# 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
```

### 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_validate
```

```
from sklearn.model_selection import cross_validate
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RandomizedSearchCV, GridSearchCV
import imblearn
from imblearn.over_sampling import SMOTE

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 | Max |
|---|---|---|---|---|---|---|---|---|
| **0** | 40 | 1 | 2 | 140 | 289 | 0 | 0 | |
| **1** | 49 | 0 | 3 | 160 | 180 | 0 | 0 | |
| **2** | 37 | 1 | 2 | 130 | 283 | 0 | 1 | |

## 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 272 duplicate rows in this dataset.

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 918 entries, 0 to 917
Data columns (total 12 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Age                918 non-null    int64
 1   Sex                918 non-null    int64
 2   ChestPainType      918 non-null    int64
 3   RestingBP_s        918 non-null    int64
 4   Cholesterol        918 non-null    int64
 5   FastingBloodSugar  918 non-null    int64
 6   RestingECG         918 non-null    int64
```

```
 7   MaxHeartRate       918 non-null    int64
 8   ExerciseAngina     918 non-null    int64
 9   OldPeak            918 non-null    float64
 10  ST_Slope           918 non-null    int64
 11  Target             918 non-null    int64
dtypes: float64(1), int64(11)
memory usage: 86.2 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 is 1 outlier:

     Age  Sex  ChestPainType  RestingBP_s  Cholesterol  FastingBloodSugar  \
```

```
516   68    1              3          150          195                  1

     RestingECG  MaxHeartRate  ExerciseAngina  OldPeak  ST_Slope  Target
516           0           132               0      0.0         0       1



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()
```

```
0      1
1    395
2    459
3     63
Name: ST_Slope, dtype: int64


<class 'pandas.core.frame.DataFrame'>
Int64Index: 917 entries, 0 to 917
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 |

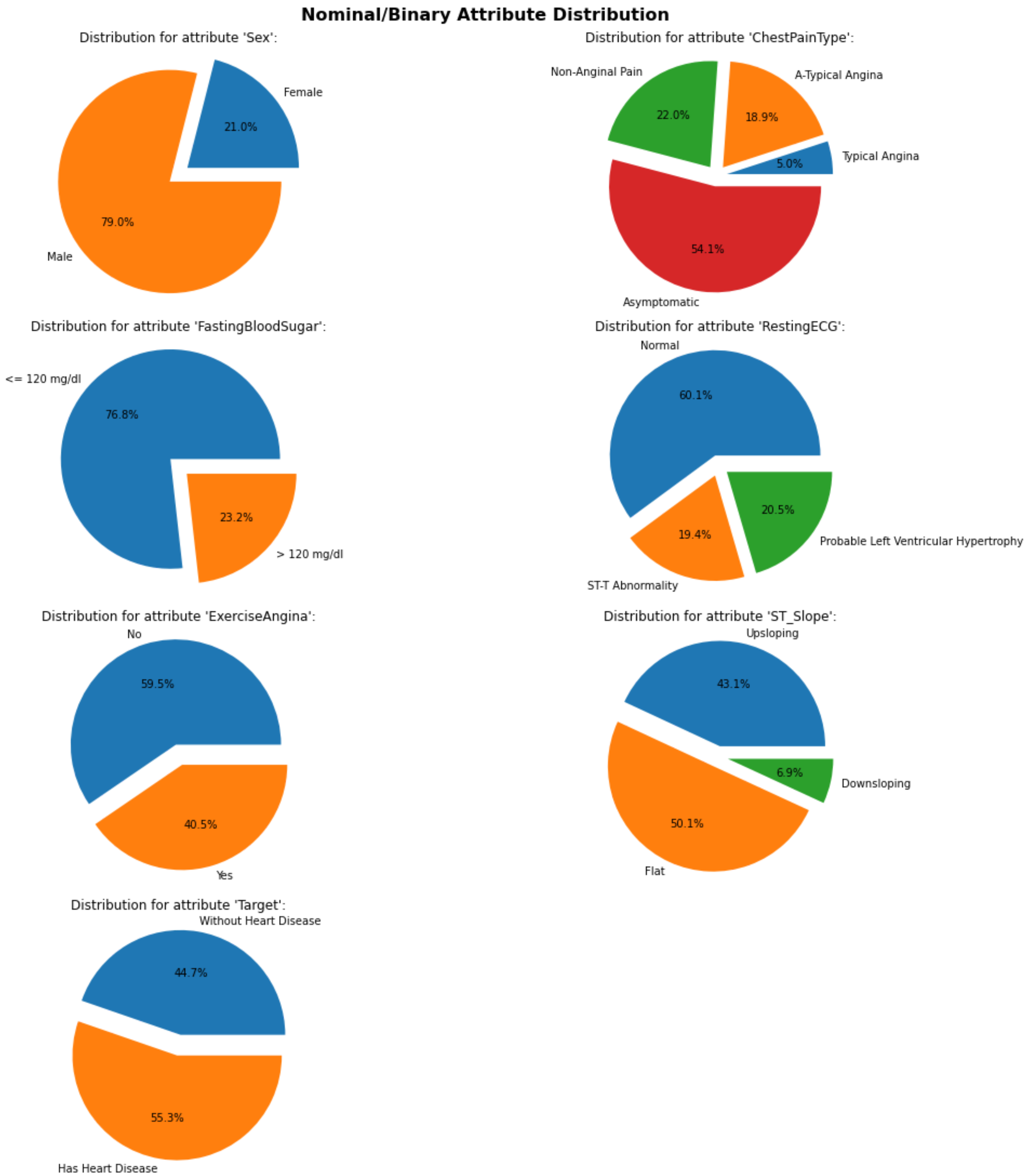
### Visualizing All Attributes:

```
#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
```

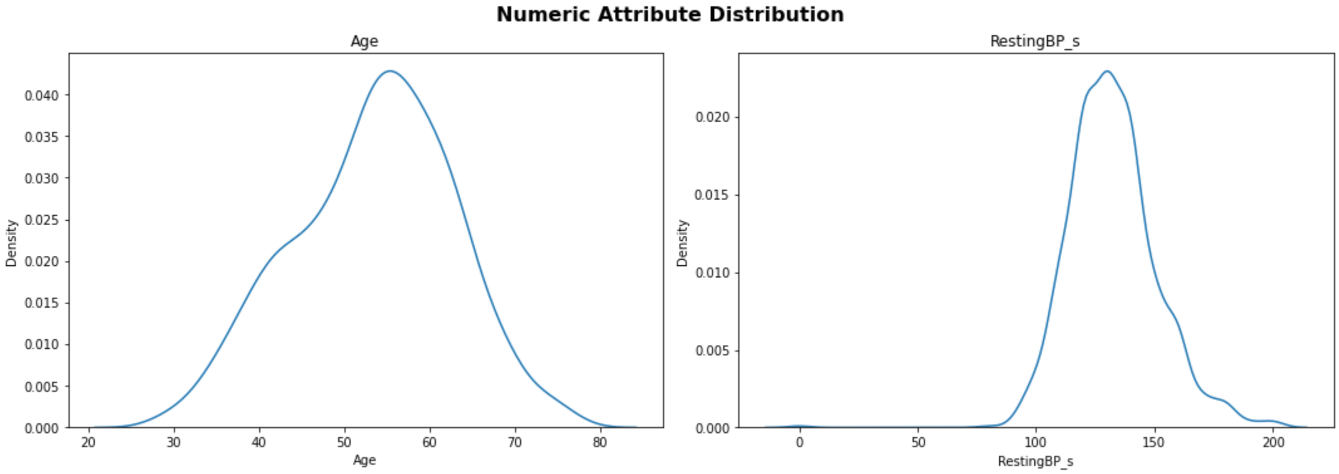
```
#
#Breaking up the dataset into two groups, those with heart disease and those without.
with_heart_disease = df[df['Target'] == 1]
no_heart_disease = df[df['Target'] == 0]

#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')
```



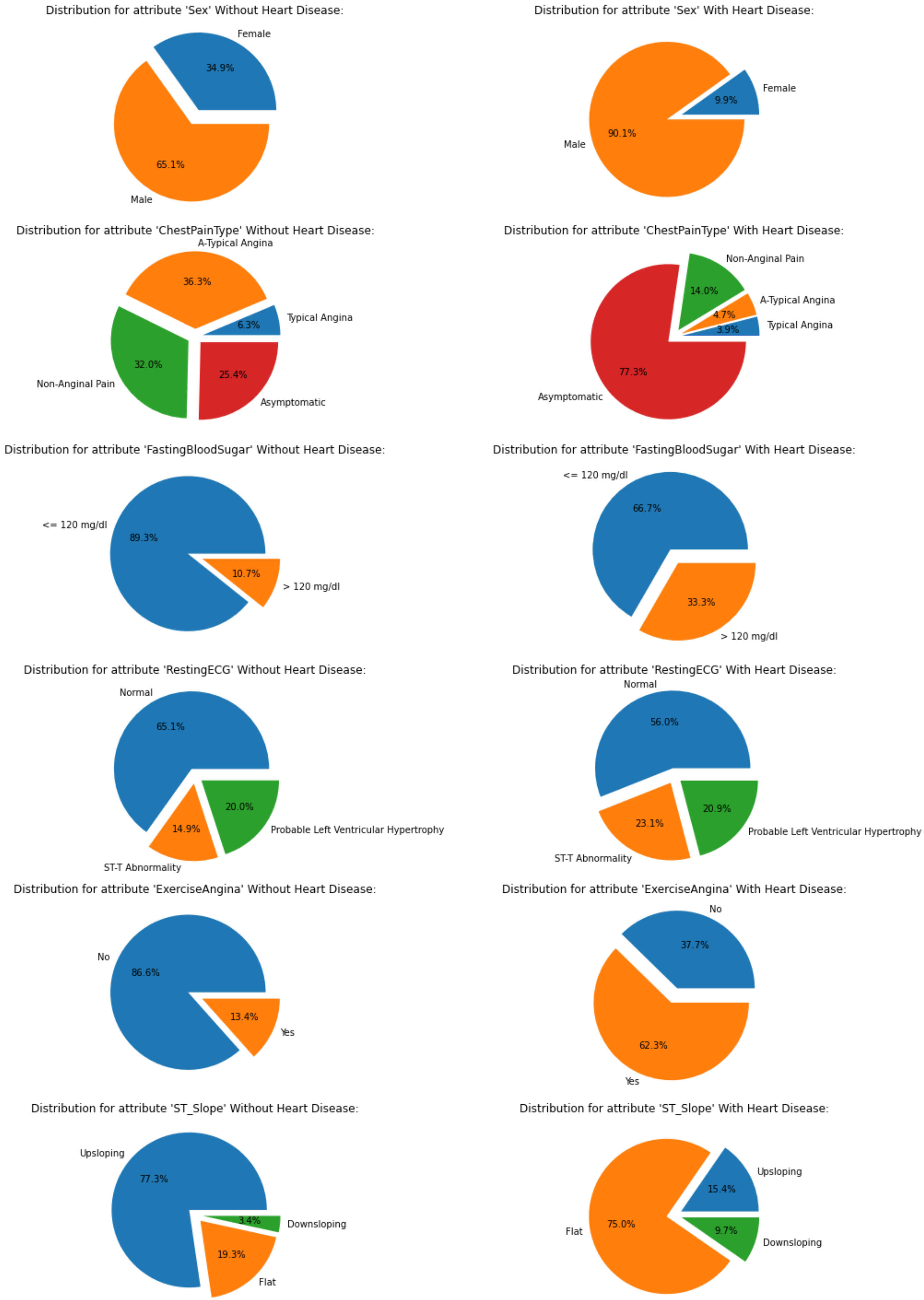

```
#Visualizing the overall distribution of the numeric attributes.
plt.figure(figsize=(15,15))

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')
```

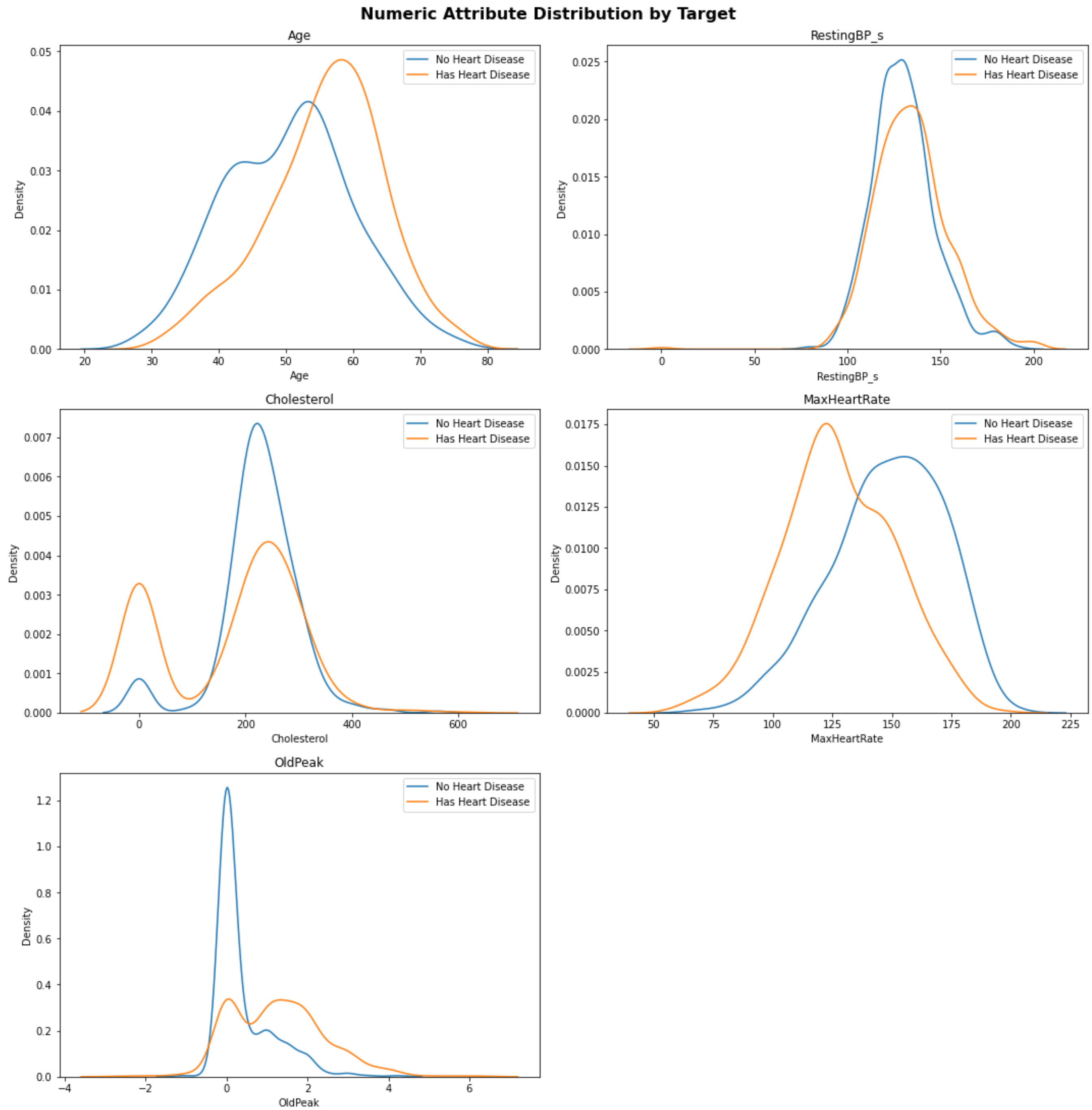



```
#Visualizing the distribution of the categorical attributes, by target
ax=1
plt.figure(figsize=(15,20))
plt.axis('equal')

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**

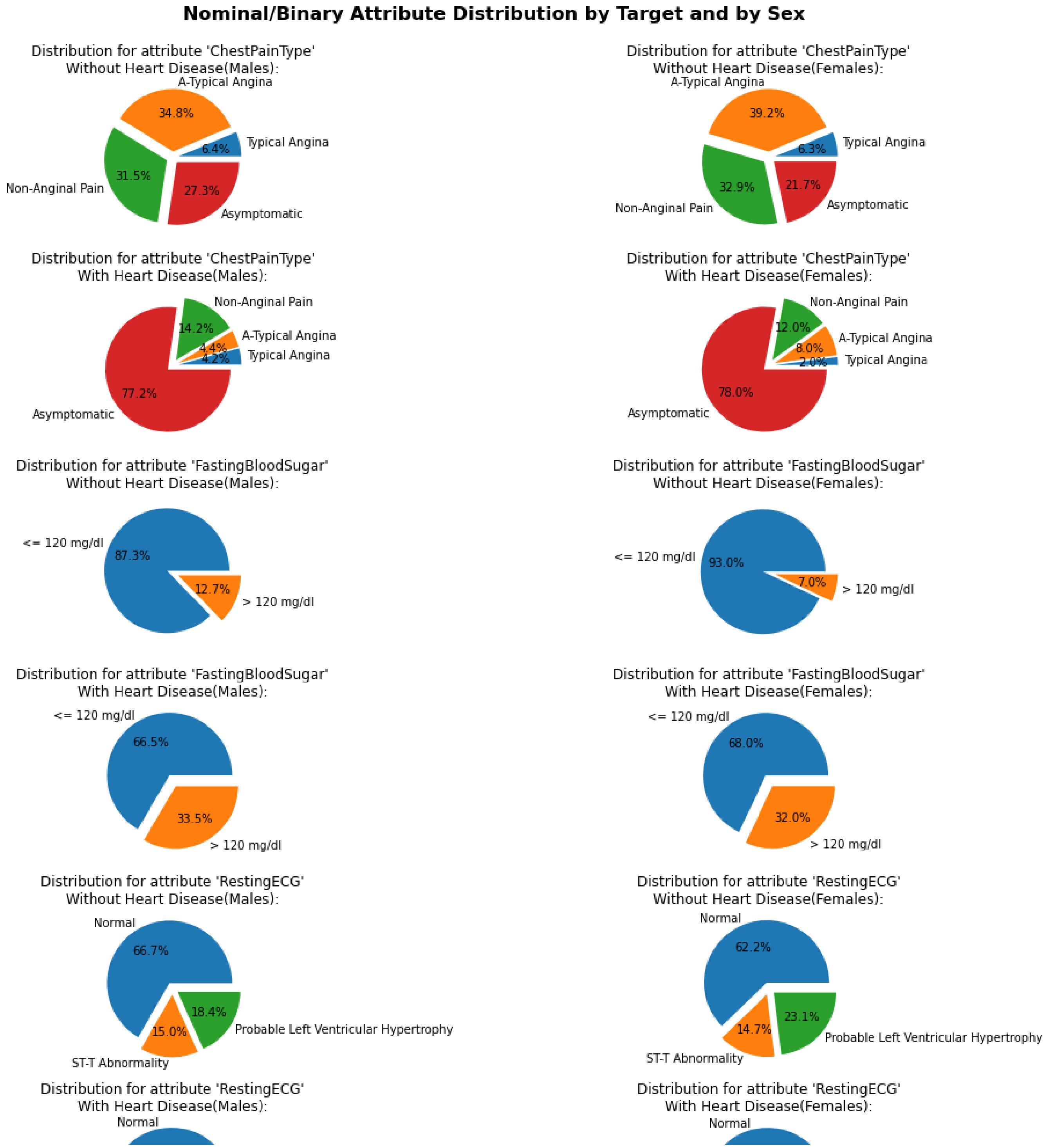


```
#Visualizing the distribution of the numeric attributes by Target:
plt.figure(figsize=(15,15))

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')
```



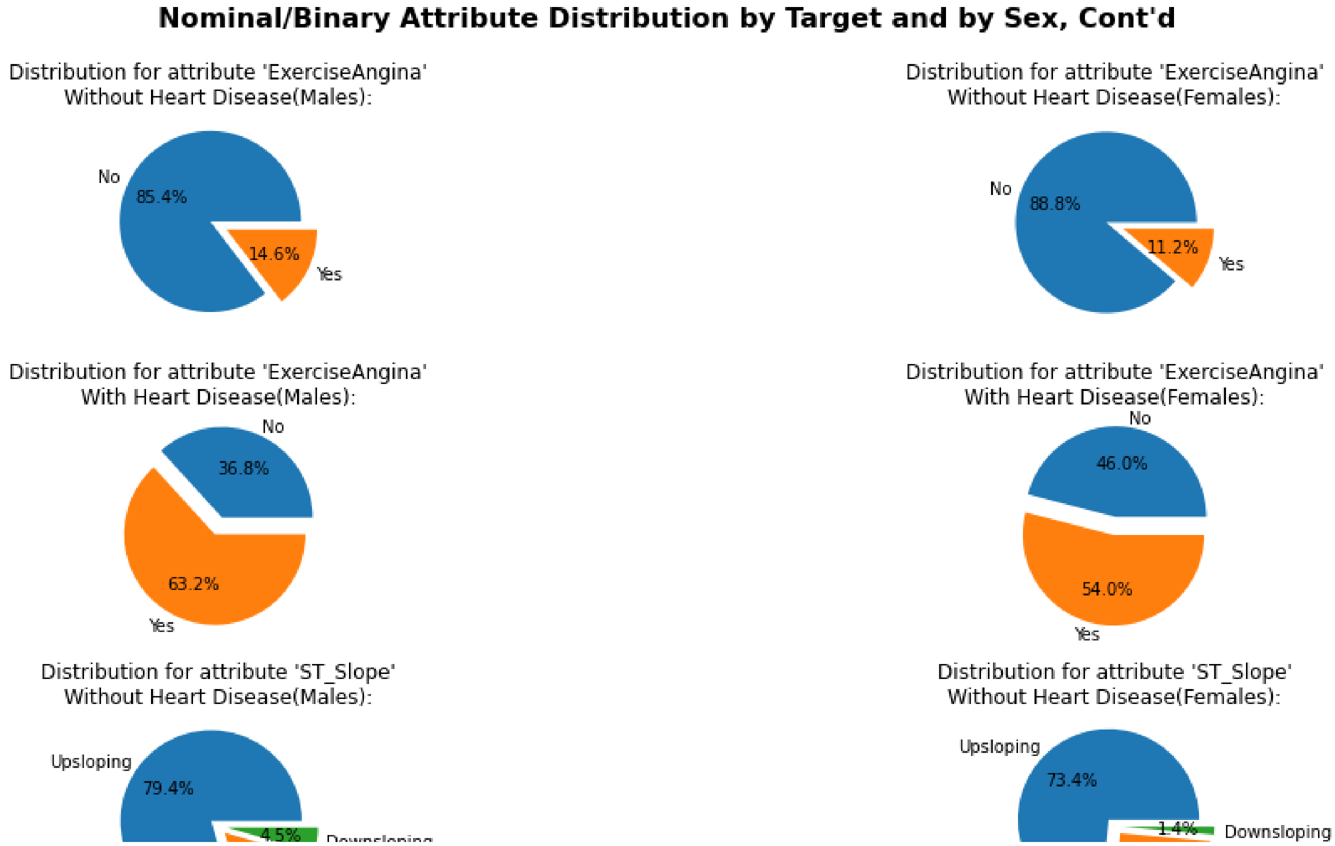

```
#Visualizing the distribution of the categorical attributes, by target and by sex
#For report purposes, breaking this up into two separate pages.
#This one for attributes 'ChestPainType', 'FastingBloodSugar', 'RestingECG'
#
ax=1
plt.figure(figsize=(15,30))
plt.axis('equal')

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')
```

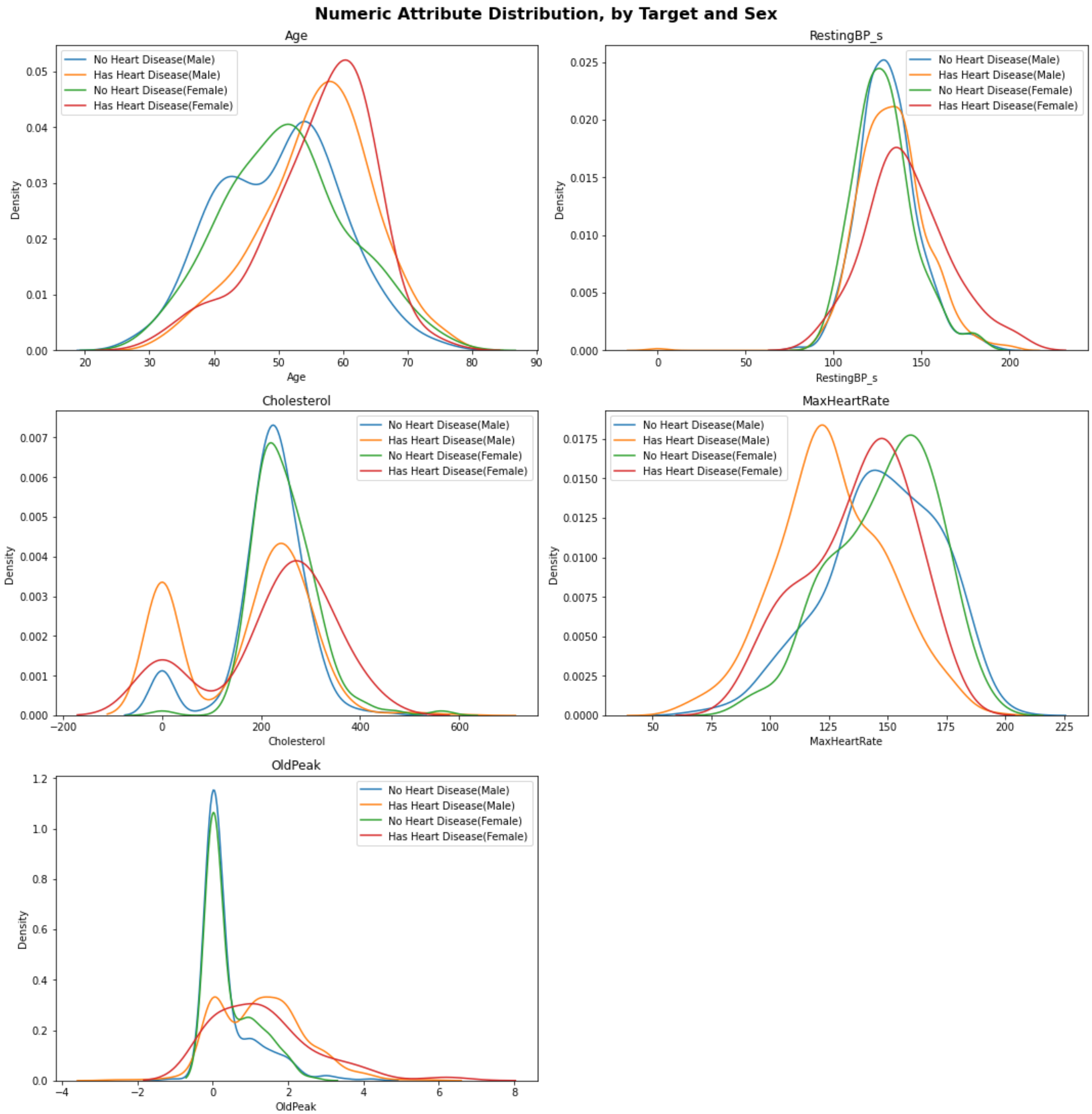



```
#Visualizing the distribution of the categorical attributes, by target and by sex
#For report purposes, breaking this up into two separate pages.
#This one for attributes 'ExerciseAngina','ST_Slope'
#
ax=1
plt.figure(figsize=(15,30))
plt.axis('equal')

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")
```

```
For attribute 'Age': The mean is 53.49509269356598, stdev3 is 28.276802628148687
For 1.5IQR the lower range is 27.5 the upper range is 79.5
For mean +/- 3STDEV the lower range is 25.218290065417293 the upper range is 81.77189


Using 1.5IQR Method:
For attribute 'Age': There are no outliers.




Using mean +/- 3STDEV Method:
For attribute 'Age': There are no outliers.




For attribute 'RestingBP_s': The mean is 132.3773173391494, stdev3 is 55.545341581484
For 1.5IQR the lower range is 90.0 the upper range is 170.0
For mean +/- 3STDEV the lower range is 76.83197575766476 the upper range is 187.92265


Using 1.5IQR Method:
```

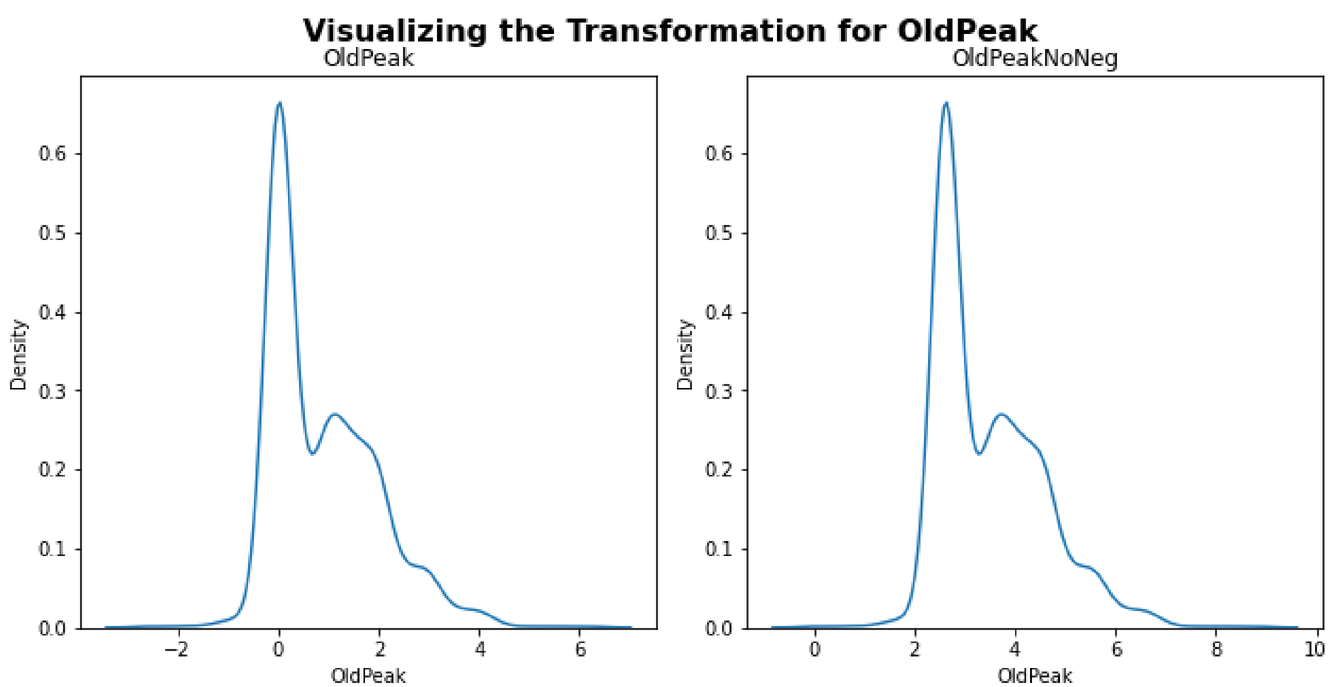
```
For attribute 'RestingBP_s': There are 28 outliers:

     Age  Sex  ChestPainType  RestingBP_s  Cholesterol  FastingBloodSugar  \
109   39    1              2          190          241                  0
123   58    0              2          180          393                  0
189   53    1              4          180          285                  0
190   46    1              4          180          280                  0
241   54    1              4          200          198                  0
274   45    0              2          180          295                  0
275   59    1              3          180          213                  0
278   57    0              4          180          347                  0
314   53    1              4           80            0                  0
365   64    0              4          200            0                  0
372   63    1              4          185            0                  0
399   61    1              3          200            0                  1
411   54    1              4          180            0                  1
423   60    1              3          180            0                  0
449   55    1              3            0            0                  0
475   59    1              4          178            0                  1
550   55    1              4          172          260                  0
585   57    1              2          180          285                  1
592   61    1              4          190          287                  1
673   59    0              4          174          249                  0
702   59    1              1          178          270                  0
725   55    0              4          180          327                  0
732   56    0              4          200          288                  1
759   54    1              2          192          283                  0
774   66    0              4          178          228                  1
780   64    0              4          180          325                  0
855   68    1              3          180          274                  1
880   52    1              3          172          199                  1

     RestingECG  MaxHeartRate  ExerciseAngina  OldPeak  ST_Slope  Target
109           0           106               0      0.0         1       0
123           0           110               1      1.0         2       1
189           1           120               1      1.5         2       1
```

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

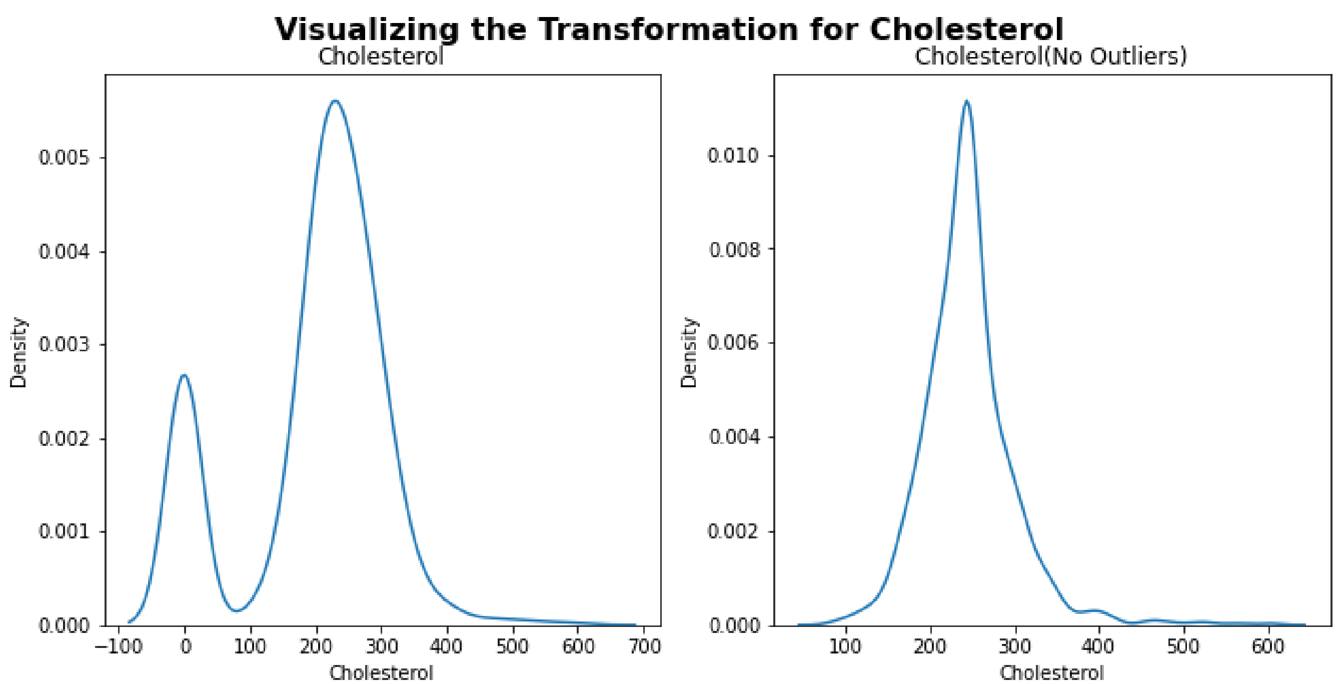
```
#Before we can run multinomial Naive Bayes we must remove any negative numbers in the data
mins = df.min()
print(mins)
print("\n\n")
#There are only negative values for attribute 'OldPeak'
#Applying a simple shift to eliminate any negatives.

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())
```

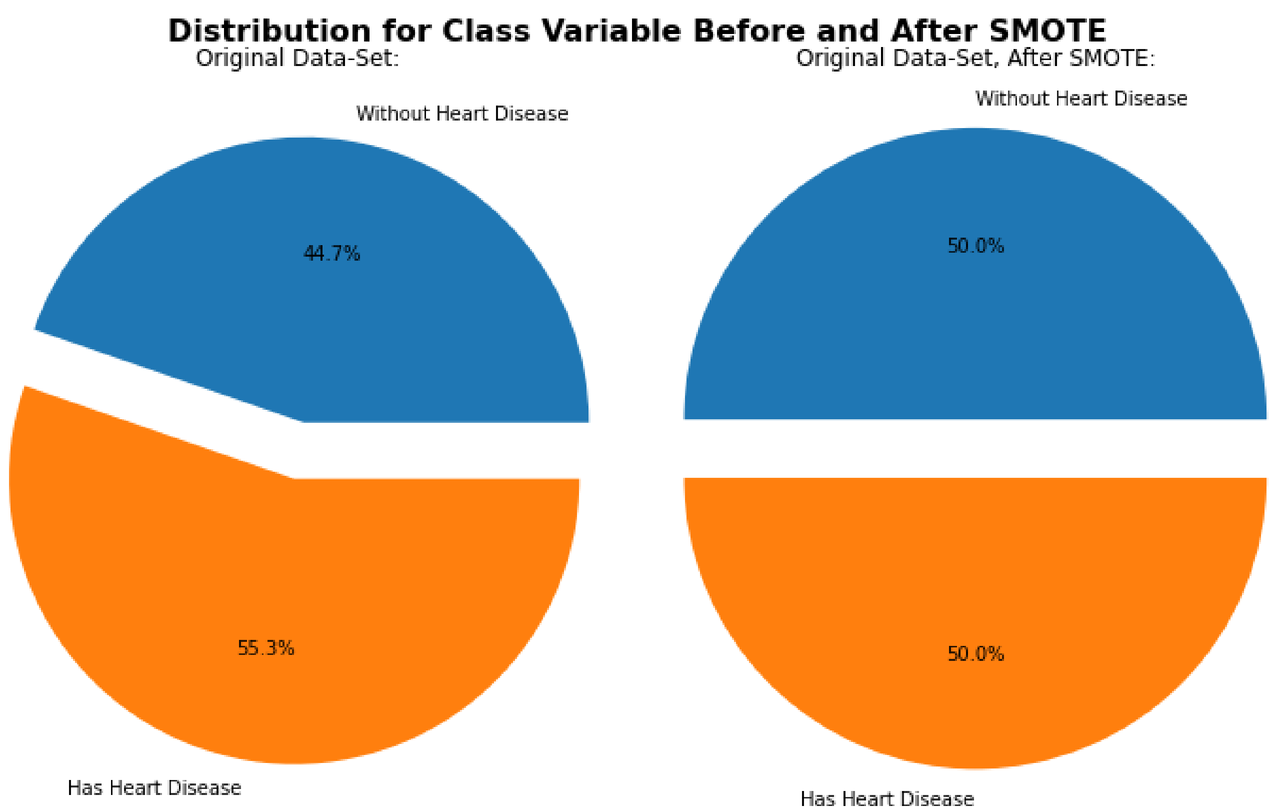
### Create ONE-HOT columns for all categorical attributes:

```
#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, After 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, After ONEHOT"

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

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

```

### Use SMOTE to Balance the Class Variable:

```
# Balance the dataset using SMOTE.

def do_SMOTE(df, classCol="Target"):
  oversample = SMOTE()
  X = df.drop(columns=classCol)
  Y = df[classCol]
  X, Y = oversample.fit_resample(X, Y)
  X[classCol] = Y
  X.name = df.name+", After SMOTE"
  return(X)



df_smote = do_SMOTE(df)
df_no_negs_smote = do_SMOTE(df_no_negs)

df_normalized_smote = do_SMOTE(df_normalized)
df_outliers_addressed_smote = do_SMOTE(df_outliers_addressed)

df_onehot_smote = do_SMOTE(df_onehot)
df_onehot_no_negs_smote = do_SMOTE(df_onehot_no_negs)
df_onehot_outliers_addressed_smote = do_SMOTE(df_onehot_outliers_addressed)
```

```
df_onehot_normalized_smote = do_SMOTE(df_onehot_normalized)


#Visualizing the distribution of the class variable before and after SMOTE for the basic d
ax=1
plt.figure(figsize=(10,6))

for this_df in (df,df_smote):
  plt.subplot(1,2,ax)
  these_labels = labels['Target']
  plt.title(f"{this_df.name}:")
  plt.pie(this_df['Target'].value_counts().sort_index(),
        autopct = '%1.1f%%', labels=these_labels,
        explode=tuple([0.1] * len(these_labels)))
  plt.axis('equal')
  ax+=1

plt.suptitle('Distribution for Class Variable Before and After SMOTE',y=1.01, size = 16, c
plt.tight_layout()
plt.savefig("smote.pdf",dpi=1200, bbox_inches='tight')

```




## 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_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)
```

## 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)

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
[[76 15]
 [13 80]]

TP:  80 , FP:  15 , TN:  76 , FN: 13
TPR:  0.8421052631578947 , FPR:  0.16483516483516483 FNR:  0.13978494623655913
              precision    recall  f1-score   support

           0      0.854     0.835     0.844        91
           1      0.842     0.860     0.851        93

    accuracy                          0.848       184
   macro avg      0.848     0.848     0.848       184
weighted avg      0.848     0.848     0.848       184

Accuracy is: 0.8478260869565217
F1 Weighted is: 0.8477901120361805

DT with 4 levels:
Successfuly trained the decision tree for 4 levels...
Confusion Matrix
[[71 13]
 [11 89]]

TP:  89 , FP:  13 , TN:  71 , FN: 11
TPR:  0.8725490196078431 , FPR:  0.15476190476190477 FNR:  0.11
              precision    recall  f1-score   support

           0      0.866     0.845     0.855        84
           1      0.873     0.890     0.881       100

    accuracy                          0.870       184
   macro avg      0.869     0.868     0.868       184
weighted avg      0.869     0.870     0.869       184

Accuracy is: 0.8695652173913043
F1 Weighted is: 0.8694251824344299

DT with 5 levels:
Successfuly trained the decision tree for 5 levels...
Confusion Matrix
[[65 16]
 [15 88]]

TP:  88 , FP:  16 , TN:  65 , FN: 15
TPR:  0.8461538461538461 , FPR:  0.19753086419753085 FNR:  0.14563106796116504
              precision    recall  f1-score   support

           0      0.812     0.802     0.807        81
```

```
           1      0.846     0.854     0.850       103

    accuracy                          0.832       184
   macro avg      0.829     0.828     0.829       184
weighted avg      0.831     0.832     0.831       184

Accuracy is: 0.8315217391304348
```

### Validation:

```
#Create a subroutine to invoke cross_validate for a given model, and dataset.
#
def doCV(test_num, test_name, model, this_df, folds=5, classCol='Target', verbose=0):
  X = this_df.drop(classCol,axis=1)
  Y = this_df[classCol]

  scoring = {'acc': 'accuracy',
             'rec': 'recall_macro'}
  scores = cross_validate(model, X, Y, scoring=scoring,
                          cv=folds, return_train_score=True)

  ACC = scores['test_acc']
  ACC_mean = ACC.mean() * 100
  ACC_std = ACC.std()

  TPR = scores['test_rec']
  TPR_mean = TPR.mean() * 100
  TPR_std = TPR.std()

  FNR = 1-TPR
  FNR_mean = FNR.mean() * 100
  FNR_std = FNR.std()

  if verbose > 0:
    print(f"For TEST #{test_num}: {test_name:35s} After Cross-Val, Using Data-Set: {this_d

  #Store all results in an easy lookup-table
  results[f"LAST_TEST"] = test_num
  results[f"TEST{test_num}#NAME"] = test_name
  results[f"TEST{test_num}#DFNAME"] = this_df.name
  results[f"TEST{test_num}#DF_ROWS"] = this_df.shape[0]
  results[f"TEST{test_num}#ACCURACY_MEAN"] = ACC_mean
  results[f"TEST{test_num}#ACCURACY_STD"] = ACC_std
  results[f"TEST{test_num}#ACCURACY"] = ACC
  results[f"TEST{test_num}#TPR_MEAN"] = TPR_mean
  results[f"TEST{test_num}#TPR_STD"] = TPR_std
  results[f"TEST{test_num}#FNR_MEAN"] = FNR_mean
  results[f"TEST{test_num}#FNR_STD"] = FNR_std
  results[f"TEST{test_num}#FNR"] = FNR

#Create subroutines to display test results, and obtain accuracy and FNR for a given test.
```

```
def getTestAccuracy(x):
  value = results[f"TEST{x}#ACCURACY_MEAN"]
  return(value)

def getTestF1(x):
  value = results[f"TEST{x}#F1_WEIGHTED_MEAN"]
  return(value)

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

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

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

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


def displayResult(x):
  name = results[f"TEST{x}#NAME"]
  df_name = results[f"TEST{x}#DFNAME"]
  accuracy = results[f"TEST{x}#ACCURACY_MEAN"]
  #f1_weighted = results[f"TEST{x}#F1_WEIGHTED_MEAN"]
  tpr = results[f"TEST{x}#TPR_MEAN"]
  #fpr = results[f"TEST{x}#FPR_MEAN"]
  fnr = results[f"TEST{x}#FNR_MEAN"]

  print(f"Using ML Model: {name} and {df_name}:")
  #print(f"TEST #{x}: Average Accuracy is {accuracy:.2f}%, Average F1(weighted) is {f1_wei
  print(f"TEST #{x}: Average Accuracy is {accuracy:.2f}%, Average TPR is {tpr:.2f}%, Avera
  print("")
```

```
#Initial Models used:
dt3 = tree.DecisionTreeClassifier(max_depth=3,criterion='gini')
dt4 = tree.DecisionTreeClassifier(max_depth=4,criterion='gini')
dt5 = tree.DecisionTreeClassifier(max_depth=5,criterion='gini')
dt6 = tree.DecisionTreeClassifier(max_depth=6,criterion='gini')
dt7 = tree.DecisionTreeClassifier(max_depth=7,criterion='gini')
dt8 = tree.DecisionTreeClassifier(max_depth=8,criterion='gini')
dt9 = tree.DecisionTreeClassifier(max_depth=9,criterion='gini')
dt10 = tree.DecisionTreeClassifier(max_depth=10,criterion='gini')
```

```
dt11 = tree.DecisionTreeClassifier(max_depth=11,criterion='gini')
lr = LogisticRegression(max_iter=2000)
knn = KNeighborsClassifier()
svm = SVC()
mn_nb = MultinomialNB()
ga_nb = GaussianNB()
rf = RandomForestClassifier()
xgb = XGBClassifier()
```

### Initial Validation:

```
results = {}
test_num = 0
folds = 10
classCol = "Target"
df_list = [df,df_normalized,df_onehot,df_outliers_addressed,df_smote,df_normalized_smote,d

for test in ["DT_3","DT_4","DT_5","DT_6","DT_7","DT_8","DT_9","DT_10","DT_11","GA_NB","LR"
  if test == "GA_NB":
    model = ga_nb
    test_name = "Gaussian Naive Bayes(GA-NB)"
  elif test == "LR":
    model = lr
    test_name = "Logistic Regression(LR)"
  elif test == "SVM":
    model = svm
    test_name = "Support Vector Machines(SVM)"
  elif test == "KNN":
    model = knn
    test_name = "K Nearest Neighbours(KNN)"
  elif test == "RF":
    model = rf
    test_name = "Random Forest(RF)"
  elif test == "XGB":
    model = xgb
    test_name = "XG Boost(XGB)"
  elif test == "DT_3":
    model = dt3
    test_name = "Decision Tree: 3 levels"
  elif test == "DT_4":
    model = dt4
    test_name = "Decision Tree: 4 levels"
  elif test == "DT_5":
    model = dt5
    test_name = "Decision Tree: 5 levels"
  elif test == "DT_6":
    model = dt6
    test_name = "Decision Tree: 6 levels"
  elif test == "DT_7":
```

```
    model = dt7
    test_name = "Decision Tree: 7 levels"
  elif test == "DT_8":
    model = dt8
    test_name = "Decision Tree: 8 levels"
  elif test == "DT_9":
    model = dt9
    test_name = "Decision Tree: 9 levels"
  elif test == "DT_10":
    model = dt10
    test_name = "Decision Tree: 10 levels"
  elif test == "DT_11":
    model = dt11
    test_name = "Decision Tree: 11 levels"

  for idx,df_not_used in enumerate(df_list):
    this_df = df_list[idx]
    doCV(test_num,test_name,model,this_df,folds,classCol,1)
    test_num += 1
```

```
For TEST #0: Decision Tree: 3 levels          After Cross-Val, Using Data-Set: Ori
For TEST #1: Decision Tree: 3 levels          After Cross-Val, Using Data-Set: Nor
For TEST #2: Decision Tree: 3 levels          After Cross-Val, Using Data-Set: Ori
For TEST #3: Decision Tree: 3 levels          After Cross-Val, Using Data-Set: Out
For TEST #4: Decision Tree: 3 levels          After Cross-Val, Using Data-Set: Ori
For TEST #5: Decision Tree: 3 levels          After Cross-Val, Using Data-Set: Nor
For TEST #6: Decision Tree: 3 levels          After Cross-Val, Using Data-Set: Ori
For TEST #7: Decision Tree: 3 levels          After Cross-Val, Using Data-Set: Out
For TEST #8: Decision Tree: 4 levels          After Cross-Val, Using Data-Set: Ori
For TEST #9: Decision Tree: 4 levels          After Cross-Val, Using Data-Set: Nor
For TEST #10: Decision Tree: 4 levels          After Cross-Val, Using Data-Set: Or
For TEST #11: Decision Tree: 4 levels          After Cross-Val, Using Data-Set: Ou
For TEST #12: Decision Tree: 4 levels          After Cross-Val, Using Data-Set: Or
For TEST #13: Decision Tree: 4 levels          After Cross-Val, Using Data-Set: No
For TEST #14: Decision Tree: 4 levels          After Cross-Val, Using Data-Set: Or
For TEST #15: Decision Tree: 4 levels          After Cross-Val, Using Data-Set: Ou
For TEST #16: Decision Tree: 5 levels          After Cross-Val, Using Data-Set: Or
For TEST #17: Decision Tree: 5 levels          After Cross-Val, Using Data-Set: No
For TEST #18: Decision Tree: 5 levels          After Cross-Val, Using Data-Set: Or
For TEST #19: Decision Tree: 5 levels          After Cross-Val, Using Data-Set: Ou
For TEST #20: Decision Tree: 5 levels          After Cross-Val, Using Data-Set: Or
For TEST #21: Decision Tree: 5 levels          After Cross-Val, Using Data-Set: No
For TEST #22: Decision Tree: 5 levels          After Cross-Val, Using Data-Set: Or
For TEST #23: Decision Tree: 5 levels          After Cross-Val, Using Data-Set: Ou
For TEST #24: Decision Tree: 6 levels          After Cross-Val, Using Data-Set: Or
For TEST #25: Decision Tree: 6 levels          After Cross-Val, Using Data-Set: No
For TEST #26: Decision Tree: 6 levels          After Cross-Val, Using Data-Set: Or
For TEST #27: Decision Tree: 6 levels          After Cross-Val, Using Data-Set: Ou
For TEST #28: Decision Tree: 6 levels          After Cross-Val, Using Data-Set: Or
For TEST #29: Decision Tree: 6 levels          After Cross-Val, Using Data-Set: No
For TEST #30: Decision Tree: 6 levels          After Cross-Val, Using Data-Set: Or
For TEST #31: Decision Tree: 6 levels          After Cross-Val, Using Data-Set: Ou
For TEST #32: Decision Tree: 7 levels          After Cross-Val, Using Data-Set: Or
For TEST #33: Decision Tree: 7 levels          After Cross-Val, Using Data-Set: No
```

```
For TEST #34: Decision Tree: 7 levels          After Cross-Val, Using Data-Set: Or
For TEST #35: Decision Tree: 7 levels          After Cross-Val, Using Data-Set: Ou
For TEST #36: Decision Tree: 7 levels          After Cross-Val, Using Data-Set: Or
For TEST #37: Decision Tree: 7 levels          After Cross-Val, Using Data-Set: No
For TEST #38: Decision Tree: 7 levels          After Cross-Val, Using Data-Set: Or
For TEST #39: Decision Tree: 7 levels          After Cross-Val, Using Data-Set: Ou
For TEST #40: Decision Tree: 8 levels          After Cross-Val, Using Data-Set: Or
For TEST #41: Decision Tree: 8 levels          After Cross-Val, Using Data-Set: No
For TEST #42: Decision Tree: 8 levels          After Cross-Val, Using Data-Set: Or
For TEST #43: Decision Tree: 8 levels          After Cross-Val, Using Data-Set: Ou
For TEST #44: Decision Tree: 8 levels          After Cross-Val, Using Data-Set: Or
For TEST #45: Decision Tree: 8 levels          After Cross-Val, Using Data-Set: No
For TEST #46: Decision Tree: 8 levels          After Cross-Val, Using Data-Set: Or
For TEST #47: Decision Tree: 8 levels          After Cross-Val, Using Data-Set: Ou
For TEST #48: Decision Tree: 9 levels          After Cross-Val, Using Data-Set: Or
For TEST #49: Decision Tree: 9 levels          After Cross-Val, Using Data-Set: No
For TEST #50: Decision Tree: 9 levels          After Cross-Val, Using Data-Set: Or
For TEST #51: Decision Tree: 9 levels          After Cross-Val, Using Data-Set: Ou
For TEST #52: Decision Tree: 9 levels          After Cross-Val, Using Data-Set: Or
For TEST #53: Decision Tree: 9 levels          After Cross-Val, Using Data-Set: No
For TEST #54: Decision Tree: 9 levels          After Cross-Val, Using Data-Set: Or
For TEST #55: Decision Tree: 9 levels          After Cross-Val, Using Data-Set: Ou
For TEST #56: Decision Tree: 10 levels         After Cross-Val, Using Data-Set: Or
```

```python
#View these results again, this time just the top 10 using each sort method:
range_limit = min(10,results[f"LAST_TEST"]+1) #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) (top 10)")
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 10)")
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) (top 10)

Using ML Model: Random Forest(RF) and Original Data-Set, After SMOTE:
TEST #108: Average Accuracy is 87.57%, Average TPR is 87.62%, Average FNR is 12.38%
```

```
Using ML Model: XG Boost(XGB) and Original Data-Set, After SMOTE:
TEST #116: Average Accuracy is 86.87%, Average TPR is 86.93%, Average FNR is 13.07%

Using ML Model: Random Forest(RF) and Original Data-Set, After ONEHOT, After SMOTE:
TEST #110: Average Accuracy is 86.68%, Average TPR is 86.74%, Average FNR is 13.26%

Using ML Model: Random Forest(RF) and Outliers Addressed Data-Set, After ONEHOT, Afte
TEST #111: Average Accuracy is 86.48%, Average TPR is 86.53%, Average FNR is 13.47%

Using ML Model: XG Boost(XGB) and Original Data-Set, After ONEHOT, After SMOTE:
TEST #118: Average Accuracy is 86.48%, Average TPR is 86.53%, Average FNR is 13.47%

Using ML Model: Random Forest(RF) and Normalized Data-Set, After SMOTE:
TEST #109: Average Accuracy is 86.09%, Average TPR is 86.14%, Average FNR is 13.86%

Using ML Model: XG Boost(XGB) and Normalized Data-Set, After SMOTE:
TEST #117: Average Accuracy is 85.98%, Average TPR is 86.03%, Average FNR is 13.97%

Using ML Model: Random Forest(RF) and Outliers Addressed Data-Set:
TEST #107: Average Accuracy is 85.91%, Average TPR is 85.64%, Average FNR is 14.36%

Using ML Model: Random Forest(RF) and Normalized Data-Set:
TEST #105: Average Accuracy is 85.91%, Average TPR is 85.54%, Average FNR is 14.46%

Using ML Model: Logistic Regression(LR) and Outliers Addressed Data-Set, After ONEHOT
TEST #87: Average Accuracy is 85.79%, Average TPR is 85.85%, Average FNR is 14.15%



Results of ML Models: (sorted by False Negative Rate(FNR)) (top 10)

Using ML Model: Random Forest(RF) and Original Data-Set, After SMOTE:
TEST #108: Average Accuracy is 87.57%, Average TPR is 87.62%, Average FNR is 12.38%

Using ML Model: XG Boost(XGB) and Original Data-Set, After SMOTE:
TEST #116: Average Accuracy is 86.87%, Average TPR is 86.93%, Average FNR is 13.07%

Using ML Model: Random Forest(RF) and Original Data-Set, After ONEHOT, After SMOTE:
TEST #110: Average Accuracy is 86.68%, Average TPR is 86.74%, Average FNR is 13.26%

Using ML Model: Random Forest(RF) and Outliers Addressed Data-Set, After ONEHOT, Afte
TEST #111: Average Accuracy is 86.48%, Average TPR is 86.53%, Average FNR is 13.47%

Using ML Model: XG Boost(XGB) and Original Data-Set, After ONEHOT, After SMOTE:
TEST #118: Average Accuracy is 86.48%, Average TPR is 86.53%, Average FNR is 13.47%

Using ML Model: Random Forest(RF) and Normalized Data-Set, After SMOTE:
TEST #109: Average Accuracy is 86.09%, Average TPR is 86.14%, Average FNR is 13.86%

Using ML Model: XG Boost(XGB) and Normalized Data-Set, After SMOTE:
TEST #117: Average Accuracy is 85.98%, Average TPR is 86.03%, Average FNR is 13.97%
```

## Hypertuning:

```
#For Hyper-tuning and secondary validation, only three dataset variations will be used.
#"Original Data-Set, After ONEHOT, After SMOTE"
#"Outliers Addressed Data-Set, After ONEHOT, After SMOTE"
#"Original Data-Set, After SMOTE"
#
#Copying these over to easier variables for these two stages.
df1 = df_smote.copy()
df1.name = df_smote.name #For some reason df.copy() doesn't copy the "name" attribute.

df2 = df_onehot_smote.copy()
df2.name = df_onehot_smote.name #For some reason df.copy() doesn't copy the "name" attribu

df3 = df_onehot_outliers_addressed_smote.copy()
df3.name = df_onehot_outliers_addressed_smote.name #For some reason df.copy() doesn't copy

df_list = []
df_list = [df1, df2, df3]

classCol = "Target"
```

```
#Based on the initial results, for hyper-tuning, focusing on models RF, XGB, and LR.

#Since hyper-tuning takes the most time to run, of everything in this project,
#wrapping this section up in a "if" statement so I can turn it off for future top down run
#Once hypertuned values are retrieved, the output will be copied into a text block for ref
#perform_tuning = True
perform_tuning = False

#Models used for Hyper-tuning.
rf = RandomForestClassifier()
xgb = XGBClassifier()
lr = LogisticRegression(max_iter=10000)

#Parameter Options tried for Hyper-tuning.
params_rf = {'n_estimators':[100,200,300,400,500], 'min_samples_leaf':[5, 10, 15, 20, 25,
params_xgb = {'n_estimators': [100,200,300,400,500,600,700,800,900,1000], 'learning_rate':
params_lr = {'solver':['newton-cg', 'lbfgs', 'sag', 'saga', 'liblinear'], 'penalty':['l2']

folds = 20

if perform_tuning:
  params_rf = {'n_estimators':[100,200,300,400,500], 'min_samples_leaf':[5, 10, 15, 20, 25
  params_xgb = {'n_estimators': [100,200,300,400,500,600,700,800,900,1000], 'learning_rate
  params_lr = {'solver':['newton-cg', 'lbfgs', 'sag', 'saga', 'liblinear'], 'penalty':['l2

  for idx,df_not_used in enumerate(df_list):
    this_df = df_list[idx]

    feature_names=this_df.columns[this_df.columns != classCol]
```

```
#80% training and 20% test
X_train, X_test, Y_train, Y_test = train_test_split(this_df.loc[:, feature_names], thi

grid_rf = GridSearchCV(rf, param_grid=params_rf, cv=folds)
grid_rf.fit(X_train, Y_train)
print(f"Using Data-Set: {this_df.name}:")
print("Hyper-Tuned Parameters for Random Forest:", grid_rf.best_params_)
print("")

rs_xgb =  RandomizedSearchCV(xgb, param_distributions=params_xgb, cv=folds)
rs_xgb.fit(X_train, Y_train)
print(f"Using Data-Set: {this_df.name}:")
print("Hyper-Tuned Parameters for XGBoost:", rs_xgb.best_params_)
print("")

grid_lr = GridSearchCV(lr, param_grid=params_lr, cv=folds)
grid_lr.fit(X_train, Y_train)
print(f"Using Data-Set: {this_df.name}:")
print("Hyper-Tuned Parameters for Logistic Regression:", grid_lr.best_params_)
print("")

```

**Results from hyper-tuning:**

Using Data-Set: Original Data-Set, After SMOTE: Hyper-Tuned Parameters for Random Forest: {'min_samples_leaf': 10, 'n_estimators': 500}

Using Data-Set: Original Data-Set, After SMOTE: Hyper-Tuned Parameters for XGBoost: {'n_estimators': 1000, 'learning_rate': 0.3}

Using Data-Set: Original Data-Set, After SMOTE: Hyper-Tuned Parameters for Logistic Regression: {'C': 100, 'penalty': 'l2', 'solver': 'newton-cg'}

Using Data-Set: Original Data-Set, After ONEHOT, After SMOTE: Hyper-Tuned Parameters for Random Forest: {'min_samples_leaf': 5, 'n_estimators': 400}

Using Data-Set: Original Data-Set, After ONEHOT, After SMOTE: Hyper-Tuned Parameters for XGBoost: {'n_estimators': 500, 'learning_rate': 0.2}

Using Data-Set: Original Data-Set, After ONEHOT, After SMOTE: Hyper-Tuned Parameters for Logistic Regression: {'C': 100, 'penalty': 'l2', 'solver': 'newton-cg'}

Using Data-Set: Outliers Addressed Data-Set, After ONEHOT, After SMOTE: Hyper-Tuned Parameters for Random Forest: {'min_samples_leaf': 5, 'n_estimators': 200}

Using Data-Set: Outliers Addressed Data-Set, After ONEHOT, After SMOTE: Hyper-Tuned Parameters for XGBoost: {'n_estimators': 400, 'learning_rate': 0.5}

Using Data-Set: Outliers Addressed Data-Set, After ONEHOT, After SMOTE: Hyper-Tuned Parameters for Logistic Regression: {'C': 10, 'penalty': 'l2', 'solver': 'liblinear'}

```
#Apply Hyper-tuned parameters.
#I know there is an automated way of applying the Hyper-tuned parameters from the output o
#grid search, but this works too.

rf_tuned_df1 = RandomForestClassifier(min_samples_leaf = 10, n_estimators = 500)
xgb_tuned_df1 = XGBClassifier(n_estimators = 1000, learning_rate = 0.3)
lr_tuned_df1 = LogisticRegression(max_iter=10000, C=100, penalty="l2", solver="newton-cg")

rf_tuned_df2 = RandomForestClassifier(min_samples_leaf = 5, n_estimators = 400)
xgb_tuned_df2 = XGBClassifier(n_estimators = 500, learning_rate = 0.2)
lr_tuned_df2 = LogisticRegression(max_iter=10000, C=100, penalty="l2", solver="newton-cg")

rf_tuned_df3 = RandomForestClassifier(min_samples_leaf = 5, n_estimators = 200)
xgb_tuned_df3 = XGBClassifier(n_estimators = 400, learning_rate = 0.5)
lr_tuned_df3 = LogisticRegression(max_iter=10000, C=10, penalty="l2", solver="liblinear")

```

## Secondary Validation: (After Hypertuning):

```
#This will start testing all over again.
results = {}
test_num = 0
folds = 20

for test in ["LR_DF1","LR_DF2","LR_DF3","LR_TUNED_DF1","LR_TUNED_DF2","LR_TUNED_DF3",
             "RF_DF1","RF_DF2","RF_DF3","RF_TUNED_DF1","RF_TUNED_DF2","RF_TUNED_DF3",
             "XGB_DF1","XGB_DF2","XGB_DF3","XGB_TUNED_DF1","XGB_TUNED_DF2","XGB_TUNED_DF3"
  if test == "LR_DF1":
    model = lr
    test_name = "Logistic Regression(LR)"
    this_df = df1
  elif test == "LR_DF2":
    model = lr
    test_name = "Logistic Regression(LR)"
    this_df = df2
  elif test == "LR_DF3":
    model = lr
    test_name = "Logistic Regression(LR)"
    this_df = df3

  elif test == "LR_TUNED_DF1":
    model = lr_tuned_df1
    test_name = "Logistic Regression(LR) - Tuned"
    this_df = df1
  elif test == "LR_TUNED_DF2":
    model = lr_tuned_df2
```

```
    test_name = "Logistic Regression(LR) - Tuned"
    this_df = df2
  elif test == "LR_TUNED_DF3":
    model = lr_tuned_df3
    test_name = "Logistic Regression(LR) - Tuned"
    this_df = df3

  elif test == "RF_DF1":
    model = rf
    test_name = "Random Forest(RF)"
    this_df = df1
  elif test == "RF_DF2":
    model = rf
    test_name = "Random Forest(RF)"
    this_df = df2
  elif test == "RF_DF3":
    model = rf
    test_name = "Random Forest(RF)"
    this_df = df3

  elif test == "RF_TUNED_DF1":
    model = rf_tuned_df1
    test_name = "Random Forest(RF) - Tuned"
    this_df = df1
  elif test == "RF_TUNED_DF2":
    model = rf_tuned_df2
    test_name = "Random Forest(RF) - Tuned"
    this_df = df2
  elif test == "RF_TUNED_DF3":
    model = rf_tuned_df3
    test_name = "Random Forest(RF) - Tuned"
    this_df = df3

  elif test == "XGB_DF1":
    model = xgb
    test_name = "XG Boost(XGB)"
    this_df = df1
  elif test == "XGB_DF2":
    model = xgb
    test_name = "XG Boost(XGB)"
    this_df = df2
  elif test == "XGB_DF3":
    model = xgb
    test_name = "XG Boost(XGB)"
    this_df = df3

  elif test == "XGB_TUNED_DF1":
    model = xgb_tuned_df1
    test_name = "XG Boost(XGB) - Tuned"
    this_df = df1
  elif test == "XGB_TUNED_DF2":
```

```
    model = xgb_tuned_df2
    test_name = "XG Boost(XGB) - Tuned"
    this_df = df2
  elif test == "XGB_TUNED_DF3":
    model = xgb_tuned_df3
    test_name = "XG Boost(XGB) - Tuned"
    this_df = df3

  doCV(test_num,test_name,model,this_df,folds,classCol,1)
  test_num += 1
```

```
For TEST #0: Logistic Regression(LR)             After Cross-Val, Using Data-Set: Origir
For TEST #1: Logistic Regression(LR)             After Cross-Val, Using Data-Set: Origir
For TEST #2: Logistic Regression(LR)             After Cross-Val, Using Data-Set: Outlie
For TEST #3: Logistic Regression(LR) - Tuned     After Cross-Val, Using Data-Set: Origir
For TEST #4: Logistic Regression(LR) - Tuned     After Cross-Val, Using Data-Set: Origir
For TEST #5: Logistic Regression(LR) - Tuned     After Cross-Val, Using Data-Set: Outlie
For TEST #6: Random Forest(RF)                   After Cross-Val, Using Data-Set: Origir
For TEST #7: Random Forest(RF)                   After Cross-Val, Using Data-Set: Origir
For TEST #8: Random Forest(RF)                   After Cross-Val, Using Data-Set: Outlie
For TEST #9: Random Forest(RF) - Tuned           After Cross-Val, Using Data-Set: Origir
For TEST #10: Random Forest(RF) - Tuned          After Cross-Val, Using Data-Set: Origi
For TEST #11: Random Forest(RF) - Tuned          After Cross-Val, Using Data-Set: Outli
For TEST #12: XG Boost(XGB)                      After Cross-Val, Using Data-Set: Origi
For TEST #13: XG Boost(XGB)                      After Cross-Val, Using Data-Set: Origi
For TEST #14: XG Boost(XGB)                      After Cross-Val, Using Data-Set: Outli
For TEST #15: XG Boost(XGB) - Tuned              After Cross-Val, Using Data-Set: Origi
For TEST #16: XG Boost(XGB) - Tuned              After Cross-Val, Using Data-Set: Origi
For TEST #17: XG Boost(XGB) - Tuned              After Cross-Val, Using Data-Set: Outli
```

```
#View these results, after hyper-tuning, just the top 5 using each sort method:
range_limit = min(5,results[f"LAST_TEST"]+1) #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("")


#View these results, after hyper-tuning, just the bottom 5 using each sort method:
range_limit = min(5,results[f"LAST_TEST"]+1) #Bottom 5 results desired.

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

print("Results of ML Models: (sorted by accuracy) (bottom 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 of ML Models: (sorted by accuracy) (top 5)

Using ML Model: Random Forest(RF) and Original Data-Set, After SMOTE:
TEST #6: Average Accuracy is 88.44%, Average TPR is 88.37%, Average FNR is 11.63%

Using ML Model: Random Forest(RF) - Tuned and Original Data-Set, After ONEHOT, After
TEST #10: Average Accuracy is 88.24%, Average TPR is 88.18%, Average FNR is 11.82%

Using ML Model: XG Boost(XGB) and Original Data-Set, After SMOTE:
TEST #12: Average Accuracy is 88.24%, Average TPR is 88.17%, Average FNR is 11.83%

Using ML Model: Random Forest(RF) and Original Data-Set, After ONEHOT, After SMOTE:
TEST #7: Average Accuracy is 88.14%, Average TPR is 88.09%, Average FNR is 11.91%

Using ML Model: Random Forest(RF) and Outliers Addressed Data-Set, After ONEHOT, Afte
TEST #8: Average Accuracy is 87.85%, Average TPR is 87.78%, Average FNR is 12.22%



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

Using ML Model: Random Forest(RF) and Original Data-Set, After SMOTE:
TEST #6: Average Accuracy is 88.44%, Average TPR is 88.37%, Average FNR is 11.63%

Using ML Model: Random Forest(RF) - Tuned and Original Data-Set, After ONEHOT, After
TEST #10: Average Accuracy is 88.24%, Average TPR is 88.18%, Average FNR is 11.82%

Using ML Model: XG Boost(XGB) and Original Data-Set, After SMOTE:
TEST #12: Average Accuracy is 88.24%, Average TPR is 88.17%, Average FNR is 11.83%

Using ML Model: Random Forest(RF) and Original Data-Set, After ONEHOT, After SMOTE:
TEST #7: Average Accuracy is 88.14%, Average TPR is 88.09%, Average FNR is 11.91%

Using ML Model: Random Forest(RF) and Outliers Addressed Data-Set, After ONEHOT, Afte
TEST #8: Average Accuracy is 87.85%, Average TPR is 87.78%, Average FNR is 12.22%
```

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

Using ML Model: Logistic Regression(LR) - Tuned and Original Data-Set, After SMOTE:
TEST #3: Average Accuracy is 85.30%, Average TPR is 85.19%, Average FNR is 14.81%

Using ML Model: Logistic Regression(LR) and Original Data-Set, After SMOTE:
TEST #0: Average Accuracy is 85.30%, Average TPR is 85.19%, Average FNR is 14.81%

Using ML Model: XG Boost(XGB) - Tuned and Outliers Addressed Data-Set, After ONEHOT, 
TEST #17: Average Accuracy is 86.17%, Average TPR is 86.10%, Average FNR is 13.90%

Using ML Model: XG Boost(XGB) - Tuned and Original Data-Set, After SMOTE:
TEST #15: Average Accuracy is 86.36%, Average TPR is 86.28%, Average FNR is 13.72%

Using ML Model: Logistic Regression(LR) - Tuned and Original Data-Set, After ONEHOT, 
TEST #4: Average Accuracy is 86.37%, Average TPR is 86.28%, Average FNR is 13.72%
```

```
#Comparing Results.
#Due to random splits, the results change from run to run, but for the most part XGBoost w
#whereas Random Forest produces very similar results tuned, or not tuned.  Consistently th
#
#
#Using ML Model: Random Forest(RF) and Original Data-Set, After SMOTE:
#TEST #6: Average Accuracy is 88.44%, Average TPR is 88.37%, Average FNR is 11.63%

#Using ML Model: Random Forest(RF) - Tuned and Original Data-Set, After ONEHOT, After SMOT
#TEST #10: Average Accuracy is 88.24%, Average TPR is 88.18%, Average FNR is 11.82%

#Using ML Model: XG Boost(XGB) and Original Data-Set, After SMOTE:
#TEST #12: Average Accuracy is 88.24%, Average TPR is 88.17%, Average FNR is 11.83%

#Using ML Model: Random Forest(RF) and Original Data-Set, After ONEHOT, After SMOTE:
#TEST #7: Average Accuracy is 88.14%, Average TPR is 88.09%, Average FNR is 11.91%

#
# But are the differences between models, and data-sets statistically significant?
#
# For example, we have these three pairs of results:
#
# Pair #1: Same Data-set/Different models.
#Using ML Model: Random Forest(RF) and Original Data-Set, After SMOTE:
#TEST #6: Average Accuracy is 88.44%, Average TPR is 88.37%, Average FNR is 11.63%
#
#Using ML Model: XG Boost(XGB) and Original Data-Set, After SMOTE:
#TEST #12: Average Accuracy is 88.24%, Average TPR is 88.17%, Average FNR is 11.83%
#
```

```
#
# Pair #2: Same Data-set/RF vs RF Tuned
#Using ML Model: Random Forest(RF) - Tuned and Original Data-Set, After ONEHOT, After SMOT
#TEST #10: Average Accuracy is 88.24%, Average TPR is 88.18%, Average FNR is 11.82%

#Using ML Model: Random Forest(RF) and Original Data-Set, After ONEHOT, After SMOTE:
#TEST #7: Average Accuracy is 88.14%, Average TPR is 88.09%, Average FNR is 11.91%
#
# Each pair uses the same data-set, which make them suitable to compare results against
# each other using a paired dependant t-test.
#
#
def my_ttest(x,y,verbose=0):
  t,p = stats.ttest_rel(getTestAccVec(x),getTestAccVec(y))
  x_name = results[f"TEST{x}#NAME"]
  x_df_name = results[f"TEST{x}#DFNAME"]
  y_name = results[f"TEST{y}#NAME"]
  y_df_name = results[f"TEST{y}#DFNAME"]

  if p >= 0.05:
    if verbose > 0:
      print(f"TEST #{x}: {x_name}: Using Data-Set: {x_df_name} &")
      print(f"TEST #{y}: {y_name}: Using Data-Set: {y_df_name}")
      print(f"Results for Accuracy are NOT significantly different from each other, with t
      print("")
    return(False)
  else:
    if verbose > 0:
      print(f"TEST #{x}: {x_name}: Using Data-Set: {x_df_name} &")
      print(f"TEST #{y}: {y_name}: Using Data-Set: {y_df_name}")
      print(f"Results for Accuracy are significantly different from each other, with t={t}
      print("")
    return(True)

#Pair #1:
res = my_ttest(6,12,1)

#Pair #2:
res = my_ttest(10,7,1)
```

```
TEST #6: Random Forest(RF): Using Data-Set: Original Data-Set, After SMOTE &
TEST #12: XG Boost(XGB): Using Data-Set: Original Data-Set, After SMOTE
Results for Accuracy are NOT significantly different from each other, with t=0.26184748

TEST #10: Random Forest(RF) - Tuned: Using Data-Set: Original Data-Set, After ONEHOT, A
TEST #7: Random Forest(RF): Using Data-Set: Original Data-Set, After ONEHOT, After SMOT
Results for Accuracy are NOT significantly different from each other, with t=0.19792316
```

```
#The paired dependant t-test above revealed there is no significant difference between any
```