

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

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

Install Required Modules:

```
1 !pip install matplotlib
2 !pip install graphviz
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-packages (3.2.2)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /usr/local/lib/python3.7/dist-packages (from matplotlib) (2.4.7)
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.7/dist-packages (from matplotlib) (2.6.0)
Requirement already satisfied: numpy>=1.11 in /usr/local/lib/python3.7/dist-packages (from matplotlib) (1.19.5)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.7/dist-packages (from matplotlib) (1.3.1)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dist-packages (from matplotlib) (0.10.0)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/dist-packages (from matplotlib) (3.7.4)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-packages (from matplotlib) (1.16.0)
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: graphviz in /usr/local/lib/python3.7/dist-packages (0.10)
```

Load Required Libraries:

```
1 import sys
2 from google.colab import drive
3 import math
4 from statistics import mean, stdev
5 import pandas as pd
6 import numpy as np
7 from scipy import stats
8 import plotly
9 import matplotlib.pyplot as plt
10 import seaborn as sns
11
12 from sklearn import tree
13 from sklearn.naive_bayes import MultinomialNB
14 from sklearn.naive_bayes import GaussianNB
15 from sklearn.linear_model import LogisticRegression
16 from sklearn.svm import SVC
17 from sklearn.tree import DecisionTreeClassifier
18 from sklearn.neighbors import KNeighborsClassifier
```

```

19 from sklearn.ensemble import RandomForestClassifier
20 from xgboost import XGBClassifier
21
22 from sklearn.metrics import confusion_matrix
23 from sklearn.model_selection import train_test_split
24 from sklearn.metrics import classification_report
25 from sklearn import metrics
26 from sklearn.metrics import accuracy_score
27 from sklearn.metrics import f1_score
28
29 from sklearn.model_selection import KFold
30 from sklearn.model_selection import cross_val_score
31 from sklearn.model_selection import RandomizedSearchCV, GridSearchCV
32
33 import graphviz

```

Obtain Data-Set from Google Drive:

```

1 # Mounting google colab, this will prompt first time each session.
2 drive.mount('/content/drive',force_remount=True)
3 dataset_file = "/content/drive/My Drive/Colab Notebooks/heart_statlog_cleveland_hungary-fi
4 df=pd.read_csv(dataset_file,sep=',')
5 df.name = "Original Data-Set"
6 print(df.name)
7 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:

```

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

```

	Age	Sex	ChestPainType	RestingBP_s	Cholesterol	FastingBloodSugar	RestingECG	MaxHeartRate
0	40	1	2	140	289	0	0	178
1	49	0	3	160	180	0	0	172
2	57	1	2	130	386	0	1	147

Datatypes and Quantities:

```
1 #Check data types.
```

```
2 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
```

```
1 #Datatypes, counts, etc.
```

```
2 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:

```

1 #Check for NULL or missing entries. (none)
2 print(df.isna().any())
3 print("\n\n")
4 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:

```

1 #Look for rows that are 100% identical to each other.
2 #
3 dup_count = sum(df.duplicated())
4 print(f"There are {dup_count} duplicate rows in this dataset.\n")
5
6 #Dropping any duplicate entries.
7 df = df.drop_duplicates(ignore_index = True)
8 df.name = "Original Data-Set"
9
10 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:

```

1 #Check for out of bound entries(outliers) for nominal and binary attributes (including the
2 #Since all nominal and binary attributes have a valid contiguous integer range, ie 0-1, or
3 #we only need to look for those outside the range.
4 valid_values = {'Sex': [0,1], 'ChestPainType': [1,2,3,4], 'FastingBloodSugar': [0,1], 'Res
5
6 for col in ('Sex', 'ChestPainType', 'FastingBloodSugar', 'RestingECG', 'ExerciseAngina', '
7     valid = np.array(valid_values[col])
8     max_valid = valid.max()
9     min_valid = valid.min()
10    print(f"For attribute '{col}': Valid MAX: {max_valid}, Valid MIN: {min_valid}")
11    these_outliers = df[((df[col] < min_valid) | (df[col] > max_valid))]
12
13    if (these_outliers.shape[0] > 1):
14        print(f"For attribute '{col}': There are {these_outliers.shape[0]} outliers:\n")
15        print(these_outliers)
16        print("\n\n")
17    elif (these_outliers.shape[0] == 1):
18        print(f"For attribute '{col}': There is {these_outliers.shape[0]} outlier:\n")
19        print(these_outliers)
20        print("\n\n")
21    else:
22        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:

```
1 #From above, there is a problem with one entry regarding the ST_Slope attribute, it is zer
2 #
3 #The documentation at https://ieee-dataport.org/open-access/heart-disease-dataset-comprehe
4 #Shows a range of 0-2, but in the definition of the mapped nominal values it shows:
5 #
6 # -- Value 1: upsloping
7 # -- Value 2: flat
8 # -- Value 3: downsloping
9 #
10 print(df['ST_Slope'].value_counts().sort_index())
11 print("\n")
12 #
13 #
14 #Since there is only one entry out of range, making the assumption that the correct range
15 #
16 #Will simply drop this entry.
17 df = df[df['ST_Slope'] != 0]
18 df.name = "Original Data-Set"
19
20 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:

```

1 #Basic Statistics of the dataset.(Measures of Center/Central Tendency, and Measures of Var
2 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:

```

1 #Assigning descriptive labels for all possible values for all nominal/binary attributes.
2 #
3 labels = {'Sex': ['Female', 'Male'], 'ChestPainType': ['Typical Angina', 'A-Typical Angina',
4             'FastingBloodSugar': ['<= 120 mg/dl', '> 120 mg/dl'], 'RestingECG': ['Normal', '
5             'ExerciseAngina':['No', 'Yes'], 'ST_Slope': ['Upsloping', 'Flat', 'Downsloping']
6
7

```

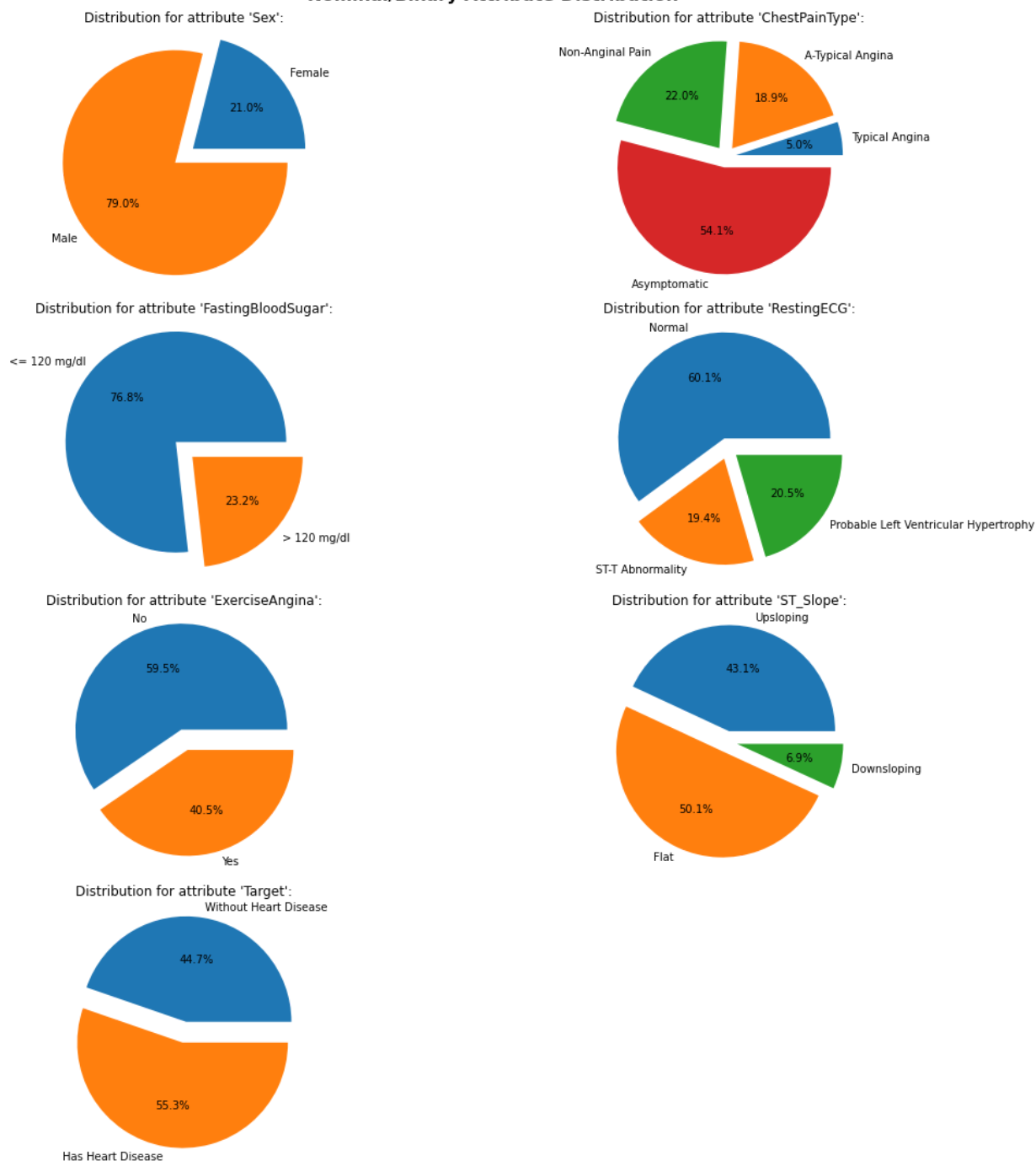
```

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

```

```
2 #
3 #Breaking up the dataset into two groups, those with heart disease and those without.
4 with_heart_disease = df[df['Target'] == 1]
5 no_heart_disease = df[df['Target'] == 0]
6
7 #Breaking up the dataset into four groups, by Sex, and those with heart disease and those
8 with_heart_disease_male = df[(df['Target'] == 1) & (df['Sex'] == 1)]
9 with_heart_disease_female = df[(df['Target'] == 1) & (df['Sex'] == 0)]
10 no_heart_disease_male = df[(df['Target'] == 0) & (df['Sex'] == 1)]
11 no_heart_disease_female = df[(df['Target'] == 0) & (df['Sex'] == 0)]
12
13 #For these groups remove the "Sex" column from the data.
14 with_heart_disease_male = with_heart_disease_male.drop("Sex",axis=1)
15 with_heart_disease_female = with_heart_disease_female.drop("Sex",axis=1)
16 no_heart_disease_male = no_heart_disease_male.drop("Sex",axis=1)
17 no_heart_disease_female = no_heart_disease_female.drop("Sex",axis=1)

1 #Visualizing the distribution of the categorical attributes, including the class variable.
2 ax=1
3 plt.figure(figsize=(15,15))
4
5 for col in ('Sex', 'ChestPainType', 'FastingBloodSugar', 'RestingECG', 'ExerciseAngina','S
6     plt.subplot(4,2,ax)
7     these_labels = labels[col]
8     plt.title(f"Distribution for attribute '{col}':")
9     plt.pie(df[col].value_counts().sort_index(),
10         autopct = '%1.1f%%', labels=these_labels,
11         explode=tuple([0.1] * len(these_labels)))
12     plt.axis('equal')
13     ax+=1
14
15 plt.suptitle('Nominal/Binary Attribute Distribution',y=1.01, size = 16, color = 'black', w
16 plt.tight_layout()
17 plt.savefig("nominal_dist.pdf",dpi=1200, bbox_inches='tight')
```

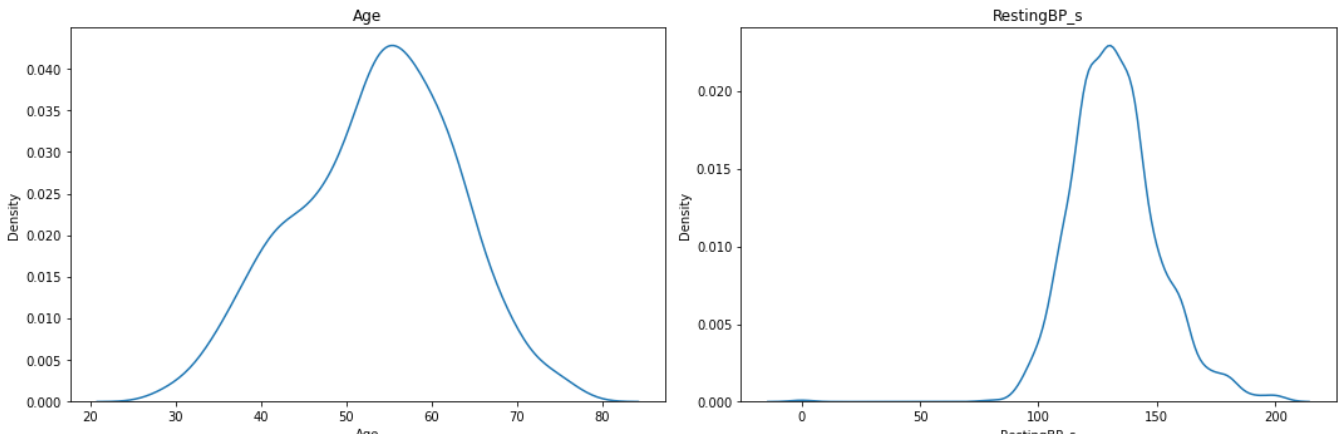

Nominal/Binary Attribute Distribution

```

1 #Visualizing the overall distribution of the numeric attributes.
2 plt.figure(figsize=(15,15))
3
4 ax=1
5 for col in ('Age', 'RestingBP_s', 'Cholesterol', 'MaxHeartRate', 'OldPeak'):
6     plt.subplot(3,2,ax)
7     plt.title(col)
8     sns.kdeplot(x=df[col])

```

```
9     ax += 1
10
11 plt.suptitle('Numeric Attribute Distribution',y=1.01, size = 16, color = 'black', weight='
12 plt.tight_layout()
13 plt.savefig("numeric_dist.pdf",dpi=1200, bbox_inches='tight')
```

Numeric Attribute Distribution

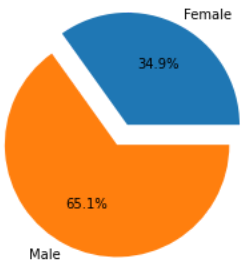
```

1 #Visualizing the distribution of the categorical attributes, by target
2 ax=1
3 plt.figure(figsize=(15,20))
4 plt.axis('equal')
5
6 for col in ('Sex','ChestPainType', 'FastingBloodSugar', 'RestingECG', 'ExerciseAngina','ST
7     plt.subplot(6,2,ax)
8     these_labels = labels[col]
9     plt.title(f"Distribution for attribute '{col}' Without Heart Disease:")
10    plt.pie(no_heart_disease[col].value_counts().sort_index(),
11        autopct = '%1.1f%%', labels=these_labels,
12        explode=tuple([0.1] * len(these_labels)))
13    ax+=1
14    plt.subplot(6,2,ax)
15    these_labels = labels[col]
16    plt.title(f"Distribution for attribute '{col}' With Heart Disease:")
17    plt.pie(with_heart_disease[col].value_counts().sort_index(),
18        autopct = '%1.1f%%', labels=these_labels,
19        explode=tuple([0.1] * len(these_labels)))
20    ax+=1
21
22 plt.suptitle('Nominal/Binary Attribute Distribution by Target',y=1.01, size = 16, color =
23 plt.tight_layout()
24 plt.savefig("nominal_dist_by_target.pdf",dpi=1200, bbox_inches='tight')

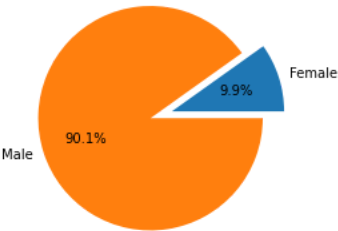
```

Nominal/Binary Attribute Distribution by Target

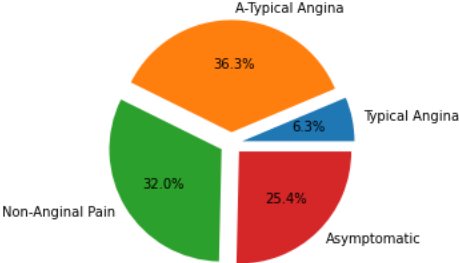
Distribution for attribute 'Sex' Without Heart Disease:



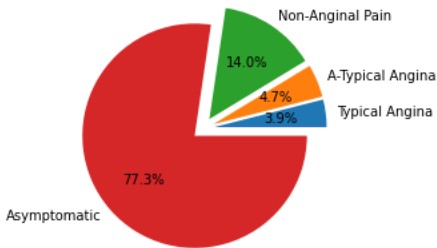
Distribution for attribute 'Sex' With Heart Disease:



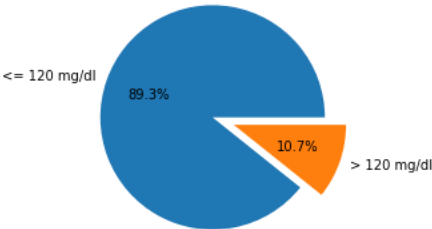
Distribution for attribute 'ChestPainType' Without Heart Disease:



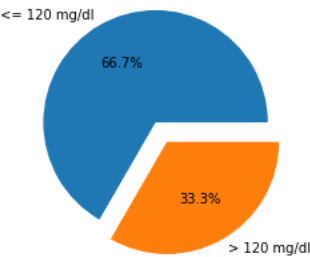
Distribution for attribute 'ChestPainType' With Heart Disease:



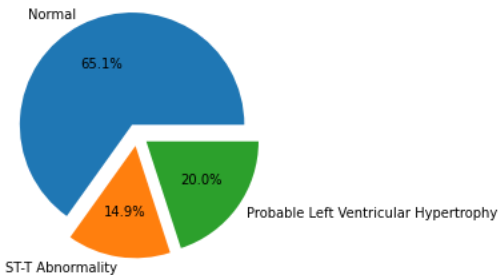
Distribution for attribute 'FastingBloodSugar' Without Heart Disease:



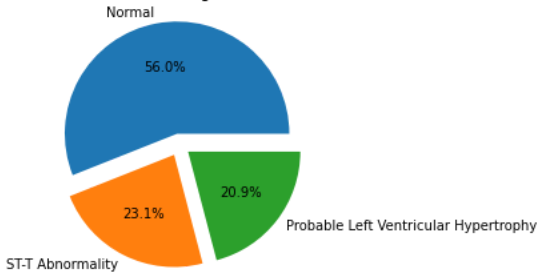
Distribution for attribute 'FastingBloodSugar' With Heart Disease:



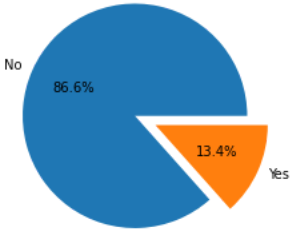
Distribution for attribute 'RestingECG' Without Heart Disease:



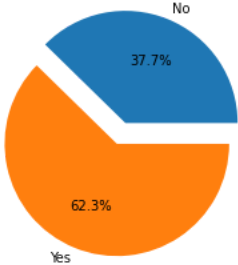
Distribution for attribute 'RestingECG' With Heart Disease:



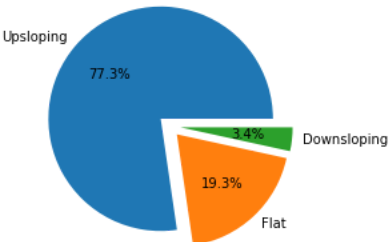
Distribution for attribute 'ExerciseAngina' Without Heart Disease:



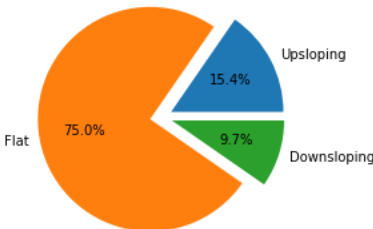
Distribution for attribute 'ExerciseAngina' With Heart Disease:



Distribution for attribute 'ST_Slope' Without Heart Disease:

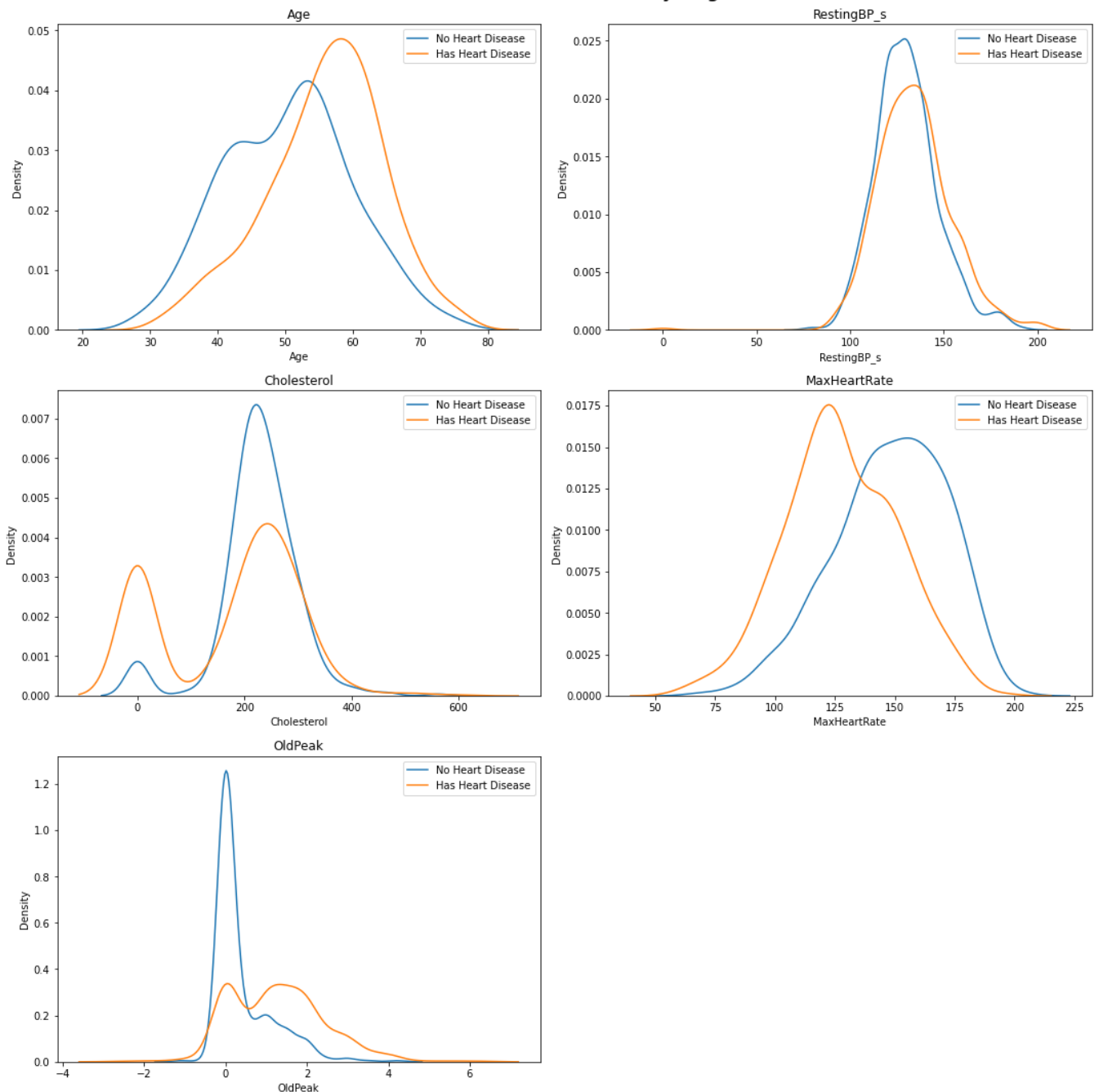


Distribution for attribute 'ST_Slope' With Heart Disease:



```
1 #Visualizing the distribution of the numeric attributes by Target:
2 plt.figure(figsize=(15,15))
3
4 ax=1
5 for col in ('Age', 'RestingBP_s', 'Cholesterol', 'MaxHeartRate', 'OldPeak'):
6     plt.subplot(3,2,ax)
7     plt.title(col)
8     sns.kdeplot(x=no_heart_disease[col],label = "No Heart Disease")
9     sns.kdeplot(x=with_heart_disease[col],label = "Has Heart Disease")
10    plt.legend()
11    ax += 1
12
13 plt.suptitle('Numeric Attribute Distribution by Target',y=1.01, size = 16, color = 'black')
14 plt.tight_layout()
15 plt.savefig("numeric_dist_by_target.pdf",dpi=1200, bbox_inches='tight')
```

Numeric Attribute Distribution by Target



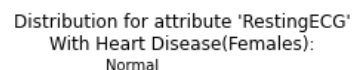
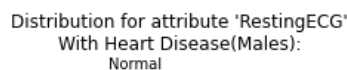
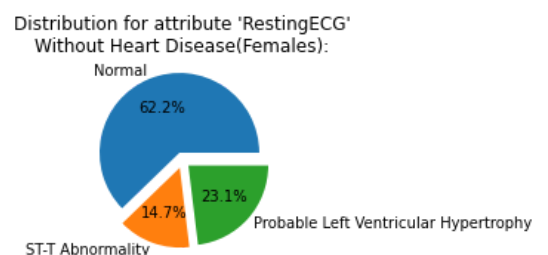
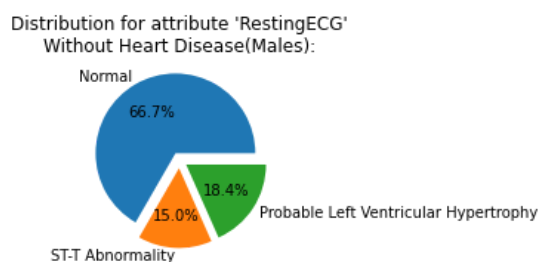
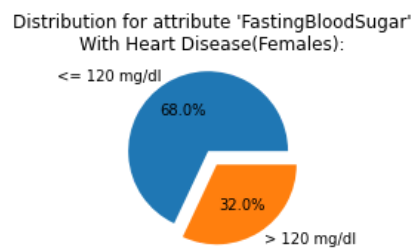
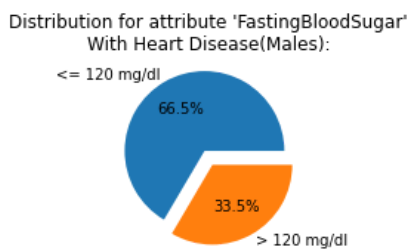
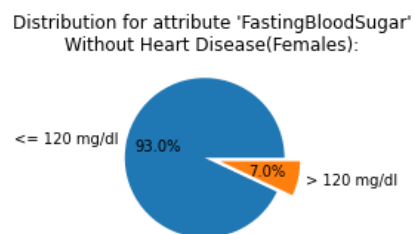
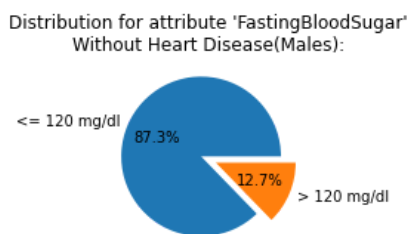
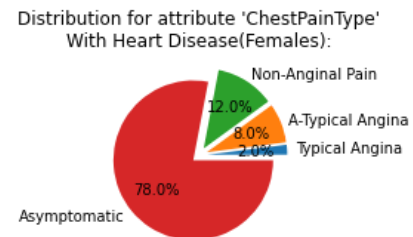
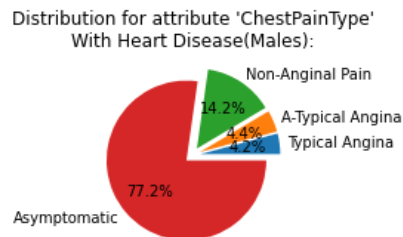
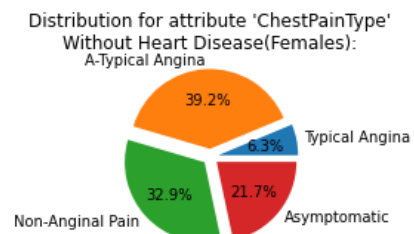
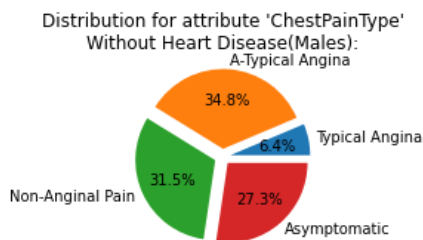
```

1 #Visualizing the distribution of the categorical attributes, by target and by sex
2 #For report purposes, breaking this up into two separate pages.
3 #This one for attributes 'ChestPainType', 'FastingBloodSugar', 'RestingECG'
4 #
5 ax=1
6 plt.figure(figsize=(15,30))
7 plt.axis('equal')
8
9 for col in ('ChestPainType', 'FastingBloodSugar', 'RestingECG'):
10     plt.subplot(12,2,ax)
11     these_labels = labels[col]
12     plt.title(f"Distribution for attribute '{col}'\nWithout Heart Disease(Males):")
13     plt.pie(no_heart_disease_male[col].value_counts().sort_index(),
14             autopct = '%1.1f%%', labels=these_labels,

```

```
15     explode=tuple([0.1] * len(these_labels)))
16     ax+=1
17
18     plt.subplot(12,2,ax)
19     these_labels = labels[col]
20     plt.title(f"Distribution for attribute '{col}'\nWithout Heart Disease(Females):")
21     plt.pie(no_heart_disease_female[col].value_counts().sort_index(),
22            autopct = '%1.1f%%', labels=these_labels,
23            explode=tuple([0.1] * len(these_labels)))
24     ax+=1
25
26     plt.subplot(12,2,ax)
27     these_labels = labels[col]
28     plt.title(f"Distribution for attribute '{col}'\nWith Heart Disease(Males):")
29     plt.pie(with_heart_disease_male[col].value_counts().sort_index(),
30            autopct = '%1.1f%%', labels=these_labels,
31            explode=tuple([0.1] * len(these_labels)))
32     ax+=1
33
34     plt.subplot(12,2,ax)
35     these_labels = labels[col]
36     plt.title(f"Distribution for attribute '{col}'\nWith Heart Disease(Females):")
37     plt.pie(with_heart_disease_female[col].value_counts().sort_index(),
38            autopct = '%1.1f%%', labels=these_labels,
39            explode=tuple([0.1] * len(these_labels)))
40     ax+=1
41
42 plt.suptitle('Nominal/Binary Attribute Distribution by Target and by Sex',y=1.01, size = 1
43 plt.tight_layout()
44 plt.savefig("nominal_dist_by_target_by_sex1.pdf",dpi=1200, bbox_inches='tight')
```

Nominal/Binary Attribute Distribution by Target and by Sex



```

1 #Visualizing the distribution of the categorical attributes, by target and by sex
2 #For report purposes, breaking this up into two separate pages.
3 #This one for attributes 'ExerciseAngina','ST_Slope'
4 #
5 ax=1
6 plt.figure(figsize=(15,30))
7 plt.axis('equal')
8
9 for col in ('ExerciseAngina','ST_Slope'):
10     plt.subplot(12,2,ax)
11     these_labels = labels[col]
12     plt.title(f"Distribution for attribute '{col}'\nWithout Heart Disease(Males):")

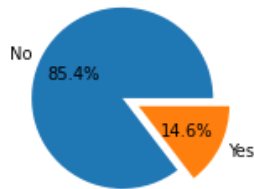
```



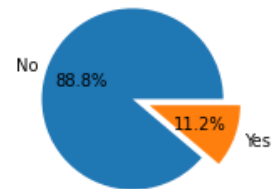
```
13 plt.pie(no_heart_disease_male[col].value_counts().sort_index(),
14         autopct = '%1.1f%%', labels=these_labels,
15         explode=tuple([0.1] * len(these_labels)))
16 ax+=1
17
18 plt.subplot(12,2,ax)
19 these_labels = labels[col]
20 plt.title(f"Distribution for attribute '{col}'\nWithout Heart Disease(Females):")
21 plt.pie(no_heart_disease_female[col].value_counts().sort_index(),
22         autopct = '%1.1f%%', labels=these_labels,
23         explode=tuple([0.1] * len(these_labels)))
24 ax+=1
25
26 plt.subplot(12,2,ax)
27 these_labels = labels[col]
28 plt.title(f"Distribution for attribute '{col}'\nWith Heart Disease(Males):")
29 plt.pie(with_heart_disease_male[col].value_counts().sort_index(),
30         autopct = '%1.1f%%', labels=these_labels,
31         explode=tuple([0.1] * len(these_labels)))
32 ax+=1
33
34 plt.subplot(12,2,ax)
35 these_labels = labels[col]
36 plt.title(f"Distribution for attribute '{col}'\nWith Heart Disease(Females):")
37 plt.pie(with_heart_disease_female[col].value_counts().sort_index(),
38         autopct = '%1.1f%%', labels=these_labels,
39         explode=tuple([0.1] * len(these_labels)))
40 ax+=1
41
42 plt.suptitle('Nominal/Binary Attribute Distribution by Target and by Sex, Cont\'d',y=1.01,
43 plt.tight_layout()
44 plt.savefig("nominal_dist_by_target_by_sex2.pdf",dpi=1200, bbox_inches='tight')
```

Nominal/Binary Attribute Distribution by Target and by Sex, Cont'd

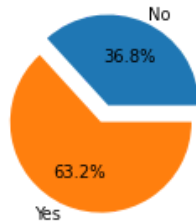
Distribution for attribute 'ExerciseAngina'
Without Heart Disease(Males):



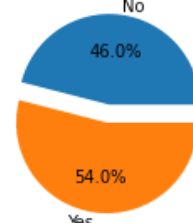
Distribution for attribute 'ExerciseAngina'
Without Heart Disease(Females):



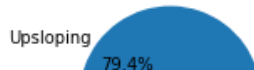
Distribution for attribute 'ExerciseAngina'
With Heart Disease(Males):



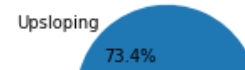
Distribution for attribute 'ExerciseAngina'
With Heart Disease(Females):



Distribution for attribute 'ST_Slope'
Without Heart Disease(Males):



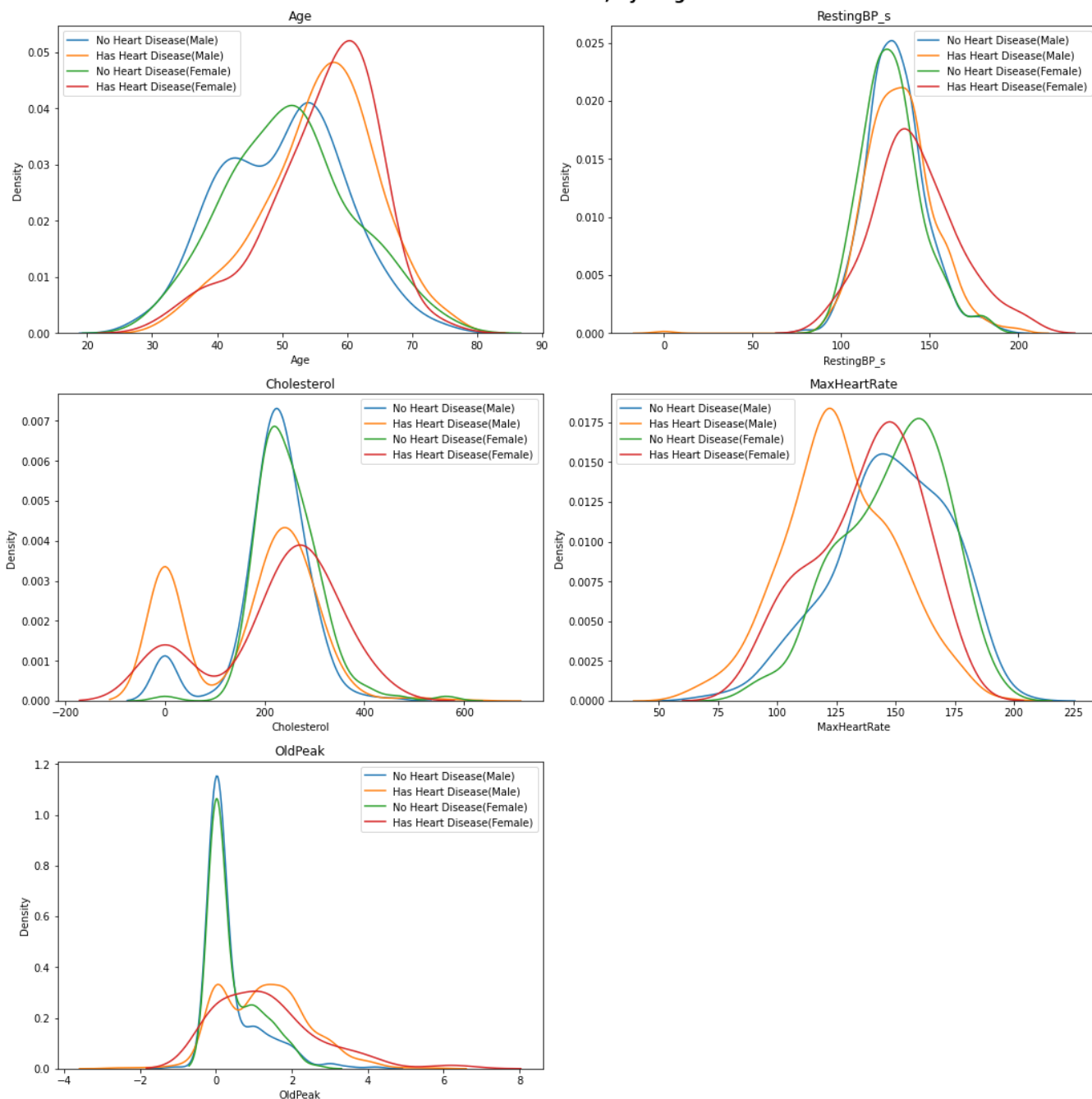
Distribution for attribute 'ST_Slope'
Without Heart Disease(Females):



```

1 #Visualizing the distribution of the numerical attributes, by target and by sex
2 plt.figure(figsize=(15,15))
3
4 ax=1
5 for col in ('Age', 'RestingBP_s', 'Cholesterol', 'MaxHeartRate', 'OldPeak'):
6     plt.subplot(3,2,ax)
7     plt.title(col)
8     sns.kdeplot(x=no_heart_disease_male[col],label = "No Heart Disease(Male)")
9     sns.kdeplot(x=with_heart_disease_male[col],label = "Has Heart Disease(Male)")
10    sns.kdeplot(x=no_heart_disease_female[col],label = "No Heart Disease(Female)")
11    sns.kdeplot(x=with_heart_disease_female[col],label = "Has Heart Disease(Female)")
12    plt.legend()
13    ax += 1
14
15 plt.suptitle('Numeric Attribute Distribution, by Target and Sex',y=1.01, size = 16, color
16 plt.tight_layout()
17 plt.savefig("numeric_dist_by_target_by_sex.pdf",dpi=1200, bbox_inches='tight')

```

Numeric Attribute Distribution, by Target and Sex**Outlier Detection:**

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

```
2 #Using for outlier detection three methods.
3 #Note: for the next stage in this project, Module 3,
4 #one or more of these outlier detection methods will be used.
5 #For now, we only want to see how many outliers per attribute are detected with each appro
6 #
7 #Methods:
8 # #1 1.5IQR range
9 # #2 mean +/- 3*ST-DEV (same as GT Absolute(Z-Score))
10 # #3 Rejecting those with a value of zero (based on visualization, only needed for 'Chole
11
12 def IQR1_5_upper(data, col):
13     Q3 = np.quantile(data[col], 0.75)
14     Q1 = np.quantile(data[col], 0.25)
15     IQR = Q3 - Q1
16     return(Q3+(1.5*IQR))
17
18 def IQR1_5_lower(data, col):
19     Q3 = np.quantile(data[col], 0.75)
20     Q1 = np.quantile(data[col], 0.25)
21     IQR = Q3 - Q1
22     return(Q1-(1.5*IQR))
23
24
25 for col in ('Age', 'RestingBP_s', 'Cholesterol', 'MaxHeartRate', 'OldPeak'):
26     upper1 = IQR1_5_upper(df,col)
27     lower1 = IQR1_5_lower(df,col)
28     stdev3 = 3*df[col].std()
29     mean = df[col].mean()
30     upper2 = mean + stdev3
31     lower2 = mean - stdev3
32
33     these_outliers1 = df[(df[col] < lower1) | (df[col] > upper1)]
34     these_outliers2 = df[(df[col] < lower2) | (df[col] > upper2)]
35     these_outliers3 = df[df[col] == 0]
36
37     print(f"For attribute '{col}': The mean is {mean}, stdev3 is {stdev3}")
38     print(f"For 1.5IQR the lower range is {lower1} the upper range is {upper1}")
39     print(f"For mean +/- 3STDEV the lower range is {lower2} the upper range is {upper2}")
40     print(f"\n")
41
42     print(f"Using 1.5IQR Method:")
43     if (these_outliers1.shape[0] > 1):
44         print(f"For attribute '{col}': There are {these_outliers1.shape[0]} outliers:\n")
45         print(these_outliers1)
46         print("\n")
47     elif (these_outliers1.shape[0] == 1):
48         print(f"For attribute '{col}': There is {these_outliers1.shape[0]} outlier:\n")
49         print(these_outliers1)
50         print("\n")
51     else:
52         print(f"For attribute '{col}': There are no outliers.\n")
```

```

53     print("\n")
54
55     print(f"Using mean +/- 3STDEV Method:")
56     if (these_outliers2.shape[0] > 1):
57         print(f"For attribute '{col}': There are {these_outliers2.shape[0]} outliers:\n")
58         print(these_outliers2)
59         print("\n")
60     elif (these_outliers2.shape[0] == 1):
61         print(f"For attribute '{col}': There is {these_outliers2.shape[0]} outlier:\n")
62         print(these_outliers2)
63         print("\n")
64     else:
65         print(f"For attribute '{col}': There are no outliers.\n")
66         print("\n")
67
68     if(col == 'Cholesterol'):
69         print(f"Identifying 'zero' values(for 'Cholesterol') Method:")
70         if (these_outliers3.shape[0] > 1):
71             print(f"For attribute '{col}': There are {these_outliers3.shape[0]} outliers:\n")
72             print(these_outliers3)
73             print("\n")
74         elif (these_outliers3.shape[0] == 1):
75             print(f"For attribute '{col}': There is {these_outliers3.shape[0]} outlier:\n")
76             print(these_outliers3)
77             print("\n")
78         else:
79             print(f"For attribute '{col}': There are no outliers.\n")
80             print("\n")
81
82     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:

```

1 #Before we can run multinomial Naive Bayes we must remove any negative numbers in the data
2 mins = df.min()
3 print(mins)
4 print("\n\n")
5 #There are only negative values for attribute 'OldPeak'
6 #Applying a simple shift to eliminate any negatives.
7
8 df_no_negs = df.copy()
9 df_no_negs.name = "No Negatives Data-Set"
10
11 df_no_negs['OldPeak'] = df_no_negs['OldPeak'] + abs(df_no_negs['OldPeak'].min())
12
13 #Visualizing the overall distribution of 'OldPeak' before and after modification for negat
14 plt.figure(figsize=(10,5))

```

```

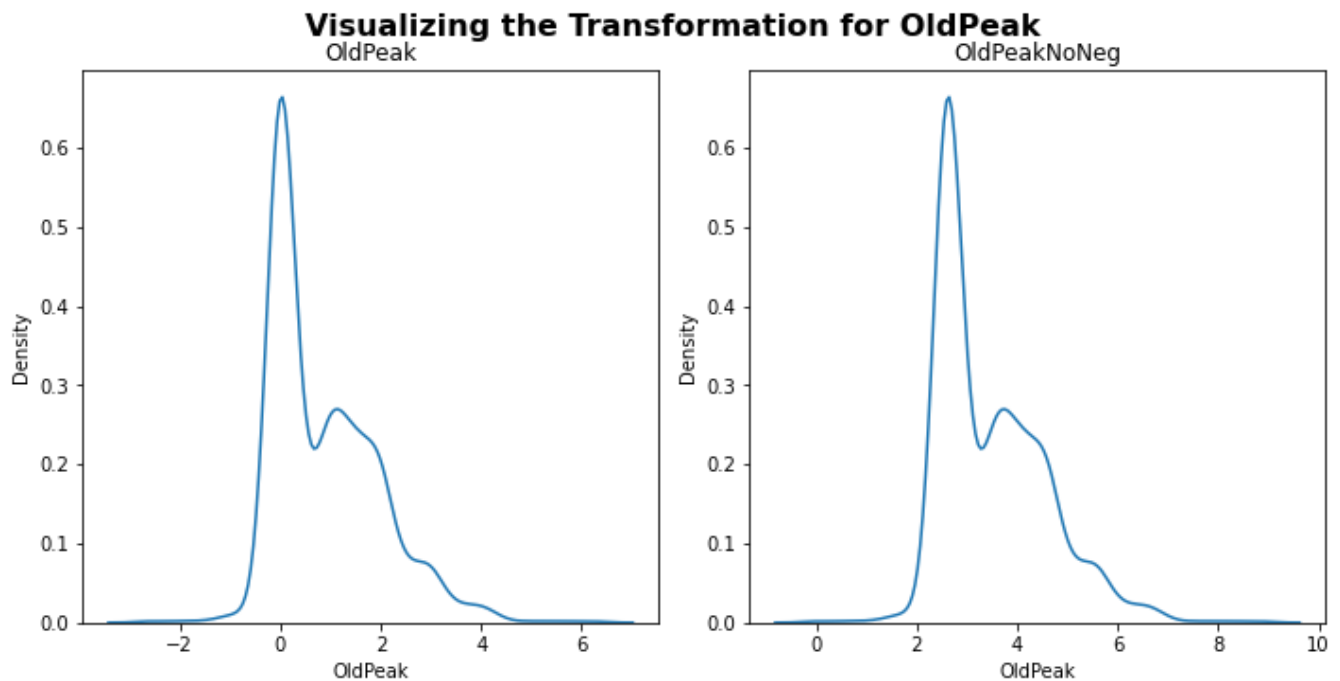
15
16 plt.subplot(1,2,1)
17 plt.title('OldPeak')
18 sns.kdeplot(x=df['OldPeak'])
19
20 plt.subplot(1,2,2)
21 plt.title('OldPeakNoNeg')
22 sns.kdeplot(x=df_no_negs['OldPeak'])
23
24 plt.suptitle('Visualizing the Transformation for OldPeak',y=1.01, size = 16, color = 'black')
25 plt.tight_layout()
26 plt.savefig("oldpeak_transformation.pdf",dpi=1200, bbox_inches='tight')
27

```

```

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

```



```

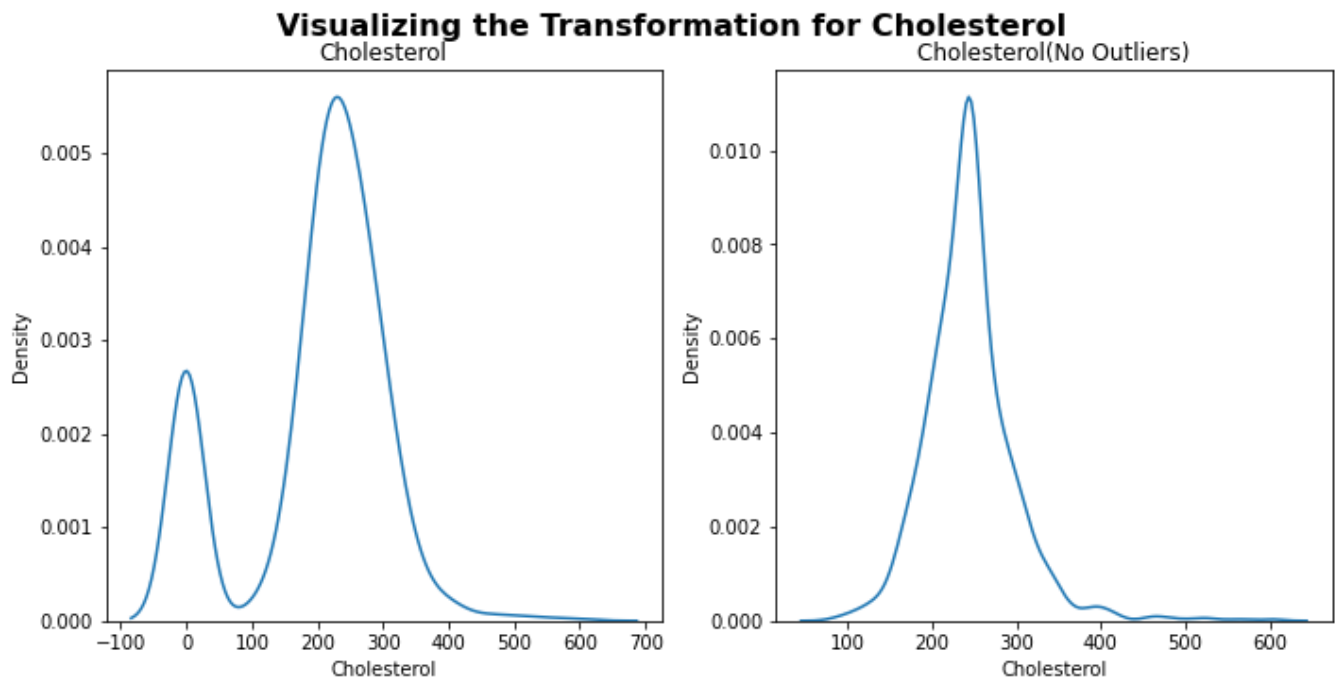
1 #Only addressing outliers for attribute Cholesterol, specifically the instances where Chol
2 #All other outliers appear in much smaller quantities.
3

```

```

>
4 df_outliers_addressed = df_no_negs.copy()
5 df_outliers_addressed.name = "Outliers Addressed Data-Set"
6
7 cholesterol_mean = df_outliers_addressed['Cholesterol'][df_outliers_addressed['Cholesterol']
8
9 df_outliers_addressed['Cholesterol'].replace(to_replace=0.0, value=cholesterol_mean, inplace=True)
10
11
12 #Visualizing the overall distribution Cholesterol before and after dealing with outliers.
13 plt.figure(figsize=(10,5))
14
15 plt.subplot(1,2,1)
16 plt.title('Cholesterol')
17 sns.kdeplot(x=df['Cholesterol'])
18
19 plt.subplot(1,2,2)
20 plt.title('Cholesterol(No Outliers)')
21 sns.kdeplot(x=df_outliers_addressed['Cholesterol'])
22
23 plt.suptitle('Visualizing the Transformation for Cholesterol',y=1.01, size = 16, color = 'black')
24 plt.tight_layout()

```



Normalize Numeric Data for potential model training:

```

1 #Normalizing numeric data to see if this helps, or hinders the accuracy of the ML models.
2 df_normalized = df_outliers_addressed.copy()
3 df_normalized.name = "Normalized Data-Set"
4
5 for col in ('Age', 'RestingBP_s', 'Cholesterol', 'MaxHeartRate', 'OldPeak'):

```



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

Create ONE-HOT columns for all categorical attributes:

```
1 #For Nominal & Binary (ie categorical) attributes, perform one-hot conversion.
2 #convert only categorical variables/features to dummy/one-hot features
3 cat_cols = ['Sex','ChestPainType', 'FastingBloodSugar', 'RestingECG', 'ExerciseAngina','ST
4
5 df_onehot = pd.get_dummies(df, columns=cat_cols, prefix = cat_cols)
6 df_onehot.name = "Original Data-Set, onehot"
7
8 df_onehot_no_negs = pd.get_dummies(df_no_negs, columns=cat_cols, prefix = cat_cols)
9 df_onehot_no_negs.name = "No negatives Data-Set, onehot"
10
11 df_onehot_outliers_addressed = pd.get_dummies(df_outliers_addressed, columns=cat_cols, pre
12 df_onehot_outliers_addressed.name = "Outliers Addressed Data-Set, onehot"
13
14 df_onehot_normalized = pd.get_dummies(df_normalized, columns=cat_cols, prefix = cat_cols)
15 df_onehot_normalized.name = "Normalized Data-Set, onehot"
16
```

ML Algorithms:

```
1 def do_DT(df,levels,class_col_name,verbose=0):
2     #not disabling randomness.
3     #np.random.seed(0)
4
5     # Split dataset into training set and test set
6     feature_names=df.columns[df.columns != class_col_name ]
7     # 80% training and 20% test
8     X_train, X_test, Y_train, Y_test = train_test_split(df.loc[:, feature_names], df[class_c
9
10    clf = tree.DecisionTreeClassifier(max_depth=levels,criterion='gini')
11    clf = clf.fit(X_train, Y_train)
12    if (verbose >= 1):
13        print(f"Successfully trained the decision tree for {levels} levels...")
14
15    # Let's make the predictions on the test set that we set aside earlier using the trained
16    Y_pred = clf.predict(X_test)
17
18    cf=confusion_matrix(Y_test, Y_pred)
19    tn, fp, fn, tp=cf.ravel()
20    tpr=0.0
21    fpr=0.0
22    tpr = tp/(tp+fp)
23    fpr = fp/(fp+tn)
24    fnr = fn/(fn+tp)
25
```

```
26 if (verbose >= 2):
27     print ("Confusion Matrix")
28     print(cf)
29     print("")
30     print ("TP: ", tp,", FP: ", fp,", TN: ", tn,", FN:", fn)
31     print ("TPR: ",tpr,", FPR: ",fpr, "FNR: ",fnr)
32
33 #print precision, recall, and accuracy from the perspective of each of the class (0 and
34 if (verbose >= 2):
35     print(classification_report(Y_test, Y_pred, digits=3))
36
37 accuracy = accuracy_score(Y_test, Y_pred)
38 f1_weighted = f1_score(Y_test, Y_pred,average='weighted')
39
40 if (verbose >= 1):
41     print(f"Accuracy is: {accuracy}")
42     print(f"F1 Weighted is: {f1_weighted}")
43     print("")
44
45 return(accuracy,f1_weighted,tpr,fpr,fnr)

1 def do_mnNB(df,class_col_name,verbose=0):
2     #not disabling randomness.
3     #np.random.seed(0)
4
5     # Split dataset into training set and test set
6     feature_names=df.columns[df.columns != class_col_name ]
7     # 80% training and 20% test
8     X_train, X_test, Y_train, Y_test = train_test_split(df.loc[:, feature_names], df[class_c
9
10    #Create a MultiNomial NB Classifier
11    nb = MultinomialNB()
12
13    #Train the model using the training sets
14    nb.fit(X_train, Y_train)
15
16    #Predict the response for test dataset
17    Y_pred = nb.predict(X_test)
18
19    if (verbose >= 2):
20        print ("Total Columns (including class)",len(df.columns))
21        print("Classes ",nb.classes_)
22        print("Number of records for classes ",nb.class_count_)
23        print("Log prior probability for classes ", nb.class_log_prior_)
24        print("Log conditional probability for each feature given a class\n",nb.feature_log_pr
25
26    cf=confusion_matrix(Y_test, Y_pred)
27    tn, fp, fn, tp=cf.ravel()
28    tpr = tp/(tp+fp)
29    fpr = fp/(fp+tn)
30    fnr = fn/(fn+tp)
```

```

31
32 if (verbose >= 2):
33     print ("Confusion Matrix")
34     print(cf)
35     print("")
36     print ("TP: ", tp,",", FP: ", fp,",", TN: ", tn,",", FN:",", fn)
37     print ("TPR: ",tpr,",", FPR: ",fpr,",", FNR: ",fnr)
38
39 if (verbose >= 2):
40     print(classification_report(Y_test, Y_pred, digits=3))
41
42 accuracy = accuracy_score(Y_test, Y_pred)
43 f1_weighted = f1_score(Y_test, Y_pred,average='weighted')
44
45 if (verbose >= 1):
46     print(f"Accuracy is: {accuracy}")
47     print(f"F1 Weighted is: {f1_weighted}")
48     print("")
49
50 return(accuracy,f1_weighted,tpr,fpr,fnr)


1 def do_gaNB(df,class_col_name,verbose=0):
2     #not disabling randomness.
3     #np.random.seed(0)
4
5     # Split dataset into training set and test set
6     feature_names=df.columns[df.columns != class_col_name ]
7     # 80% training and 20% test
8     X_train, X_test, Y_train, Y_test = train_test_split(df.loc[:, feature_names], df[class_c
9
10    #Create a Gaussian NB Classifier
11    nb = GaussianNB()
12
13    #Train the model using the training sets
14    nb.fit(X_train, Y_train)
15
16    #Predict the response for test dataset
17    Y_pred = nb.predict(X_test)
18
19    if (verbose >= 2):
20        print ("Total Columns (including class)",len(df.columns))
21        print("Number of records for classes ",nb.class_count_)
22
23    cf=confusion_matrix(Y_test, Y_pred)
24    tn, fp, fn, tp=cf.ravel()
25    tpr = tp/(tp+fp)
26    fpr = fp/(fp+tn)
27    fnr = fn/(fn+tp)
28
29    if (verbose >= 2):

```

```
30     print ("Confusion Matrix")
31     print(cf)
32     print("")
33     print ("TP: ", tp,", FP: ", fp,", TN: ", tn,", FN:", fn)
34     print ("TPR: ",tpr,", FPR: ",fpr, "FNR: ",fnr)
35
36     if (verbose >= 2):
37         print(classification_report(Y_test, Y_pred, digits=3))
38
39     accuracy = accuracy_score(Y_test, Y_pred)
40     f1_weighted = f1_score(Y_test, Y_pred,average='weighted')
41
42     if (verbose >= 1):
43         print(f"Accuracy is: {accuracy}")
44         print(f"F1 Weighted is: {f1_weighted}")
45         print("")
46
47     return(accuracy,f1_weighted,tpr,fpr,fnr)


1 def do_LR(df,class_col_name,verbose=0):
2     #not disabling randomness.
3     #np.random.seed(0)
4
5     # Split dataset into training set and test set
6     feature_names=df.columns[df.columns != class_col_name ]
7     # 80% training and 20% test
8     X_train, X_test, Y_train, Y_test = train_test_split(df.loc[:, feature_names], df[class_c
9
10    lr = LogisticRegression(max_iter=2000)
11
12    #Train the model using the training sets
13    lr.fit(X_train, Y_train)
14
15    #Predict the response for test dataset
16    Y_pred = lr.predict(X_test)
17
18    if (verbose >= 2):
19        print ("Total Columns (including class)",len(df.columns))
20
21    cf=confusion_matrix(Y_test, Y_pred)
22    tn, fp, fn, tp=cf.ravel()
23    tpr = tp/(tp+fp)
24    fpr = fp/(fp+tn)
25    fnr = fn/(fn+tp)
26
27    if (verbose >= 2):
28        print ("Confusion Matrix")
29        print(cf)
30        print("")
31        print ("TP: ", tp,", FP: ", fp,", TN: ", tn,", FN:", fn)
```

```
32     print ("TPR: ",tpr,", FPR: ",fpr, "FNR: ",fnr)
33
34     if (verbose >= 2):
35         print(classification_report(Y_test, Y_pred, digits=3))
36
37     accuracy = accuracy_score(Y_test, Y_pred)
38     f1_weighted = f1_score(Y_test, Y_pred,average='weighted')
39
40     if (verbose >= 1):
41         print(f"Accuracy is: {accuracy}")
42         print(f"F1 Weighted is: {f1_weighted}")
43         print("")
44
45     return(accuracy,f1_weighted,tpr,fpr,fnr)


1 def do_KNN(df,class_col_name,verbose=0):
2     #not disabling randomness.
3     #np.random.seed(0)
4
5     # Split dataset into training set and test set
6     feature_names=df.columns[df.columns != class_col_name ]
7     # 80% training and 20% test
8     X_train, X_test, Y_train, Y_test = train_test_split(df.loc[:, feature_names], df[class_c
9
10    knn = KNeighborsClassifier()
11
12    #Train the model using the training sets
13    knn.fit(X_train, Y_train)
14
15    #Predict the response for test dataset
16    Y_pred = knn.predict(X_test)
17
18    if (verbose >= 2):
19        print ("Total Columns (including class)",len(df.columns))
20
21    cf=confusion_matrix(Y_test, Y_pred)
22    tn, fp, fn, tp=cf.ravel()
23    tpr = tp/(tp+fp)
24    fpr = fp/(fp+tn)
25    fnr = fn/(fn+tp)
26
27    if (verbose >= 2):
28        print ("Confusion Matrix")
29        print(cf)
30        print("")
31        print ("TP: ", tp,", FP: ", fp,", TN: ", tn,", FN:", fn)
32        print ("TPR: ",tpr,", FPR: ",fpr, "FNR: ",fnr)
33
34    if (verbose >= 2):
35        print(classification_report(Y_test, Y_pred, digits=3))
```

```
36
37 accuracy = accuracy_score(Y_test, Y_pred)
38 f1_weighted = f1_score(Y_test, Y_pred, average='weighted')
39
40 if (verbose >= 1):
41     print(f"Accuracy is: {accuracy}")
42     print(f"F1 Weighted is: {f1_weighted}")
43     print("")
44
45 return (accuracy, f1_weighted, tpr, fpr, fnr)


1 def do_RF(df, class_col_name, verbose=0):
2     #not disabling randomness.
3     #np.random.seed(0)
4
5     # Split dataset into training set and test set
6     feature_names = df.columns[df.columns != class_col_name]
7     # 80% training and 20% test
8     X_train, X_test, Y_train, Y_test = train_test_split(df.loc[:, feature_names], df[class_c
9
10    rf = RandomForestClassifier()
11
12    #Train the model using the training sets
13    rf.fit(X_train, Y_train)
14
15    #Predict the response for test dataset
16    Y_pred = rf.predict(X_test)
17
18    if (verbose >= 2):
19        print("Total Columns (including class)", len(df.columns))
20
21    cf = confusion_matrix(Y_test, Y_pred)
22    tn, fp, fn, tp = cf.ravel()
23    tpr = tp / (tp + fp)
24    fpr = fp / (fp + tn)
25    fnr = fn / (fn + tp)
26
27    if (verbose >= 2):
28        print("Confusion Matrix")
29        print(cf)
30        print("")
31        print("TP: ", tp, ", FP: ", fp, ", TN: ", tn, ", FN: ", fn)
32        print("TPR: ", tpr, ", FPR: ", fpr, "FNR: ", fnr)
33
34    if (verbose >= 2):
35        print(classification_report(Y_test, Y_pred, digits=3))
36
37    accuracy = accuracy_score(Y_test, Y_pred)
38    f1_weighted = f1_score(Y_test, Y_pred, average='weighted')
39
```

```
40 if (verbose >= 1):
41     print(f"Accuracy is: {accuracy}")
42     print(f"F1 Weighted is: {f1_weighted}")
43     print("")
44
45     return(accuracy,f1_weighted,tpr,fpr,fnr)
46
47
48 def do_RF_tuned(df,class_col_name,verbose=0):
49     #not disabling randomness.
50     #np.random.seed(0)
51
52     # Split dataset into training set and test set
53     feature_names=df.columns[df.columns != class_col_name ]
54     # 80% training and 20% test
55     X_train, X_test, Y_train, Y_test = train_test_split(df.loc[:, feature_names], df[class_c
56
57     rf = RandomForestClassifier(min_samples_leaf = 5, n_estimators = 400)
58
59     #Train the model using the training sets
60     rf.fit(X_train, Y_train)
61
62     #Predict the response for test dataset
63     Y_pred = rf.predict(X_test)
64
65     if (verbose >= 2):
66         print ("Total Columns (including class)",len(df.columns))
67
68     cf=confusion_matrix(Y_test, Y_pred)
69     tn, fp, fn, tp=cf.ravel()
70     tpr = tp/(tp+fp)
71     fpr = fp/(fp+tn)
72     fnr = fn/(fn+tp)
73
74     if (verbose >= 2):
75         print ("Confusion Matrix")
76         print(cf)
77         print("")
78         print ("TP: ", tp,", FP: ", fp,", TN: ", tn,", FN:", fn)
79         print ("TPR: ",tpr,", FPR: ",fpr, "FNR: ",fnr)
80
81     if (verbose >= 2):
82         print(classification_report(Y_test, Y_pred, digits=3))
83
84     accuracy = accuracy_score(Y_test, Y_pred)
85     f1_weighted = f1_score(Y_test, Y_pred,average='weighted')
86
87     if (verbose >= 1):
88         print(f"Accuracy is: {accuracy}")
89         print(f"F1 Weighted is: {f1_weighted}")
90         print("")
```

```
91
92  return(accuracy,f1_weighted,tpr,fpr,fnr)

1 def do_SVM(df,class_col_name,verbose=0):
2     #not disabling randomness.
3     #np.random.seed(0)
4
5     # Split dataset into training set and test set
6     feature_names=df.columns[df.columns != class_col_name ]
7     # 80% training and 20% test
8     X_train, X_test, Y_train, Y_test = train_test_split(df.loc[:, feature_names], df[class_c
9
10    svm = SVC()
11
12    #Train the model using the training sets
13    svm.fit(X_train, Y_train)
14
15    #Predict the response for test dataset
16    Y_pred = svm.predict(X_test)
17
18    if (verbose >= 2):
19        print ("Total Columns (including class)",len(df.columns))
20
21    cf=confusion_matrix(Y_test, Y_pred)
22    tn, fp, fn, tp=cf.ravel()
23    tpr = tp/(tp+fp)
24    fpr = fp/(fp+tn)
25    fnr = fn/(fn+tp)
26
27
28    if (verbose >= 2):
29        print ("Confusion Matrix")
30        print(cf)
31        print("")
32        print ("TP: ", tp,", FP: ", fp,", TN: ", tn,", FN:", fn)
33        print ("TPR: ",tpr,", FPR: ",fpr, "FNR: ",fnr)
34
35    if (verbose >= 2):
36        print(classification_report(Y_test, Y_pred, digits=3))
37
38    accuracy = accuracy_score(Y_test, Y_pred)
39    f1_weighted = f1_score(Y_test, Y_pred,average='weighted')
40
41    if (verbose >= 1):
42        print(f"Accuracy is: {accuracy}")
43        print(f"F1 Weighted is: {f1_weighted}")
44        print("")
45
46    return(accuracy,f1_weighted,tpr,fpr,fnr)
```



```
1 def do_XGB(df,class_col_name,verbose=0):
2     #not disabling randomness.
3     #np.random.seed(0)
4
5     # Split dataset into training set and test set
6     feature_names=df.columns[df.columns != class_col_name ]
7     # 80% training and 20% test
8     X_train, X_test, Y_train, Y_test = train_test_split(df.loc[:, feature_names], df[class_c
9
10    xgb = XGBClassifier()
11
12    #Train the model using the training sets
13    xgb.fit(X_train, Y_train)
14
15    #Predict the response for test dataset
16    Y_pred = xgb.predict(X_test)
17
18    if (verbose >= 2):
19        print ("Total Columns (including class)",len(df.columns))
20
21    cf=confusion_matrix(Y_test, Y_pred)
22    tn, fp, fn, tp=cf.ravel()
23    tpr = tp/(tp+fp)
24    fpr = fp/(fp+tn)
25    fnr = fn/(fn+tp)
26
27    if (verbose >= 2):
28        print ("Confusion Matrix")
29        print(cf)
30        print("")
31        print ("TP: ", tp,", FP: ", fp,", TN: ", tn,", FN:", fn)
32        print ("TPR: ",tpr,", FPR: ",fpr, "FNR: ",fnr)
33
34    if (verbose >= 2):
35        print(classification_report(Y_test, Y_pred, digits=3))
36
37    accuracy = accuracy_score(Y_test, Y_pred)
38    f1_weighted = f1_score(Y_test, Y_pred,average='weighted')
39
40    if (verbose >= 1):
41        print(f"Accuracy is: {accuracy}")
42        print(f"F1 Weighted is: {f1_weighted}")
43        print("")
44
45    return(accuracy,f1_weighted,tpr,fpr,fnr)
46
47 def do_XGB_tuned(df,class_col_name,verbose=0):
48     #not disabling randomness.
49     #np.random.seed(0)
50
51     # Split dataset into training set and test set
```

```

52 feature_names=df.columns[df.columns != class_col_name ]
53 # 80% training and 20% test
54 X_train, X_test, Y_train, Y_test = train_test_split(df.loc[:, feature_names], df[class_c
55
56 xgb = XGBClassifier(n_estimators = 500, learning_rate = 0.1)
57
58 #Train the model using the training sets
59 xgb.fit(X_train, Y_train)
60
61 #Predict the response for test dataset
62 Y_pred = xgb.predict(X_test)
63
64 if (verbose >= 2):
65     print ("Total Columns (including class)",len(df.columns))
66
67 cf=confusion_matrix(Y_test, Y_pred)
68 tn, fp, fn, tp=cf.ravel()
69 tpr = tp/(tp+fp)
70 fpr = fp/(fp+tn)
71 fnr = fn/(fn+tp)
72
73 if (verbose >= 2):
74     print ("Confusion Matrix")
75     print(cf)
76     print("")
77     print ("TP: ", tp,", FP: ", fp,", TN: ", tn,", FN:", fn)
78     print ("TPR: ",tpr,", FPR: ",fpr, "FNR: ",fnr)
79
80 if (verbose >= 2):
81     print(classification_report(Y_test, Y_pred, digits=3))
82
83 accuracy = accuracy_score(Y_test, Y_pred)
84 f1_weighted = f1_score(Y_test, Y_pred,average='weighted')
85
86 if (verbose >= 1):
87     print(f"Accuracy is: {accuracy}")
88     print(f"F1 Weighted is: {f1_weighted}")
89     print("")
90
91 return(accuracy,f1_weighted,tpr,fpr,fnr)

```

Initial Run of All ML Algorithms:

```

1 #Initial Run of all ML Algorithms just to make sure everything works correctly.
2 #Using original data:
3
4 for i in range(3,11):
5     print(f"DT with {i} levels:")
6     do_DT(df,i,'Target',5)

```

```

7
8 print(f"MN_NB:")
9 do_mnNB(df_no_negs, 'Target', 5)
10
11 print(f"GA_NB:")
12 do_gaNB(df, 'Target', 5)
13
14 print(f"LR:")
15 do_LR(df, 'Target', 5)
16
17 print(f"KNN:")
18 do_KNN(df, 'Target', 5)
19
20 print(f"RF:")
21 do_RF(df, 'Target', 5)
22
23 print(f"SVM:")
24 do_SVM(df, 'Target', 5)
25
26 print(f"XGB:")
27 do_XGB(df, 'Target', 5)

```

DT with 3 levels:

Successfully 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:

Successfully 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:
 Successfully 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:

```

1 #Decided against cross-fold validation, and instead using an iterative approach.
2 #In lieu of cross-fold validation, 100 random 80/20 splits of the data-set were used, and
3 #average TPR, average TNR, and average FNR were calculated over the 100 iterations.
4 #
5 #Once the initial validation is performed, the iterative approach will be extended to 1000
6
7 #The reason for this approach versus the more common, professional, cross-fold validation
8 #This approach is possible due to fact the data-set is relatively small, making each itera
9 #This approach would not be possible using very large data-sets.
10
11
12 #Original Cross-fold validation code (no longer used)
13 #cv = KFold(n_splits=10, random_state=1, shuffle=True)
14 #lr = LogisticRegression(max_iter=2000)
15 #nb = GaussianNB()
16 #rf = RandomForestClassifier()
17 #xgb = XGBClassifier()
18 #
19 #X = df.drop('Target',axis=1)
20 #Y = df['Target']
21 #
22 #LR_acc = cross_val_score(lr, X, Y, scoring='accuracy', cv=cv, n_jobs=-1)
23 #NB_acc = cross_val_score(nb, X, Y, scoring='accuracy', cv=cv, n_jobs=-1)

```

```
24 #RFTree_acc = cross_val_score(rf, X, Y, scoring='accuracy', cv=cv, n_jobs=-1)
25 #XGB_acc = cross_val_score(xgb, X, Y, scoring='accuracy', cv=cv, n_jobs=-1)
26 #
27 #
28 #print('Accuracy after Cross-Val - Logistic Regression(LR): %.4f (%.4f)' % (mean(LR_acc),
29 #print('Accuracy after Cross-Val - Gaussian Naive Bayes(gaNB): %.4f (%.4f)' % (mean(NB_acc
30 #print('Accuracy after Cross-Val - Random Forest(RF): %.4f (%.4f)' % (mean(RFTree_acc), st
31 #print('Accuracy after Cross-Val - XGBoost(XGB): %.4f (%.4f)' % (mean(XGB_acc), stdev(XGB_
32 #
33 #
34
35
36 #iterative approach methods:
37 def getTestAccuracy(x):
38     value = results[f"TEST{x}#AVERAGE_ACCURACY"]
39     return(value)
40
41 def getTestF1(x):
42     value = results[f"TEST{x}#AVERAGE_F1_WEIGHTED"]
43     return(value)
44
45 def getTestFNR(x):
46     value = results[f"TEST{x}#AVERAGE_FNR"]
47     return(value)
48
49 def displayResult(x):
50     name = results[f"TEST{x}#NAME"]
51     df_name = results[f"TEST{x}#DFNAME"]
52     accuracy = results[f"TEST{x}#AVERAGE_ACCURACY"]*100
53     f1_weighted = results[f"TEST{x}#AVERAGE_F1_WEIGHTED"]*100
54     tpr = results[f"TEST{x}#AVERAGE_TPR"]*100
55     fpr = results[f"TEST{x}#AVERAGE_FPR"]*100
56     fnr = results[f"TEST{x}#AVERAGE_FNR"]*100
57
58     print(f"Using ML Model: {name} and {df_name}:")
59     print(f"Average Accuracy is {accuracy:.2f}%, Average F1(weighted) is {f1_weighted:.2f}%",
60     print("")
61
62 def iterativeValidation(test,df,iterations,verbose=0):
63     if verbose > 0:
64         print("")
65         print("")
66         print("")
67
68     df_name = df.name
69
70     if test == "MN_NB":
71         test_name = "Multinomial Naive Bayes(MN-NB)"
72     elif test == "GA_NB":
73         test_name = "Gaussian Naive Bayes(GA-NB)"
74     elif test == "LR":
```

```
75     test_name = "Logistic Regression(LR)"
76 elif test == "SVM":
77     test_name = "Support Vector Machines(SVM)"
78 elif test == "KNN":
79     test_name = "K Nearest Neighbours(KNN)"
80 elif test == "RF":
81     test_name = "Random Forest(RF)"
82 elif test == "RF_TUNED":
83     test_name = "Random Forest(RF) Tuned"
84 elif test == "XGB":
85     test_name = "XG Boost(XGB)"
86 elif test == "XGB_TUNED":
87     test_name = "XG Boost(XGB) Tuned"
88 elif test == "DT_3":
89     test_name = "Decision Tree: 3 levels"
90 elif test == "DT_4":
91     test_name = "Decision Tree: 4 levels"
92 elif test == "DT_5":
93     test_name = "Decision Tree: 5 levels"
94 elif test == "DT_6":
95     test_name = "Decision Tree: 6 levels"
96 elif test == "DT_7":
97     test_name = "Decision Tree: 7 levels"
98 elif test == "DT_8":
99     test_name = "Decision Tree: 8 levels"
100 elif test == "DT_9":
101     test_name = "Decision Tree: 9 levels"
102 elif test == "DT_10":
103     test_name = "Decision Tree: 10 levels"
104
105 if verbose > 0:
106     print(f"Performing {test_name} Analysis:")
107 test_num=results["NEXT_TEST"]
108
109 accuracy_sum = 0
110 f1_weighted_sum = 0
111 tpr_sum = 0
112 fpr_sum = 0
113 fnr_sum = 0
114
115 for n in range(iterations):
116     if test == "MN_NB":
117         result = do_mnNB(df, 'Target', 0)
118     elif test == "GA_NB":
119         result = do_gaNB(df, 'Target', 0)
120     elif test == "LR":
121         result = do_LR(df, 'Target', 0)
122     elif test == "SVM":
123         result = do_SVM(df, 'Target', 0)
124     elif test == "KNN":
125         result = do_KNN(df, 'Target', 0)
```

```
126     elif test == "RF_TUNED":
127         result = do_RF(df, 'Target', 0)
128     elif test == "RF":
129         result = do_RF_tuned(df, 'Target', 0)
130     elif test == "XGB":
131         result = do_XGB(df, 'Target', 0)
132     elif test == "XGB_TUNED":
133         result = do_XGB_tuned(df, 'Target', 0)
134     elif test == "DT_3":
135         result = do_DT(df, 3, 'Target', 0)
136     elif test == "DT_4":
137         result = do_DT(df, 4, 'Target', 0)
138     elif test == "DT_5":
139         result = do_DT(df, 5, 'Target', 0)
140     elif test == "DT_6":
141         result = do_DT(df, 6, 'Target', 0)
142     elif test == "DT_7":
143         result = do_DT(df, 7, 'Target', 0)
144     elif test == "DT_8":
145         result = do_DT(df, 8, 'Target', 0)
146     elif test == "DT_9":
147         result = do_DT(df, 9, 'Target', 0)
148     elif test == "DT_10":
149         result = do_DT(df, 10, 'Target', 0)
150
151     accuracy_sum += result[0]
152     f1_weighted_sum += result[1]
153     tpr_sum += result[2]
154     fpr_sum += result[3]
155     fnr_sum += result[4]
156
157     results[f"TEST{test_num}#NAME"] = test_name
158     results[f"TEST{test_num}#DFNAME"] = df.name
159     results[f"TEST{test_num}#AVERAGE_ACCURACY"] = accuracy_sum/iterations
160     results[f"TEST{test_num}#AVERAGE_F1_WEIGHTED"] = f1_weighted_sum/iterations
161     results[f"TEST{test_num}#AVERAGE_TPR"] = tpr_sum/iterations
162     results[f"TEST{test_num}#AVERAGE_FPR"] = fpr_sum/iterations
163     results[f"TEST{test_num}#AVERAGE_FNR"] = fnr_sum/iterations
164     results[f"LAST_TEST"] = test_num
165     results[f"NEXT_TEST"] += 1
166
167
168     name = results[f"TEST{test_num}#NAME"]
169     df_name = results[f"TEST{test_num}#DFNAME"]
170     accuracy = results[f"TEST{test_num}#AVERAGE_ACCURACY"]*100
171     f1_weighted = results[f"TEST{test_num}#AVERAGE_F1_WEIGHTED"]*100
172     tpr = results[f"TEST{test_num}#AVERAGE_TPR"]*100
173     fpr = results[f"TEST{test_num}#AVERAGE_FPR"]*100
174     fnr = results[f"TEST{test_num}#AVERAGE_FNR"]*100
175
176     if verbose > 0:
```

```

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

```

Initial Validation:

```

1 #Initial Validation tests. (100 iterations, averaged, all models, all dataset combinations
2 iterations = 100
3 results = {}
4 results["NEXT_TEST"] = 0
5
6 iterativeValidation("DT_3",df,iterations)
7 iterativeValidation("DT_4",df,iterations)
8 iterativeValidation("DT_5",df,iterations)
9 iterativeValidation("DT_6",df,iterations)
10 iterativeValidation("DT_7",df,iterations)
11 iterativeValidation("DT_8",df,iterations)
12 iterativeValidation("DT_9",df,iterations)
13 iterativeValidation("DT_10",df,iterations)
14
15 iterativeValidation("MN_NB",df_no_negs,iterations)
16 iterativeValidation("GA_NB",df,iterations)
17 iterativeValidation("LR",df,iterations)
18 iterativeValidation("KNN",df,iterations)
19 iterativeValidation("SVM",df,iterations)
20 iterativeValidation("RF",df,iterations)
21 iterativeValidation("XGB",df,iterations)
22
23 iterativeValidation("DT_3",df_onehot,iterations)
24 iterativeValidation("DT_4",df_onehot,iterations)
25 iterativeValidation("DT_5",df_onehot,iterations)
26 iterativeValidation("DT_6",df_onehot,iterations)
27 iterativeValidation("DT_7",df_onehot,iterations)
28 iterativeValidation("DT_8",df_onehot,iterations)
29 iterativeValidation("DT_9",df_onehot,iterations)
30 iterativeValidation("DT_10",df_onehot,iterations)
31
32 iterativeValidation("MN_NB",df_onehot_no_negs,iterations)
33 iterativeValidation("GA_NB",df_onehot,iterations)
34 iterativeValidation("LR",df_onehot,iterations)
35 iterativeValidation("KNN",df_onehot,iterations)
36 iterativeValidation("SVM",df_onehot,iterations)
37 iterativeValidation("RF",df_onehot,iterations)
38 iterativeValidation("XGB",df_onehot,iterations)
39
40
41 iterativeValidation("DT_3",df_outliers_addressed,iterations)
42 iterativeValidation("DT_4",df_outliers_addressed,iterations)
43 iterativeValidation("DT_5",df_outliers_addressed,iterations)
44 iterativeValidation("DT_6",df_outliers_addressed,iterations)

```



```
44 iterativeValidation("DT_6",df_outliers_addressed,iterations)
45 iterativeValidation("DT_7",df_outliers_addressed,iterations)
46 iterativeValidation("DT_8",df_outliers_addressed,iterations)
47 iterativeValidation("DT_9",df_outliers_addressed,iterations)
48 iterativeValidation("DT_10",df_outliers_addressed,iterations)
49
50 iterativeValidation("MN_NB",df_no_negs,iterations)
51 iterativeValidation("GA_NB",df_outliers_addressed,iterations)
52 iterativeValidation("LR",df_outliers_addressed,iterations)
53 iterativeValidation("KNN",df_outliers_addressed,iterations)
54 iterativeValidation("SVM",df_outliers_addressed,iterations)
55 iterativeValidation("RF",df_outliers_addressed,iterations)
56 iterativeValidation("XGB",df_outliers_addressed,iterations)
57
58 iterativeValidation("DT_3",df_normalized,iterations)
59 iterativeValidation("DT_4",df_normalized,iterations)
60 iterativeValidation("DT_5",df_normalized,iterations)
61 iterativeValidation("DT_6",df_normalized,iterations)
62 iterativeValidation("DT_7",df_normalized,iterations)
63 iterativeValidation("DT_8",df_normalized,iterations)
64 iterativeValidation("DT_9",df_normalized,iterations)
65 iterativeValidation("DT_10",df_normalized,iterations)
66
67 iterativeValidation("MN_NB",df_normalized,iterations)
68 iterativeValidation("GA_NB",df_normalized,iterations)
69 iterativeValidation("LR",df_normalized,iterations)
70 iterativeValidation("KNN",df_normalized,iterations)
71 iterativeValidation("SVM",df_normalized,iterations)
72 iterativeValidation("RF",df_normalized,iterations)
73 iterativeValidation("XGB",df_normalized,iterations)

1 #Collate initial results
2 range_limit = min(10,results[f"LAST_TEST"]) #Top 10 results desired.
3
4 results_list = list(range(0,results[f"LAST_TEST"]+1))
5 results_list.sort(key=getTestAccuracy, reverse=True)
6
7 print("Results of ML Models: (sorted by accuracy)")
8 print("")
9 for i in results_list[0:range_limit]: #Count starts at zero, so dont need to add one.
10     displayResult(i)
11 print("")
12 print("")
13
14
15 results_list = list(range(0,results[f"LAST_TEST"]+1))
16 results_list.sort(key=getTestFNR, reverse=False)
17
18 print("Results of ML Models: (sorted by False Negative Rate(FNR))")
19 print("")
20 for i in results_list[0:range_limit]: #Count starts at zero, so dont need to add one.
```

```
21 displayResult(i)
22 print("")
23 print("")
24
25
26
27 #View these results again, this time just the top 5 using each sort method:
28 range_limit = min(5,results[f"LAST_TEST"]) #Top 5 results desired.
29
30 results_list = list(range(0,results[f"LAST_TEST"]+1))
31 results_list.sort(key=getTestAccuracy, reverse=True)
32
33 print("Results of ML Models: (sorted by accuracy) (top 5)")
34 print("")
35 for i in results_list[0:range_limit]: #Count starts at zero, so dont need to add one.
36     displayResult(i)
37     print("")
38     print("")
39
40
41 results_list = list(range(0,results[f"LAST_TEST"]+1))
42 results_list.sort(key=getTestFNR, reverse=False)
43
44 print("Results of ML Models: (sorted by False Negative Rate(FNR)) (top 5)")
45 print("")
46 for i in results_list[0:range_limit]: #Count starts at zero, so dont need to add one.
47     displayResult(i)
48     print("")
49     print("")
50
51
```

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:

```
1 #Secondary Validation, 1000 iterations, on top models only)
2 iterations = 1000
3 results = {}
4 results["NEXT_TEST"] = 0
5
6 iterativeValidation("XGB",df_normalized,iterations)
7 iterativeValidation("XGB",df_outliers_addressed,iterations)
8 iterativeValidation("XGB",df,iterations)
9 iterativeValidation("XGB",df_onehot,iterations)
10
11 iterativeValidation("RF",df_onehot,iterations)
12 iterativeValidation("RF",df,iterations)
13 iterativeValidation("RF",df_outliers_addressed,iterations)
```

```

14 iterativeValidation("RF",df_normalized,iterations)
15
16

1 #Collate secondary results
2 range_limit = min(3,results[f"LAST_TEST"]) #Top 3 results desired.
3
4 results_list = list(range(0,results[f"LAST_TEST"]+1))
5 results_list.sort(key=getTestAccuracy, reverse=True)
6
7 print("Results of ML Models: (sorted by accuracy)")
8 print("")
9 for i in results_list[0:range_limit]: #Count starts at zero, so dont need to add one.
10     displayResult(i)
11 print("")
12 print("")
13
14
15 results_list = list(range(0,results[f"LAST_TEST"]+1))
16 results_list.sort(key=getTestFNR, reverse=False)
17
18 print("Results of ML Models: (sorted by False Negative Rate(FNR))")
19 print("")
20 for i in results_list[0:range_limit]: #Count starts at zero, so dont need to add one.
21     displayResult(i)
22 print("")
23 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%, Average

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%, Average

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%, Average

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%, Average

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%, Average

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%, Average

Hypertuning:

```

1 rf = RandomForestClassifier()
2 xgb = XGBClassifier()
3 feature_names=df_onehot.columns[df_onehot.columns != "Target"]
4 # 80% training and 20% test
5 X_train, X_test, Y_train, Y_test = train_test_split(df_onehot.loc[:, feature_names], df_on
6
7
8 params_rf = {'n_estimators':[100,200,300,400,500], 'min_samples_leaf':[5, 10, 15, 20, 25,
9 grid_rf = GridSearchCV(rf, param_grid=params_rf, cv=10)
10 grid_rf.fit(X_train, Y_train)
11 print("Hyper-Tuned Parameters for Random Forest:", grid_rf.best_params_)
12
13
14 params_xgb = {'n_estimators': [100,200,300,400,500,600,700,800,900,1000], 'learning_rate':
15 rs_xgb = RandomizedSearchCV(xgb, param_distributions=params_xgb, cv=10)
16 rs_xgb.fit(X_train, Y_train)
17 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}

```

```

1 #Tertiary Validation(after hypertuning), 1000 iterations, on top models only)
2 iterations = 1000
3 results = {}
4 results["NEXT_TEST"] = 0
5
6 iterativeValidation("XGB_TUNED",df_normalized,iterations)
7 iterativeValidation("XGB_TUNED",df_outliers_addressed,iterations)
8 iterativeValidation("XGB_TUNED",df,iterations)
9 iterativeValidation("XGB_TUNED",df_onehot,iterations)
10
11 iterativeValidation("RF_TUNED",df_onehot,iterations)
12 iterativeValidation("RF_TUNED",df,iterations)
13 iterativeValidation("RF_TUNED",df_outliers_addressed,iterations)
14 iterativeValidation("RF_TUNED",df_normalized,iterations)

```

Tertiary Validation (After Hypertuning):

```

1 #Collate tertiary results
2 range_limit = min(3,results[f"LAST_TEST"]) #Top 3 results desired.
3
4 results_list = list(range(0,results[f"LAST_TEST"]+1))

```

```
5 results_list.sort(key=getTestAccuracy, reverse=True)
6
7 print("Results of ML Models: (sorted by accuracy)")
8 print("")
9 for i in results_list[0:range_limit]: #Count starts at zero, so dont need to add one.
10     displayResult(i)
11 print("")
12 print("")
13
14
15 results_list = list(range(0,results[f"LAST_TEST")+1))
16 results_list.sort(key=getTestFNR, reverse=False)
17
18 print("Results of ML Models: (sorted by False Negative Rate(FNR))")
19 print("")
20 for i in results_list[0:range_limit]: #Count starts at zero, so dont need to add one.
21     displayResult(i)
22 print("")
23 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%, Average

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%, Average

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%, Average

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%, Average

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%, Average

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%, Average



Colab paid products - Cancel contracts here

✓ 0s completed at 9:06 PM

