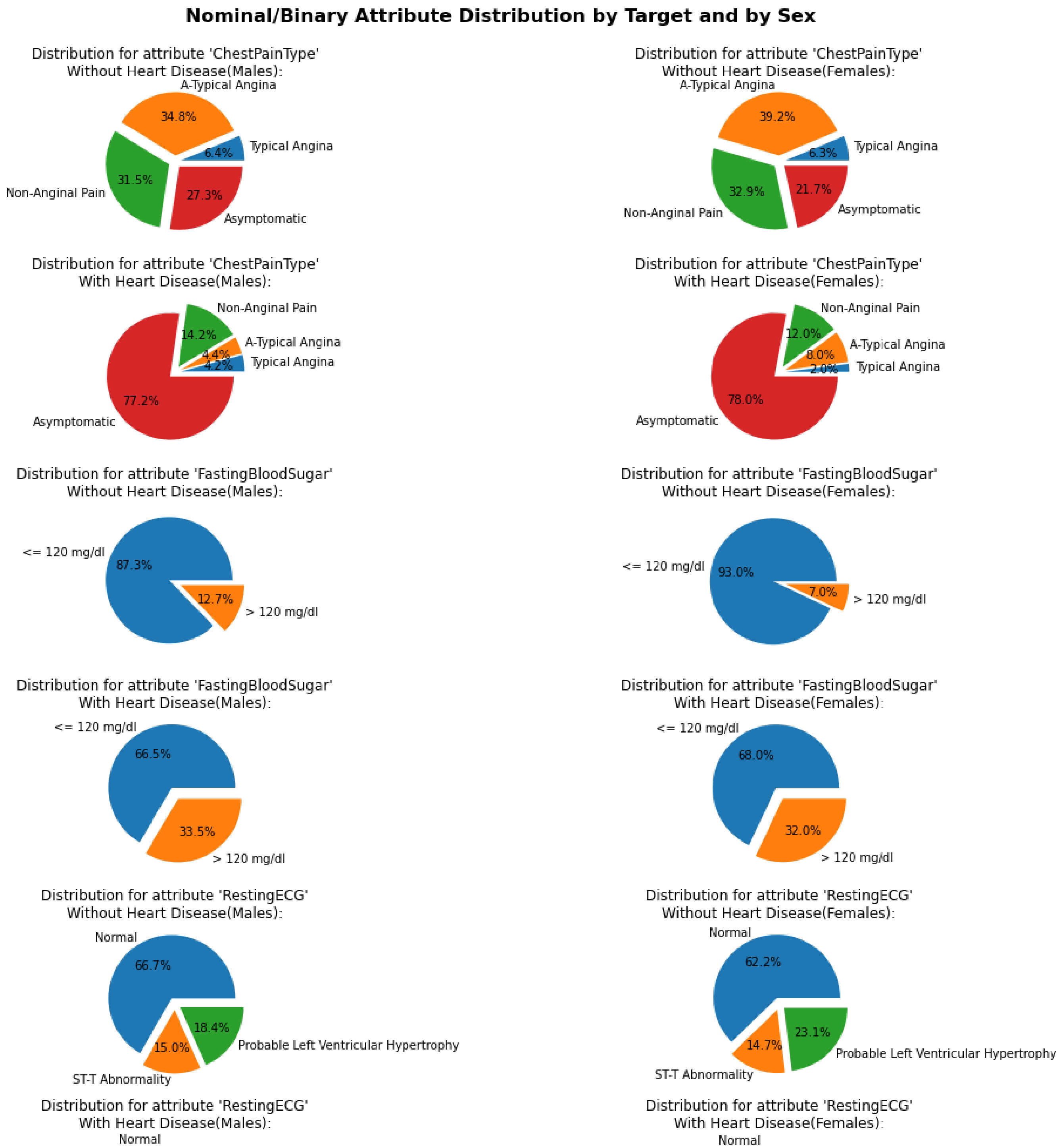


```
#Visualizing the distribution of the numeric attributes by Target:
plt.figure(figsize=(15,15))

ax=1
for col in ('Age', 'RestingBP_s', 'Cholesterol', 'MaxHeartRate', 'OldPeak'):
    plt.subplot(3,2,ax)
    plt.title(col)
    sns.kdeplot(x=no_heart_disease[col],label = "No Heart Disease")
    sns.kdeplot(x=with_heart_disease[col],label = "Has Heart Disease")
    plt.legend()
    ax += 1

plt.suptitle('Numeric Attribute Distribution by Target',y=1.01, size = 16, color = 'black'
plt.tight_layout()
plt.savefig("numeric_dist_by_target.pdf",dpi=1200, bbox_inches='tight')
```



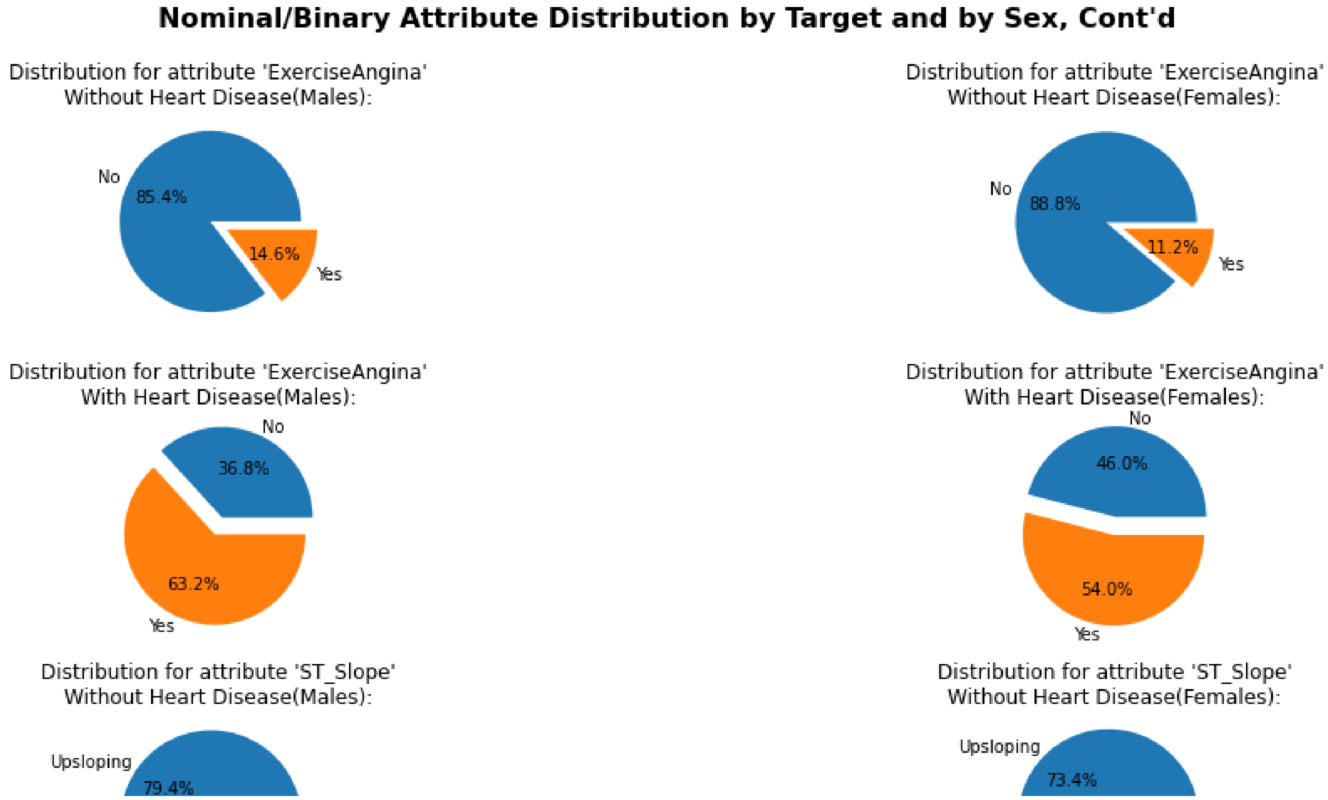

```
#Visualizing the distribution of the categorical attributes, by target and by sex
#For report purposes, breaking this up into two separate pages.
#This one for attributes 'ChestPainType', 'FastingBloodSugar', 'RestingECG'
#
ax=1
plt.figure(figsize=(15,30))
plt.axis('equal')

for col in ('ChestPainType', 'FastingBloodSugar', 'RestingECG'):
  plt.subplot(12,2,ax)
  these_labels = labels[col]
  plt.title(f"Distribution for attribute '{col}'\nWithout Heart Disease(Males):")
  plt.pie(no_heart_disease_male[col].value_counts().sort_index(),
        autopct = '%1.1f%%', labels=these_labels,
```

```
          explode=tuple([0.1] * len(these_labels)))
    ax+=1

    plt.subplot(12,2,ax)
    these_labels = labels[col]
    plt.title(f"Distribution for attribute '{col}'\nWithout Heart Disease(Females):")
    plt.pie(no_heart_disease_female[col].value_counts().sort_index(),
          autopct = '%1.1f%%', labels=these_labels,
          explode=tuple([0.1] * len(these_labels)))
    ax+=1

    plt.subplot(12,2,ax)
    these_labels = labels[col]
    plt.title(f"Distribution for attribute '{col}'\nWith Heart Disease(Males):")
    plt.pie(with_heart_disease_male[col].value_counts().sort_index(),
          autopct = '%1.1f%%', labels=these_labels,
          explode=tuple([0.1] * len(these_labels)))
    ax+=1

    plt.subplot(12,2,ax)
    these_labels = labels[col]
    plt.title(f"Distribution for attribute '{col}'\nWith Heart Disease(Females):")
    plt.pie(with_heart_disease_female[col].value_counts().sort_index(),
          autopct = '%1.1f%%', labels=these_labels,
          explode=tuple([0.1] * len(these_labels)))
    ax+=1

plt.suptitle('Nominal/Binary Attribute Distribution by Target and by Sex',y=1.01, size = 1
plt.tight_layout()
plt.savefig("nominal_dist_by_target_by_sex1.pdf",dpi=1200, bbox_inches='tight')
```

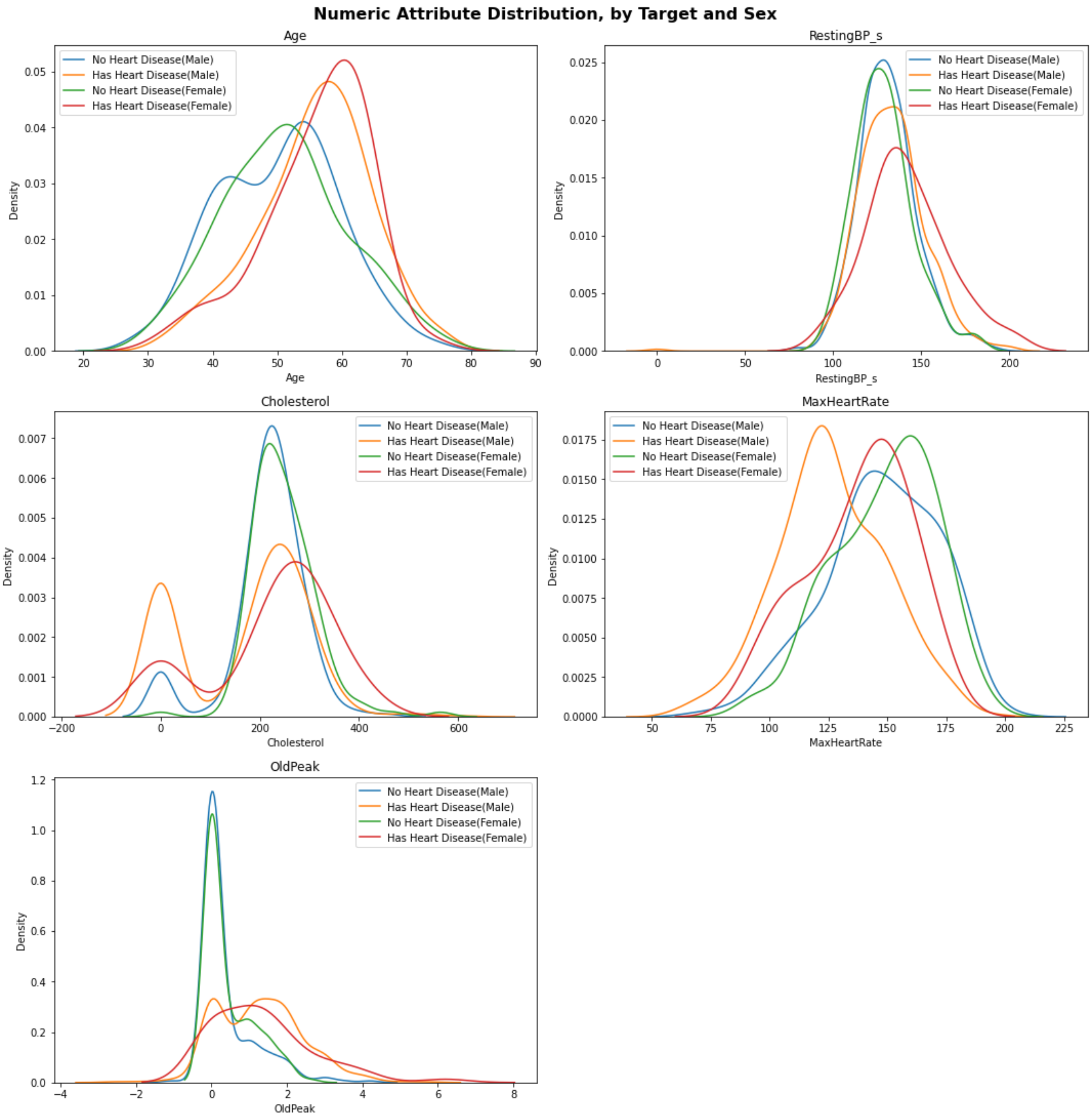



```
#Visualizing the distribution of the categorical attributes, by target and by sex
#For report purposes, breaking this up into two separate pages.
#This one for attributes 'ExerciseAngina','ST_Slope'
#
ax=1
plt.figure(figsize=(15,30))
plt.axis('equal')

for col in ('ExerciseAngina','ST_Slope'):
  plt.subplot(12,2,ax)
  these_labels = labels[col]
  plt.title(f"Distribution for attribute '{col}'\nWithout Heart Disease(Males):")
```

```
    plt.pie(no_heart_disease_male[col].value_counts().sort_index(),
          autopct = '%1.1f%%', labels=these_labels,
          explode=tuple([0.1] * len(these_labels)))
    ax+=1

    plt.subplot(12,2,ax)
    these_labels = labels[col]
    plt.title(f"Distribution for attribute '{col}'\nWithout Heart Disease(Females):")
    plt.pie(no_heart_disease_female[col].value_counts().sort_index(),
          autopct = '%1.1f%%', labels=these_labels,
          explode=tuple([0.1] * len(these_labels)))
    ax+=1

    plt.subplot(12,2,ax)
    these_labels = labels[col]
    plt.title(f"Distribution for attribute '{col}'\nWith Heart Disease(Males):")
    plt.pie(with_heart_disease_male[col].value_counts().sort_index(),
          autopct = '%1.1f%%', labels=these_labels,
          explode=tuple([0.1] * len(these_labels)))
    ax+=1

    plt.subplot(12,2,ax)
    these_labels = labels[col]
    plt.title(f"Distribution for attribute '{col}'\nWith Heart Disease(Females):")
    plt.pie(with_heart_disease_female[col].value_counts().sort_index(),
          autopct = '%1.1f%%', labels=these_labels,
          explode=tuple([0.1] * len(these_labels)))
    ax+=1

plt.suptitle('Nominal/Binary Attribute Distribution by Target and by Sex, Cont\'d',y=1.01,
plt.tight_layout()
plt.savefig("nominal_dist_by_target_by_sex2.pdf",dpi=1200, bbox_inches='tight')
```




```
#Visualizing the distribution of the numerical attributes, by target and by sex
plt.figure(figsize=(15,15))

ax=1
for col in ('Age', 'RestingBP_s', 'Cholesterol', 'MaxHeartRate', 'OldPeak'):
    plt.subplot(3,2,ax)
    plt.title(col)
    sns.kdeplot(x=no_heart_disease_male[col],label = "No Heart Disease(Male)")
    sns.kdeplot(x=with_heart_disease_male[col],label = "Has Heart Disease(Male)")
    sns.kdeplot(x=no_heart_disease_female[col],label = "No Heart Disease(Female)")
    sns.kdeplot(x=with_heart_disease_female[col],label = "Has Heart Disease(Female)")
    plt.legend()
    ax += 1

plt.suptitle('Numeric Attribute Distribution, by Target and Sex',y=1.01, size = 16, color
plt.tight_layout()
plt.savefig("numeric_dist_by_target_by_sex.pdf",dpi=1200, bbox_inches='tight')
```




## Outlier Detection:

```
#Check for outliers on numeric attributes
```

```
#Using for outlier detection three methods.
#Note: for the next stage in this project, Module 3,
#one or more of these outlier detection methods will be used.
#For now, we only want to see how many outliers per attribute are detected with each appro
#
#Methods:
#  #1 1.5IQR range
#  #2 mean +/- 3*ST-DEV (same as GT Absolute(Z-Score))
#  #3 Rejecting those with a value of zero (based on visualization, only needed for 'Chole

def IQR1_5_upper(data, col):
  Q3 = np.quantile(data[col], 0.75)
  Q1 = np.quantile(data[col], 0.25)
  IQR = Q3 - Q1
  return(Q3+(1.5*IQR))

def IQR1_5_lower(data, col):
  Q3 = np.quantile(data[col], 0.75)
  Q1 = np.quantile(data[col], 0.25)
  IQR = Q3 - Q1
  return(Q1-(1.5*IQR))


for col in ('Age', 'RestingBP_s', 'Cholesterol', 'MaxHeartRate', 'OldPeak'):
  upper1 = IQR1_5_upper(df,col)
  lower1 = IQR1_5_lower(df,col)
  stdev3 = 3*df[col].std()
  mean = df[col].mean()
  upper2 = mean + stdev3
  lower2 = mean - stdev3

  these_outliers1 = df[(df[col] < lower1) | (df[col] > upper1)]
  these_outliers2 = df[(df[col] < lower2) | (df[col] > upper2)]
  these_outliers3 = df[df[col] == 0]

  print(f"For attribute '{col}': The mean is {mean}, stdev3 is {stdev3}")
  print(f"For 1.5IQR the lower range is {lower1} the upper range is {upper1}")
  print(f"For mean +/- 3STDEV the lower range is {lower2} the upper range is {upper2}")
  print(f"\n")

  print(f"Using 1.5IQR Method:")
  if (these_outliers1.shape[0] > 1):
    print(f"For attribute '{col}': There are {these_outliers1.shape[0]} outliers:\n")
    print(these_outliers1)
    print("\n")
  elif (these_outliers1.shape[0] == 1):
    print(f"For attribute '{col}': There is {these_outliers1.shape[0]} outlier:\n")
    print(these_outliers1)
    print("\n")
  else:
    print(f"For attribute '{col}': There are no outliers.\n")
```

```
    print("\n")

  print(f"Using mean +/- 3STDEV Method:")
  if (these_outliers2.shape[0] > 1):
    print(f"For attribute '{col}': There are {these_outliers2.shape[0]} outliers:\n")
    print(these_outliers2)
    print("\n")
  elif (these_outliers2.shape[0] == 1):
    print(f"For attribute '{col}': There is {these_outliers2.shape[0]} outlier:\n")
    print(these_outliers2)
    print("\n")
  else:
    print(f"For attribute '{col}': There are no outliers.\n")
    print("\n")

  if(col == 'Cholesterol'):
    print(f"Identifying 'zero' values(for 'Cholesterol') Method:")
    if (these_outliers3.shape[0] > 1):
      print(f"For attribute '{col}': There are {these_outliers3.shape[0]} outliers:\n")
      print(these_outliers3)
      print("\n")
    elif (these_outliers3.shape[0] == 1):
      print(f"For attribute '{col}': There is {these_outliers3.shape[0]} outlier:\n")
      print(these_outliers3)
      print("\n")
    else:
      print(f"For attribute '{col}': There are no outliers.\n")
      print("\n")

print("\n\n")
```

```
166  50   1             4         140         231                 0
324  46   1             4         100           0                 1
500  65   1             4         136         248                 0
520  61   1             4         120         282                 0
536  74   1             4         150         258                 1
558  64   1             4         134         273                 0
623  63   0             4         150         407                 0
701  59   1             1         178         270                 0
731  56   0             4         200         288                 1
770  55   1             4         140         217                 0
774  38   1             1         120         231                 0
790  51   1             4         140         298                 0
849  62   0             4         160         164                 0
899  58   1             4         114         318                 0
907  63   1             4         140         187                 0

     RestingECG  MaxHeartRate  ExerciseAngina  OldPeak  ST_Slope  Target
68            1            82               1      4.0         2       1

166           1           140               1      5.0         2       1
324           1           133               0     -2.6         2       1
500           0           140               1      4.0         3       1
520           1           135               1      4.0         3       1
536           1           130               1      4.0         3       1
```

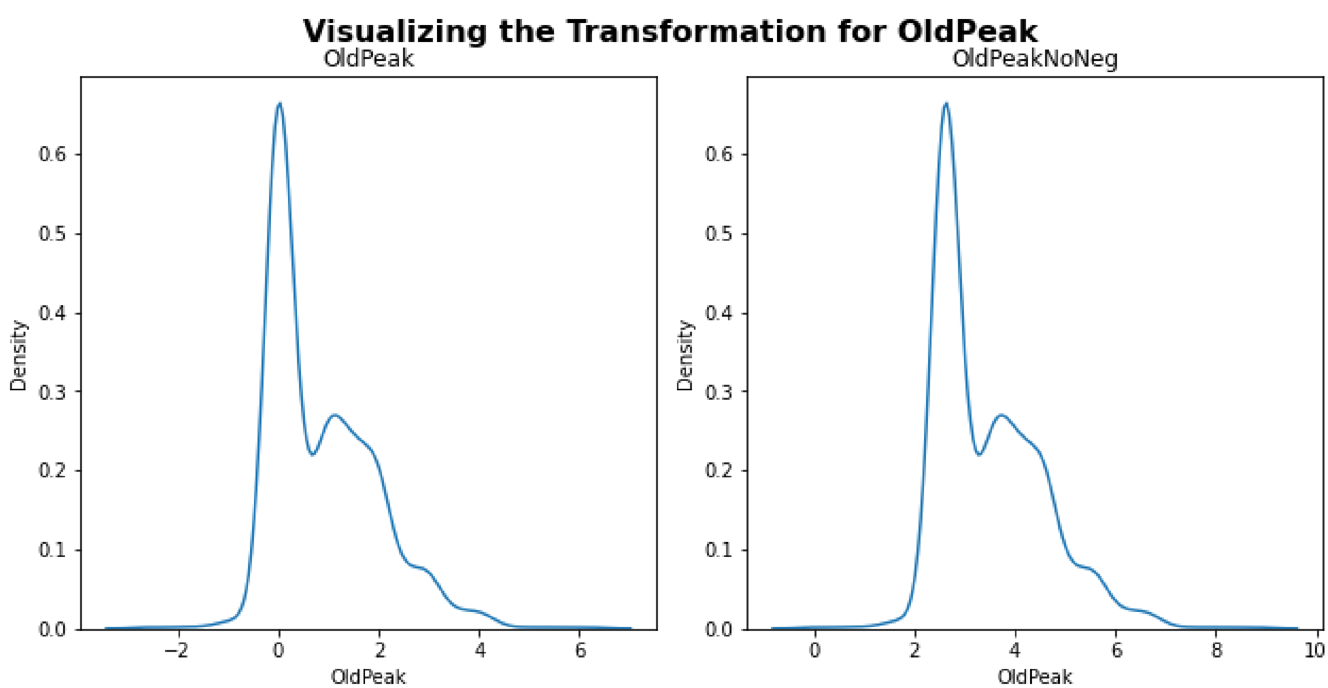
```
558            0            102                 1      4.0         3       1
623            2            154                 0      4.0         2       1
701            2            145                 0      4.2         3       0
731            2            133                 1      4.0         3       1
770            0            111                 1      5.6         3       1
774            0            182                 1      3.8         2       1
790            0            122                 1      4.2         2       1
849            2            145                 0      6.2         3       1
899            1            140                 0      4.4         3       1
907            2            144                 1      4.0         1       1


Using mean +/- 3STDEV Method:
For attribute 'OldPeak': There are 7 outliers:

     Age  Sex  ChestPainType  RestingBP_s  Cholesterol  FastingBloodSugar  \
166   50    1              4          140          231                  0
324   46    1              4          100            0                  1
701   59    1              1          178          270                  0
770   55    1              4          140          217                  0
790   51    1              4          140          298                  0
849   62    0              4          160          164                  0
899   58    1              4          114          318                  0

     RestingECG  MaxHeartRate  ExerciseAngina  OldPeak  ST_Slope  Target
166           1           140               1      5.0         2       1
324           1           133               0     -2.6         2       1
701           2           145               0      4.2         3       0
770           0           111               1      5.6         3       1
790           0           122               1      4.2         2       1
849           2           145               0      6.2         3       1
899           1           140               0      4.4         3       1
```

### Data Manipulation, for Outliers, and Model Considerations:

```
#Before we can run multinomial Naive Bayes we must remove any negative numbers in the data
mins = df.min()
print(mins)
print("\n\n")
#There are only negative values for attribute 'OldPeak'
#Applying a simple shift to eliminate any negatives.

df_no_negs = df.copy()
df_no_negs.name = "No Negatives Data-Set"

df_no_negs['OldPeak'] = df_no_negs['OldPeak'] + abs(df_no_negs['OldPeak'].min())

#Visualizing the overall distribution of 'OldPeak' before and after modification for negat
plt.figure(figsize=(10,5))
```

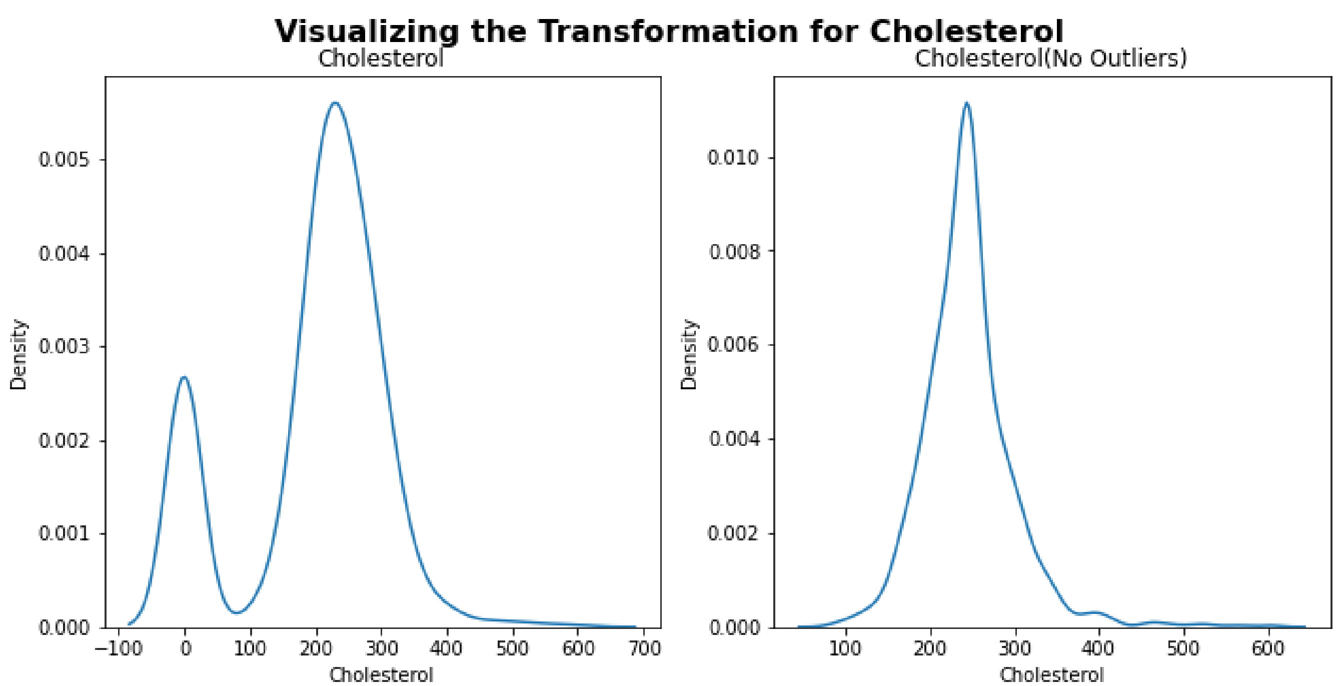
```
plt.subplot(1,2,1)
plt.title('OldPeak')
sns.kdeplot(x=df['OldPeak'])

plt.subplot(1,2,2)
plt.title('OldPeakNoNeg')
sns.kdeplot(x=df_no_negs['OldPeak'])

plt.suptitle('Visualizing the Transformation for OldPeak',y=1.01, size = 16, color = 'blac
plt.tight_layout()
plt.savefig("oldpeak_transformation.pdf",dpi=1200, bbox_inches='tight')

```

```
Age                 28.0
Sex                  0.0
ChestPainType        1.0
RestingBP_s          0.0
Cholesterol          0.0
FastingBloodSugar    0.0
RestingECG           0.0
MaxHeartRate        60.0
ExerciseAngina       0.0
OldPeak             -2.6
ST_Slope             1.0
Target               0.0
dtype: float64
```




```
#Only addressing outliers for attribute Cholesterol, specifically the instances where Chol
#All other outliers appear in much smaller quantities.
```

```
df_outliers_addressed = df_no_negs.copy()
df_outliers_addressed.name = "Outliers Addressed Data-Set"

cholesterol_mean = df_outliers_addressed['Cholesterol'][df_outliers_addressed['Cholesterol

df_outliers_addressed['Cholesterol'].replace(to_replace=0.0, value=cholesterol_mean, inpla


#Visualizing the overall distribution Cholesterol before and after dealing with outliers.
plt.figure(figsize=(10,5))

plt.subplot(1,2,1)
plt.title('Cholesterol')
sns.kdeplot(x=df['Cholesterol'])

plt.subplot(1,2,2)
plt.title('Cholesterol(No Outliers)')
sns.kdeplot(x=df_outliers_addressed['Cholesterol'])

plt.suptitle('Visualizing the Transformation for Cholesterol',y=1.01, size = 16, color = '
plt.tight_layout()
```




### Normalize Numeric Data for potential model training:

```
#Normalizing numeric data to see if this helps, or hinders the accuracy of the ML models.
df_normalized = df_outliers_addressed.copy()
df_normalized.name = "Normalized Data-Set"

for col in ('Age', 'RestingBP_s', 'Cholesterol', 'MaxHeartRate', 'OldPeak'):
```

```
  df_normalized[col] = (df[col]-df[col].min())/(df[col].max()-df[col].min())
```

### Create ONE-HOT columns for all categorical attributes:

```
#For Nominal & Binary (ie categorical) attributes, perform one-hot conversion.
#convert only categorical variables/features to dummy/one-hot features
cat_cols = ['Sex','ChestPainType', 'FastingBloodSugar', 'RestingECG', 'ExerciseAngina','ST_

df_onehot = pd.get_dummies(df, columns=cat_cols, prefix = cat_cols)
df_onehot.name = "Original Data-Set, onehot"

df_onehot_no_negs = pd.get_dummies(df_no_negs, columns=cat_cols, prefix = cat_cols)
df_onehot_no_negs.name = "No negatives Data-Set, onehot"

df_onehot_outliers_addressed = pd.get_dummies(df_outliers_addressed, columns=cat_cols, pre
df_onehot_outliers_addressed.name = "Outliers Addressed Data-Set, onehot"

df_onehot_normalized = pd.get_dummies(df_normalized, columns=cat_cols, prefix = cat_cols)
df_onehot_normalized.name = "Normalized Data-Set, onehot"

```

### ML Algorithms:

```
def do_DT(df,levels,class_col_name,verbose=0):
  #not disabling randomness.
  #np.random.seed(0)

  # Split dataset into training set and test set
  feature_names=df.columns[df.columns != class_col_name ]
  # 80% training and 20% test
  X_train, X_test, Y_train, Y_test = train_test_split(df.loc[:, feature_names], df[class_c

  clf = tree.DecisionTreeClassifier(max_depth=levels,criterion='gini')
  clf = clf.fit(X_train, Y_train)
  if (verbose >= 1):
    print(f"Successfuly trained the decision tree for {levels} levels...")

  # Let's make the prdictions on the test set  that we set aside earlier using the trained
  Y_pred = clf.predict(X_test)

  cf=confusion_matrix(Y_test, Y_pred)
  tn, fp, fn, tp=cf.ravel()
  tpr=0.0
  fpr=0.0
  tpr = tp/(tp+fp)
  fpr = fp/(fp+tn)
  fnr = fn/(fn+tp)

```

```
  if (verbose >= 2):
    print ("Confusion Matrix")
    print(cf)
    print("")
    print ("TP: ", tp,", FP: ", fp,", TN: ", tn,", FN:", fn)
    print ("TPR: ",tpr,", FPR: ",fpr, "FNR: ",fnr)

  #print precision, recall, and accuracy from the perspective of each of the class (0 and
  if (verbose >= 2):
    print(classification_report(Y_test, Y_pred, digits=3))

  accuracy = accuracy_score(Y_test, Y_pred)
  f1_weighted = f1_score(Y_test, Y_pred,average='weighted')

  if (verbose >= 1):
    print(f"Accuracy is: {accuracy}")
    print(f"F1 Weighted is: {f1_weighted}")
    print("")

  return(accuracy,f1_weighted,tpr,fpr,fnr)
```

```
def do_mnNB(df,class_col_name,verbose=0):
  #not disabling randomness.
  #np.random.seed(0)

  # Split dataset into training set and test set
  feature_names=df.columns[df.columns != class_col_name ]
  # 80% training and 20% test
  X_train, X_test, Y_train, Y_test = train_test_split(df.loc[:, feature_names], df[class_c

  #Create a MultiNomial NB Classifier
  nb = MultinomialNB()

  #Train the model using the training sets
  nb.fit(X_train, Y_train)

  #Predict the response for test dataset
  Y_pred = nb.predict(X_test)

  if (verbose >= 2):
    print ("Total Columns (including class)",len(df.columns))
    print("Classes ",nb.classes_)
    print("Number of records for classes ",nb.class_count_)
    print("Log prior probability for classes ", nb.class_log_prior_)
    print("Log conditional probability for each feature given a class\n",nb.feature_log_pr

  cf=confusion_matrix(Y_test, Y_pred)
  tn, fp, fn, tp=cf.ravel()
  tpr = tp/(tp+fp)
  fpr = fp/(fp+tn)
  fnr = fn/(fn+tp)
```

```

  if (verbose >= 2):
    print ("Confusion Matrix")
    print(cf)
    print("")
    print ("TP: ", tp,", FP: ", fp,", TN: ", tn,", FN:", fn)
    print ("TPR: ",tpr,", FPR: ",fpr, "FNR: ",fnr)

  if (verbose >= 2):
    print(classification_report(Y_test, Y_pred, digits=3))

  accuracy = accuracy_score(Y_test, Y_pred)
  f1_weighted = f1_score(Y_test, Y_pred,average='weighted')

  if (verbose >= 1):
    print(f"Accuracy is: {accuracy}")
    print(f"F1 Weighted is: {f1_weighted}")
    print("")

  return(accuracy,f1_weighted,tpr,fpr,fnr)
```

```
def do_gaNB(df,class_col_name,verbose=0):
  #not disabling randomness.
  #np.random.seed(0)

  # Split dataset into training set and test set
  feature_names=df.columns[df.columns != class_col_name ]
  # 80% training and 20% test
  X_train, X_test, Y_train, Y_test = train_test_split(df.loc[:, feature_names], df[class_c

  #Create a Gaussian NB Classifier
  nb = GaussianNB()

  #Train the model using the training sets
  nb.fit(X_train, Y_train)

  #Predict the response for test dataset
  Y_pred = nb.predict(X_test)

  if (verbose >= 2):
    print ("Total Columns (including class)",len(df.columns))
    print("Number of records for classes ",nb.class_count_)

  cf=confusion_matrix(Y_test, Y_pred)
  tn, fp, fn, tp=cf.ravel()
  tpr = tp/(tp+fp)
  fpr = fp/(fp+tn)
  fnr = fn/(fn+tp)

  if (verbose >= 2):
```

```
    print ("Confusion Matrix")
    print(cf)
    print("")
    print ("TP: ", tp,", FP: ", fp,", TN: ", tn,", FN:", fn)
    print ("TPR: ",tpr,", FPR: ",fpr, "FNR: ",fnr)

  if (verbose >= 2):
    print(classification_report(Y_test, Y_pred, digits=3))

  accuracy = accuracy_score(Y_test, Y_pred)
  f1_weighted = f1_score(Y_test, Y_pred,average='weighted')

  if (verbose >= 1):
    print(f"Accuracy is: {accuracy}")
    print(f"F1 Weighted is: {f1_weighted}")
    print("")

  return(accuracy,f1_weighted,tpr,fpr,fnr)
```

```
def do_LR(df,class_col_name,verbose=0):
  #not disabling randomness.
  #np.random.seed(0)

  # Split dataset into training set and test set
  feature_names=df.columns[df.columns != class_col_name ]
  # 80% training and 20% test
  X_train, X_test, Y_train, Y_test = train_test_split(df.loc[:, feature_names], df[class_c

  lr = LogisticRegression(max_iter=2000)

  #Train the model using the training sets
  lr.fit(X_train, Y_train)

  #Predict the response for test dataset
  Y_pred = lr.predict(X_test)

  if (verbose >= 2):
    print ("Total Columns (including class)",len(df.columns))

  cf=confusion_matrix(Y_test, Y_pred)
  tn, fp, fn, tp=cf.ravel()
  tpr = tp/(tp+fp)
  fpr = fp/(fp+tn)
  fnr = fn/(fn+tp)

  if (verbose >= 2):
    print ("Confusion Matrix")
    print(cf)
    print("")
    print ("TP: ", tp,", FP: ", fp,", TN: ", tn,", FN:", fn)
```

```
    print ("TPR: ",tpr,", FPR: ",fpr, "FNR: ",fnr)

  if (verbose >= 2):
    print(classification_report(Y_test, Y_pred, digits=3))

  accuracy = accuracy_score(Y_test, Y_pred)
  f1_weighted = f1_score(Y_test, Y_pred,average='weighted')

  if (verbose >= 1):
    print(f"Accuracy is: {accuracy}")
    print(f"F1 Weighted is: {f1_weighted}")
    print("")

  return(accuracy,f1_weighted,tpr,fpr,fnr)
```

```
def do_KNN(df,class_col_name,verbose=0):
  #not disabling randomness.
  #np.random.seed(0)

  # Split dataset into training set and test set
  feature_names=df.columns[df.columns != class_col_name ]
  # 80% training and 20% test
  X_train, X_test, Y_train, Y_test = train_test_split(df.loc[:, feature_names], df[class_c

  knn = KNeighborsClassifier()

  #Train the model using the training sets
  knn.fit(X_train, Y_train)

  #Predict the response for test dataset
  Y_pred = knn.predict(X_test)

  if (verbose >= 2):
    print ("Total Columns (including class)",len(df.columns))

  cf=confusion_matrix(Y_test, Y_pred)
  tn, fp, fn, tp=cf.ravel()
  tpr = tp/(tp+fp)
  fpr = fp/(fp+tn)
  fnr = fn/(fn+tp)

  if (verbose >= 2):
    print ("Confusion Matrix")
    print(cf)
    print("")
    print ("TP: ", tp,", FP: ", fp,", TN: ", tn,", FN:", fn)
    print ("TPR: ",tpr,", FPR: ",fpr, "FNR: ",fnr)

  if (verbose >= 2):
    print(classification_report(Y_test, Y_pred, digits=3))
```

```
  accuracy = accuracy_score(Y_test, Y_pred)
  f1_weighted = f1_score(Y_test, Y_pred,average='weighted')

  if (verbose >= 1):
    print(f"Accuracy is: {accuracy}")
    print(f"F1 Weighted is: {f1_weighted}")
    print("")

  return(accuracy,f1_weighted,tpr,fpr,fnr)
```

```
def do_RF(df,class_col_name,verbose=0):
  #not disabling randomness.
  #np.random.seed(0)

  # Split dataset into training set and test set
  feature_names=df.columns[df.columns != class_col_name ]
  # 80% training and 20% test
  X_train, X_test, Y_train, Y_test = train_test_split(df.loc[:, feature_names], df[class_c

  rf = RandomForestClassifier()

  #Train the model using the training sets
  rf.fit(X_train, Y_train)

  #Predict the response for test dataset
  Y_pred = rf.predict(X_test)

  if (verbose >= 2):
    print ("Total Columns (including class)",len(df.columns))

  cf=confusion_matrix(Y_test, Y_pred)
  tn, fp, fn, tp=cf.ravel()
  tpr = tp/(tp+fp)
  fpr = fp/(fp+tn)
  fnr = fn/(fn+tp)

  if (verbose >= 2):
    print ("Confusion Matrix")
    print(cf)
    print("")
    print ("TP: ", tp,", FP: ", fp,", TN: ", tn,", FN:", fn)
    print ("TPR: ",tpr,", FPR: ",fpr, "FNR: ",fnr)

  if (verbose >= 2):
    print(classification_report(Y_test, Y_pred, digits=3))

  accuracy = accuracy_score(Y_test, Y_pred)
  f1_weighted = f1_score(Y_test, Y_pred,average='weighted')

```

```
  if (verbose >= 1):
    print(f"Accuracy is: {accuracy}")
    print(f"F1 Weighted is: {f1_weighted}")
    print("")

  return(accuracy,f1_weighted,tpr,fpr,fnr)


def do_RF_tuned(df,class_col_name,verbose=0):
  #not disabling randomness.
  #np.random.seed(0)

  # Split dataset into training set and test set
  feature_names=df.columns[df.columns != class_col_name ]
  # 80% training and 20% test
  X_train, X_test, Y_train, Y_test = train_test_split(df.loc[:, feature_names], df[class_c

  rf = RandomForestClassifier(min_samples_leaf = 5, n_estimators = 400)

  #Train the model using the training sets
  rf.fit(X_train, Y_train)

  #Predict the response for test dataset
  Y_pred = rf.predict(X_test)

  if (verbose >= 2):
    print ("Total Columns (including class)",len(df.columns))

  cf=confusion_matrix(Y_test, Y_pred)
  tn, fp, fn, tp=cf.ravel()
  tpr = tp/(tp+fp)
  fpr = fp/(fp+tn)
  fnr = fn/(fn+tp)

  if (verbose >= 2):
    print ("Confusion Matrix")
    print(cf)
    print("")
    print ("TP: ", tp,", FP: ", fp,", TN: ", tn,", FN:", fn)
    print ("TPR: ",tpr,", FPR: ",fpr, "FNR: ",fnr)

  if (verbose >= 2):
    print(classification_report(Y_test, Y_pred, digits=3))

  accuracy = accuracy_score(Y_test, Y_pred)
  f1_weighted = f1_score(Y_test, Y_pred,average='weighted')

  if (verbose >= 1):
    print(f"Accuracy is: {accuracy}")
    print(f"F1 Weighted is: {f1_weighted}")
    print("")
```

```
  return(accuracy,f1_weighted,tpr,fpr,fnr)
```

```
def do_SVM(df,class_col_name,verbose=0):
  #not disabling randomness.
  #np.random.seed(0)

  # Split dataset into training set and test set
  feature_names=df.columns[df.columns != class_col_name ]
  # 80% training and 20% test
  X_train, X_test, Y_train, Y_test = train_test_split(df.loc[:, feature_names], df[class_c

  svm = SVC()

  #Train the model using the training sets
  svm.fit(X_train, Y_train)

  #Predict the response for test dataset
  Y_pred = svm.predict(X_test)

  if (verbose >= 2):
    print ("Total Columns (including class)",len(df.columns))

  cf=confusion_matrix(Y_test, Y_pred)
  tn, fp, fn, tp=cf.ravel()
  tpr = tp/(tp+fp)
  fpr = fp/(fp+tn)
  fnr = fn/(fn+tp)


  if (verbose >= 2):
    print ("Confusion Matrix")
    print(cf)
    print("")
    print ("TP: ", tp,", FP: ", fp,", TN: ", tn,", FN:", fn)
    print ("TPR: ",tpr,", FPR: ",fpr, "FNR: ",fnr)

  if (verbose >= 2):
    print(classification_report(Y_test, Y_pred, digits=3))

  accuracy = accuracy_score(Y_test, Y_pred)
  f1_weighted = f1_score(Y_test, Y_pred,average='weighted')

  if (verbose >= 1):
    print(f"Accuracy is: {accuracy}")
    print(f"F1 Weighted is: {f1_weighted}")
    print("")

  return(accuracy,f1_weighted,tpr,fpr,fnr)
```

```
def do_XGB(df,class_col_name,verbose=0):
  #not disabling randomness.
  #np.random.seed(0)

  # Split dataset into training set and test set
  feature_names=df.columns[df.columns != class_col_name ]
  # 80% training and 20% test
  X_train, X_test, Y_train, Y_test = train_test_split(df.loc[:, feature_names], df[class_c

  xgb = XGBClassifier()

  #Train the model using the training sets
  xgb.fit(X_train, Y_train)

  #Predict the response for test dataset
  Y_pred = xgb.predict(X_test)

  if (verbose >= 2):
    print ("Total Columns (including class)",len(df.columns))

  cf=confusion_matrix(Y_test, Y_pred)
  tn, fp, fn, tp=cf.ravel()
  tpr = tp/(tp+fp)
  fpr = fp/(fp+tn)
  fnr = fn/(fn+tp)

  if (verbose >= 2):
    print ("Confusion Matrix")
    print(cf)
    print("")
    print ("TP: ", tp,", FP: ", fp,", TN: ", tn,", FN:", fn)
    print ("TPR: ",tpr,", FPR: ",fpr, "FNR: ",fnr)

  if (verbose >= 2):
    print(classification_report(Y_test, Y_pred, digits=3))

  accuracy = accuracy_score(Y_test, Y_pred)
  f1_weighted = f1_score(Y_test, Y_pred,average='weighted')

  if (verbose >= 1):
    print(f"Accuracy is: {accuracy}")
    print(f"F1 Weighted is: {f1_weighted}")
    print("")

  return(accuracy,f1_weighted,tpr,fpr,fnr)

def do_XGB_tuned(df,class_col_name,verbose=0):
  #not disabling randomness.
  #np.random.seed(0)

  # Split dataset into training set and test set
```

```
  feature_names=df.columns[df.columns != class_col_name ]
  # 80% training and 20% test
  X_train, X_test, Y_train, Y_test = train_test_split(df.loc[:, feature_names], df[class_c

  xgb = XGBClassifier(n_estimators = 500, learning_rate = 0.1)

  #Train the model using the training sets
  xgb.fit(X_train, Y_train)

  #Predict the response for test dataset
  Y_pred = xgb.predict(X_test)

  if (verbose >= 2):
    print ("Total Columns (including class)",len(df.columns))

  cf=confusion_matrix(Y_test, Y_pred)
  tn, fp, fn, tp=cf.ravel()
  tpr = tp/(tp+fp)
  fpr = fp/(fp+tn)
  fnr = fn/(fn+tp)

  if (verbose >= 2):
    print ("Confusion Matrix")
    print(cf)
    print("")
    print ("TP: ", tp,", FP: ", fp,", TN: ", tn,", FN:", fn)
    print ("TPR: ",tpr,", FPR: ",fpr, "FNR: ",fnr)

  if (verbose >= 2):
    print(classification_report(Y_test, Y_pred, digits=3))

  accuracy = accuracy_score(Y_test, Y_pred)
  f1_weighted = f1_score(Y_test, Y_pred,average='weighted')

  if (verbose >= 1):
    print(f"Accuracy is: {accuracy}")
    print(f"F1 Weighted is: {f1_weighted}")
    print("")

  return(accuracy,f1_weighted,tpr,fpr,fnr)
```

### Initial Run of All ML Algorithms:

```
#Initial Run of all ML Algorithms just to make sure everything works correctly.
#Using original data:

for i in range(3,11):
  print(f"DT with {i} levels:")
  do_DT(df,i,'Target',5)
```

```python

print(f"MN_NB:")
do_mnNB(df_no_negs,'Target',5)

print(f"GA_NB:")
do_gaNB(df,'Target',5)

print(f"LR:")
do_LR(df,'Target',5)

print(f"KNN:")
do_KNN(df,'Target',5)

print(f"RF:")
do_RF(df,'Target',5)

print(f"SVM:")
do_SVM(df,'Target',5)

print(f"XGB:")
do_XGB(df,'Target',5)
```

```
DT with 3 levels:
Successfuly trained the decision tree for 3 levels...
Confusion Matrix
[[63 16]
 [12 93]]

TP:  93 , FP:  16 , TN:  63 , FN: 12
TPR:  0.8532110091743119 , FPR:  0.20253164556962025 FNR:  0.11428571428571428
              precision    recall  f1-score   support

           0      0.840     0.797     0.818        79
           1      0.853     0.886     0.869       105

    accuracy                          0.848       184
   macro avg      0.847     0.842     0.844       184
weighted avg      0.848     0.848     0.847       184

Accuracy is: 0.8478260869565217
F1 Weighted is: 0.8472719884747516

DT with 4 levels:
Successfuly trained the decision tree for 4 levels...
Confusion Matrix
[[ 53  18]
 [ 11 102]]

TP:  102 , FP:  18 , TN:  53 , FN: 11
TPR:  0.85 , FPR:  0.2535211267605634 FNR:  0.09734513274336283
              precision    recall  f1-score   support

           0      0.828     0.746     0.785        71
           1      0.850     0.903     0.876       113
```

```
    accuracy                           0.842       184
   macro avg      0.839     0.825     0.830       184
weighted avg      0.842     0.842     0.841       184

Accuracy is: 0.842391304347826
F1 Weighted is: 0.8406726655746996

DT with 5 levels:
Successfuly trained the decision tree for 5 levels...
Confusion Matrix
[[64 15]
 [21 84]]

TP:  84 , FP:  15 , TN:  64 , FN: 21
TPR:  0.8484848484848485 , FPR:  0.189873417721519 FNR:  0.2
              precision    recall  f1-score   support

           0      0.753     0.810     0.780        79
           1      0.848     0.800     0.824       105

    accuracy                           0.804       184
   macro avg      0.801     0.805     0.802       184
weighted avg      0.807     0.804     0.805       184

Accuracy is: 0.8043478260869565
```

### Validation:

```
#Decided against cross-fold validation, and instead using an iterative approach.
#In lieu of cross-fold validation, 100 random 80/20 splits of the data-set were used, and
#average TPR, average TNR, and average FNR were calculated over the 100 iterations.
#
#Once the initial validation is performed, the iterative approach will be extended to 1000

#The reason for this approach versus the more common, professional, cross-fold validation
#This approach is possible due to fact the data-set is relatively small, making each itera
#This approach would not be possible using very large data-sets.


#Original Cross-fold validation code (no longer used)
#cv = KFold(n_splits=10, random_state=1, shuffle=True)
#lr = LogisticRegression(max_iter=2000)
#nb = GaussianNB()
#rf = RandomForestClassifier()
#xgb = XGBClassifier()
#
#X = df.drop('Target',axis=1)
#Y = df['Target']
#
#LR_acc = cross_val_score(lr, X, Y, scoring='accuracy', cv=cv, n_jobs=-1)
#NB_acc = cross_val_score(nb, X, Y, scoring='accuracy', cv=cv, n_jobs=-1)
```

```
#RFTree_acc = cross_val_score(rf, X, Y, scoring='accuracy', cv=cv, n_jobs=-1)
#XGB_acc = cross_val_score(xgb, X, Y, scoring='accuracy', cv=cv, n_jobs=-1)
#
#
#print('Accuracy after Cross-Val - Logistic Regression(LR): %.4f (%.4f)' % (mean(LR_acc),
#print('Accuracy after Cross-Val - Gaussian Naive Bayes(gaNB): %.4f (%.4f)' % (mean(NB_acc
#print('Accuracy after Cross-Val - Random Forest(RF): %.4f (%.4f)' % (mean(RFTree_acc), st
#print('Accuracy after Cross-Val - XGBoost(XGB): %.4f (%.4f)' % (mean(XGB_acc), stdev(XGB_
#
#


#iterative approach methods:
def getTestAccuracy(x):
  value = results[f"TEST{x}#AVERAGE_ACCURACY"]
  return(value)

def getTestF1(x):
  value = results[f"TEST{x}#AVERAGE_F1_WEIGHTED"]
  return(value)

def getTestFNR(x):
  value = results[f"TEST{x}#AVERAGE_FNR"]
  return(value)

def displayResult(x):
  name = results[f"TEST{x}#NAME"]
  df_name = results[f"TEST{x}#DFNAME"]
  accuracy = results[f"TEST{x}#AVERAGE_ACCURACY"]*100
  f1_weighted = results[f"TEST{x}#AVERAGE_F1_WEIGHTED"]*100
  tpr = results[f"TEST{x}#AVERAGE_TPR"]*100
  fpr = results[f"TEST{x}#AVERAGE_FPR"]*100
  fnr = results[f"TEST{x}#AVERAGE_FNR"]*100

  print(f"Using ML Model: {name} and {df_name}:")
  print(f"Average Accuracy is {accuracy:.2f}%, Average F1(weighted) is {f1_weighted:.2f}%,
  print("")

def iterativeValidation(test,df,iterations,verbose=0):
  if verbose > 0:
    print("")
    print("")
    print("")

  df_name = df.name

  if test == "MN_NB":
    test_name = "Multinomial Naive Bayes(MN-NB)"
  elif test == "GA_NB":
    test_name = "Gaussian Naive Bayes(GA-NB)"
  elif test == "LR":
```

```
    test_name = "Logistic Regression(LR)"
  elif test == "SVM":
    test_name = "Support Vector Machines(SVM)"
  elif test == "KNN":
    test_name = "K Nearest Neighbours(KNN)"
  elif test == "RF":
    test_name = "Random Forest(RF)"
  elif test == "RF_TUNED":
    test_name = "Random Forest(RF) Tuned"
  elif test == "XGB":
    test_name = "XG Boost(XGB)"
  elif test == "XGB_TUNED":
    test_name = "XG Boost(XGB) Tuned"
  elif test == "DT_3":
    test_name = "Decision Tree: 3 levels"
  elif test == "DT_4":
    test_name = "Decision Tree: 4 levels"
  elif test == "DT_5":
    test_name = "Decision Tree: 5 levels"
  elif test == "DT_6":
    test_name = "Decision Tree: 6 levels"
  elif test == "DT_7":
    test_name = "Decision Tree: 7 levels"
  elif test == "DT_8":
    test_name = "Decision Tree: 8 levels"
  elif test == "DT_9":
    test_name = "Decision Tree: 9 levels"
  elif test == "DT_10":
    test_name = "Decision Tree: 10 levels"

  if verbose > 0:
    print(f"Performing {test_name} Analysis:")
  test_num=results["NEXT_TEST"]

  accuracy_sum = 0
  f1_weighted_sum = 0
  tpr_sum = 0
  fpr_sum = 0
  fnr_sum = 0

  for n in range(iterations):
    if test == "MN_NB":
      result = do_mnNB(df,'Target',0)
    elif test == "GA_NB":
      result = do_gaNB(df,'Target',0)
    elif test == "LR":
      result = do_LR(df,'Target',0)
    elif test == "SVM":
      result = do_SVM(df,'Target',0)
    elif test == "KNN":
      result = do_KNN(df,'Target',0)
```

```
    elif test == "RF_TUNED":
      result = do_RF(df,'Target',0)
    elif test == "RF":
      result = do_RF_tuned(df,'Target',0)
    elif test == "XGB":
      result = do_XGB(df,'Target',0)
    elif test == "XGB_TUNED":
      result = do_XGB_tuned(df,'Target',0)
    elif test == "DT_3":
      result = do_DT(df,3,'Target',0)
    elif test == "DT_4":
      result = do_DT(df,4,'Target',0)
    elif test == "DT_5":
      result = do_DT(df,5,'Target',0)
    elif test == "DT_6":
      result = do_DT(df,6,'Target',0)
    elif test == "DT_7":
      result = do_DT(df,7,'Target',0)
    elif test == "DT_8":
      result = do_DT(df,8,'Target',0)
    elif test == "DT_9":
      result = do_DT(df,9,'Target',0)
    elif test == "DT_10":
      result = do_DT(df,10,'Target',0)

    accuracy_sum += result[0]
    f1_weighted_sum += result[1]
    tpr_sum += result[2]
    fpr_sum += result[3]
    fnr_sum += result[4]

  results[f"TEST{test_num}#NAME"] = test_name
  results[f"TEST{test_num}#DFNAME"] = df.name
  results[f"TEST{test_num}#AVERAGE_ACCURACY"] = accuracy_sum/iterations
  results[f"TEST{test_num}#AVERAGE_F1_WEIGHTED"] = f1_weighted_sum/iterations
  results[f"TEST{test_num}#AVERAGE_TPR"] = tpr_sum/iterations
  results[f"TEST{test_num}#AVERAGE_FPR"] = fpr_sum/iterations
  results[f"TEST{test_num}#AVERAGE_FNR"] = fnr_sum/iterations
  results[f"LAST_TEST"] = test_num
  results[f"NEXT_TEST"] += 1


  name = results[f"TEST{test_num}#NAME"]
  df_name = results[f"TEST{test_num}#DFNAME"]
  accuracy = results[f"TEST{test_num}#AVERAGE_ACCURACY"]*100
  f1_weighted = results[f"TEST{test_num}#AVERAGE_F1_WEIGHTED"]*100
  tpr = results[f"TEST{test_num}#AVERAGE_TPR"]*100
  fpr = results[f"TEST{test_num}#AVERAGE_FPR"]*100
  fnr = results[f"TEST{test_num}#AVERAGE_FNR"]*100

  if verbose > 0:
```

```
    print(f"Using Data-Set: {df_name}:")
    print(f"Average Accuracy is {accuracy:.2f}%, Average F1(weighted) is {f1_weighted:.2f}
    print("")
    print("")
```

### Initial Validation:

```
#Initial Validation tests. (100 iterations, averaged, all models, all dataset combinations
iterations = 100
results = {}
results["NEXT_TEST"] = 0

iterativeValidation("DT_3",df,iterations)
iterativeValidation("DT_4",df,iterations)
iterativeValidation("DT_5",df,iterations)
iterativeValidation("DT_6",df,iterations)
iterativeValidation("DT_7",df,iterations)
iterativeValidation("DT_8",df,iterations)
iterativeValidation("DT_9",df,iterations)
iterativeValidation("DT_10",df,iterations)

iterativeValidation("MN_NB",df_no_negs,iterations)
iterativeValidation("GA_NB",df,iterations)
iterativeValidation("LR",df,iterations)
iterativeValidation("KNN",df,iterations)
iterativeValidation("SVM",df,iterations)
iterativeValidation("RF",df,iterations)
iterativeValidation("XGB",df,iterations)

iterativeValidation("DT_3",df_onehot,iterations)
iterativeValidation("DT_4",df_onehot,iterations)
iterativeValidation("DT_5",df_onehot,iterations)
iterativeValidation("DT_6",df_onehot,iterations)
iterativeValidation("DT_7",df_onehot,iterations)
iterativeValidation("DT_8",df_onehot,iterations)
iterativeValidation("DT_9",df_onehot,iterations)
iterativeValidation("DT_10",df_onehot,iterations)

iterativeValidation("MN_NB",df_onehot_no_negs,iterations)
iterativeValidation("GA_NB",df_onehot,iterations)
iterativeValidation("LR",df_onehot,iterations)
iterativeValidation("KNN",df_onehot,iterations)
iterativeValidation("SVM",df_onehot,iterations)
iterativeValidation("RF",df_onehot,iterations)
iterativeValidation("XGB",df_onehot,iterations)


iterativeValidation("DT_3",df_outliers_addressed,iterations)
iterativeValidation("DT_4",df_outliers_addressed,iterations)
iterativeValidation("DT_5",df_outliers_addressed,iterations)
iterativeValidation("DT_6",df_outliers_addressed,iterations)
```

```
iterativeValidation("DT_6",df_outliers_addressed,iterations)
iterativeValidation("DT_7",df_outliers_addressed,iterations)
iterativeValidation("DT_8",df_outliers_addressed,iterations)
iterativeValidation("DT_9",df_outliers_addressed,iterations)
iterativeValidation("DT_10",df_outliers_addressed,iterations)

iterativeValidation("MN_NB",df_no_negs,iterations)
iterativeValidation("GA_NB",df_outliers_addressed,iterations)
iterativeValidation("LR",df_outliers_addressed,iterations)
iterativeValidation("KNN",df_outliers_addressed,iterations)
iterativeValidation("SVM",df_outliers_addressed,iterations)
iterativeValidation("RF",df_outliers_addressed,iterations)
iterativeValidation("XGB",df_outliers_addressed,iterations)

iterativeValidation("DT_3",df_normalized,iterations)
iterativeValidation("DT_4",df_normalized,iterations)
iterativeValidation("DT_5",df_normalized,iterations)
iterativeValidation("DT_6",df_normalized,iterations)
iterativeValidation("DT_7",df_normalized,iterations)
iterativeValidation("DT_8",df_normalized,iterations)
iterativeValidation("DT_9",df_normalized,iterations)
iterativeValidation("DT_10",df_normalized,iterations)

iterativeValidation("MN_NB",df_normalized,iterations)
iterativeValidation("GA_NB",df_normalized,iterations)
iterativeValidation("LR",df_normalized,iterations)
iterativeValidation("KNN",df_normalized,iterations)
iterativeValidation("SVM",df_normalized,iterations)
iterativeValidation("RF",df_normalized,iterations)
iterativeValidation("XGB",df_normalized,iterations)
```

```
#Collate initial results
range_limit = min(10,results[f"LAST_TEST"]) #Top 10 results desired.

results_list = list(range(0,results[f"LAST_TEST"]+1))
results_list.sort(key=getTestAccuracy, reverse=True)

print("Results of ML Models: (sorted by accuracy)")
print("")
for i in results_list[0:range_limit]:  #Count starts at zero, so dont need to add one.
  displayResult(i)
print("")
print("")


results_list = list(range(0,results[f"LAST_TEST"]+1))
results_list.sort(key=getTestFNR, reverse=False)

print("Results of ML Models: (sorted by False Negative Rate(FNR))")
print("")
for i in results_list[0:range_limit]:  #Count starts at zero, so dont need to add one.
```

```
  displayResult(i)
print("")
print("")



#View these results again, this time just the top 5 using each sort method:
range_limit = min(5,results[f"LAST_TEST"]) #Top 5 results desired.

results_list = list(range(0,results[f"LAST_TEST"]+1))
results_list.sort(key=getTestAccuracy, reverse=True)

print("Results of ML Models: (sorted by accuracy) (top 5)")
print("")
for i in results_list[0:range_limit]:  #Count starts at zero, so dont need to add one.
  displayResult(i)
print("")
print("")


results_list = list(range(0,results[f"LAST_TEST"]+1))
results_list.sort(key=getTestFNR, reverse=False)

print("Results of ML Models: (sorted by False Negative Rate(FNR)) (top 5)")
print("")
for i in results_list[0:range_limit]:  #Count starts at zero, so dont need to add one.
  displayResult(i)
print("")
print("")


```

```
Using ML Model: Random Forest(RF) and Normalized Data-Set:
Average Accuracy is 86.58%, Average F1(weighted) is 86.55%, Average TPR is 86.66%, Av

Using ML Model: XG Boost(XGB) and Original Data-Set:
Average Accuracy is 87.02%, Average F1(weighted) is 86.99%, Average TPR is 87.36%, Av

Using ML Model: XG Boost(XGB) and Original Data-Set, onehot:
Average Accuracy is 86.93%, Average F1(weighted) is 86.91%, Average TPR is 87.29%, Av

Using ML Model: XG Boost(XGB) and Normalized Data-Set:
Average Accuracy is 87.11%, Average F1(weighted) is 87.10%, Average TPR is 87.98%, Av

Using ML Model: Logistic Regression(LR) and Original Data-Set, onehot:
Average Accuracy is 86.48%, Average F1(weighted) is 86.46%, Average TPR is 86.82%, Av

Using ML Model: Support Vector Machines(SVM) and Normalized Data-Set:
Average Accuracy is 85.15%, Average F1(weighted) is 85.09%, Average TPR is 85.06%, Av



Results of ML Models: (sorted by accuracy) (top 5)
```

```
Using ML Model: Random Forest(RF) and Original Data-Set, onehot:
Average Accuracy is 87.15%, Average F1(weighted) is 87.11%, Average TPR is 86.62%, Av

Using ML Model: XG Boost(XGB) and Normalized Data-Set:
Average Accuracy is 87.11%, Average F1(weighted) is 87.10%, Average TPR is 87.98%, Av

Using ML Model: XG Boost(XGB) and Outliers Addressed Data-Set:
Average Accuracy is 87.11%, Average F1(weighted) is 87.08%, Average TPR is 87.51%, Av

Using ML Model: XG Boost(XGB) and Original Data-Set:
Average Accuracy is 87.02%, Average F1(weighted) is 86.99%, Average TPR is 87.36%, Av

Using ML Model: XG Boost(XGB) and Original Data-Set, onehot:
Average Accuracy is 86.93%, Average F1(weighted) is 86.91%, Average TPR is 87.29%, Av



Results of ML Models: (sorted by False Negative Rate(FNR)) (top 5)

Using ML Model: Random Forest(RF) and Original Data-Set, onehot:
Average Accuracy is 87.15%, Average F1(weighted) is 87.11%, Average TPR is 86.62%, Av

Using ML Model: Random Forest(RF) and Original Data-Set:
Average Accuracy is 86.87%, Average F1(weighted) is 86.84%, Average TPR is 86.85%, Av

Using ML Model: Random Forest(RF) and Outliers Addressed Data-Set:
Average Accuracy is 86.27%, Average F1(weighted) is 86.22%, Average TPR is 86.00%, Av

Using ML Model: XG Boost(XGB) and Outliers Addressed Data-Set:
Average Accuracy is 87.11%, Average F1(weighted) is 87.08%, Average TPR is 87.51%, Av

Using ML Model: Random Forest(RF) and Normalized Data-Set:
Average Accuracy is 86.58%, Average F1(weighted) is 86.55%, Average TPR is 86.66%, Av
```

### Secondary Validation:

```
#Secondary Validation, 1000 iterations, on top models only)
iterations = 1000
results = {}
results["NEXT_TEST"] = 0

iterativeValidation("XGB",df_normalized,iterations)
iterativeValidation("XGB",df_outliers_addressed,iterations)
iterativeValidation("XGB",df,iterations)
iterativeValidation("XGB",df_onehot,iterations)

iterativeValidation("RF",df_onehot,iterations)
iterativeValidation("RF",df,iterations)
iterativeValidation("RF",df_outliers_addressed,iterations)
```

```
iterativeValidation("RF",df_normalized,iterations)


```

```
#Collate secondary results
range_limit = min(3,results[f"LAST_TEST"]) #Top 3 results desired.

results_list = list(range(0,results[f"LAST_TEST"]+1))
results_list.sort(key=getTestAccuracy, reverse=True)

print("Results of ML Models: (sorted by accuracy)")
print("")
for i in results_list[0:range_limit]:  #Count starts at zero, so dont need to add one.
  displayResult(i)
print("")
print("")


results_list = list(range(0,results[f"LAST_TEST"]+1))
results_list.sort(key=getTestFNR, reverse=False)

print("Results of ML Models: (sorted by False Negative Rate(FNR))")
print("")
for i in results_list[0:range_limit]:  #Count starts at zero, so dont need to add one.
  displayResult(i)
print("")
print("")
```

```
Results of ML Models: (sorted by accuracy)

Using ML Model: XG Boost(XGB) and Original Data-Set, onehot:
Average Accuracy is 87.11%, Average F1(weighted) is 87.08%, Average TPR is 87.22%, Avera

Using ML Model: XG Boost(XGB) and Original Data-Set:
Average Accuracy is 87.08%, Average F1(weighted) is 87.06%, Average TPR is 87.26%, Avera

Using ML Model: Random Forest(RF) and Original Data-Set, onehot:
Average Accuracy is 87.06%, Average F1(weighted) is 87.02%, Average TPR is 86.95%, Avera



Results of ML Models: (sorted by False Negative Rate(FNR))

Using ML Model: Random Forest(RF) and Original Data-Set, onehot:
Average Accuracy is 87.06%, Average F1(weighted) is 87.02%, Average TPR is 86.95%, Avera

Using ML Model: Random Forest(RF) and Outliers Addressed Data-Set:
Average Accuracy is 86.45%, Average F1(weighted) is 86.41%, Average TPR is 86.17%, Avera

Using ML Model: XG Boost(XGB) and Original Data-Set, onehot:
Average Accuracy is 87.11%, Average F1(weighted) is 87.08%, Average TPR is 87.22%, Avera
```

## Hypertuning:

```
rf = RandomForestClassifier()
xgb = XGBClassifier()
feature_names=df_onehot.columns[df_onehot.columns != "Target"]
# 80% training and 20% test
X_train, X_test, Y_train, Y_test = train_test_split(df_onehot.loc[:, feature_names], df_on


params_rf = {'n_estimators':[100,200,300,400,500], 'min_samples_leaf':[5, 10, 15, 20, 25,
grid_rf = GridSearchCV(rf, param_grid=params_rf, cv=10)
grid_rf.fit(X_train, Y_train)
print("Hyper-Tuned Parameters for Random Forest:", grid_rf.best_params_)


params_xgb = {'n_estimators': [100,200,300,400,500,600,700,800,900,1000], 'learning_rate':
rs_xgb =  RandomizedSearchCV(xgb, param_distributions=params_xgb, cv=10)
rs_xgb.fit(X_train, Y_train)
print("Hyper-Tuned Parameters for XGBoost:", rs_xgb.best_params_)
```

```
Hyper-Tuned Parameters for Random Forest: {'min_samples_leaf': 5, 'n_estimators': 400}
Hyper-Tuned Parameters for XGBoost: {'n_estimators': 500, 'learning_rate': 0.1}
```

```
#Tertiary Validation(after hypertuning), 1000 iterations, on top models only)
iterations = 1000
results = {}
results["NEXT_TEST"] = 0

iterativeValidation("XGB_TUNED",df_normalized,iterations)
iterativeValidation("XGB_TUNED",df_outliers_addressed,iterations)
iterativeValidation("XGB_TUNED",df,iterations)
iterativeValidation("XGB_TUNED",df_onehot,iterations)

iterativeValidation("RF_TUNED",df_onehot,iterations)
iterativeValidation("RF_TUNED",df,iterations)
iterativeValidation("RF_TUNED",df_outliers_addressed,iterations)
iterativeValidation("RF_TUNED",df_normalized,iterations)
```

## Tertiary Validation (After Hypertuning):

```
#Collate tertiary results
range_limit = min(3,results[f"LAST_TEST"]) #Top 3 results desired.

results_list = list(range(0,results[f"LAST_TEST"]+1))
```

```
results_list.sort(key=getTestAccuracy, reverse=True)

print("Results of ML Models: (sorted by accuracy)")
print("")
for i in results_list[0:range_limit]:  #Count starts at zero, so dont need to add one.
  displayResult(i)
print("")
print("")


results_list = list(range(0,results[f"LAST_TEST"]+1))
results_list.sort(key=getTestFNR, reverse=False)

print("Results of ML Models: (sorted by False Negative Rate(FNR))")
print("")
for i in results_list[0:range_limit]:  #Count starts at zero, so dont need to add one.
  displayResult(i)
print("")
print("")
```

```
Results of ML Models: (sorted by accuracy)

Using ML Model: Random Forest(RF) Tuned and Original Data-Set, onehot:
Average Accuracy is 87.03%, Average F1(weighted) is 86.99%, Average TPR is 86.82%, Avera

Using ML Model: Random Forest(RF) Tuned and Normalized Data-Set:
Average Accuracy is 86.80%, Average F1(weighted) is 86.77%, Average TPR is 86.87%, Avera

Using ML Model: Random Forest(RF) Tuned and Original Data-Set:
Average Accuracy is 86.62%, Average F1(weighted) is 86.58%, Average TPR is 86.49%, Avera



Results of ML Models: (sorted by False Negative Rate(FNR))

Using ML Model: Random Forest(RF) Tuned and Original Data-Set, onehot:
Average Accuracy is 87.03%, Average F1(weighted) is 86.99%, Average TPR is 86.82%, Avera

Using ML Model: Random Forest(RF) Tuned and Original Data-Set:
Average Accuracy is 86.62%, Average F1(weighted) is 86.58%, Average TPR is 86.49%, Avera

Using ML Model: Random Forest(RF) Tuned and Outliers Addressed Data-Set:
Average Accuracy is 86.33%, Average F1(weighted) is 86.29%, Average TPR is 86.08%, Avera
```

✓ 0s completed at 9:06 PM