Libraries

In [1]:

```
import warnings
#data manipulation
import pandas as pd
import numpy as np
#data visualization
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt
import seaborn as sns
#normalization
from sklearn.preprocessing import MinMaxScaler
#machine learning algorithm
from sklearn.model_selection import train_test_split,GridSearchCV
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, roc_
curve, auc, precision_score, recall_score, f1_score, roc_auc_score
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import GradientBoostingClassifier
```

In [2]:

```
warnings.filterwarnings('ignore', category=FutureWarning)
```

Data Prepation

In [3]:

```
#read data &check
df=pd.read_csv('/kaggle/input/framingham-heart-study/framingham_heart_study.csv')
df.head()
```

Out[3]:

	male	age	education	currentSmoker	cigsPerDay	BPMeds	prevalentStroke	prevalentHyp	diabetes	totChol	sysBP	diaBl
0	1	39	4.0	0	0.0	0.0	0	0	0	195.0	106.0	70.
1	0	46	2.0	0	0.0	0.0	0	0	0	250.0	121.0	81.
2	1	48	1.0	1	20.0	0.0	0	0	0	245.0	127.5	80.
3	0	61	3.0	1	30.0	0.0	0	1	0	225.0	150.0	95.
4	0	46	3.0	1	23.0	0.0	0	0	0	285.0	130.0	84.

In [4]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4240 entries, 0 to 4239
Data columns (total 16 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   male             4240 non-null   int64  
 1   age              4240 non-null   int64  
 2   education        4135 non-null   float64
 3   currentSmoker    4240 non-null   int64  
 4   cigsPerDay       4211 non-null   float64
 5   BPMeds           4187 non-null   float64
 6   prevalentStroke  4240 non-null   int64
```

```
 6   prevalentStroke   4240 non-null   int64
 7   prevalentHyp      4240 non-null   int64
 8   diabetes          4240 non-null   int64
 9   totChol           4190 non-null   float64
 10  sysBP             4240 non-null   float64
 11  diaBP             4240 non-null   float64
 12  BMI               4221 non-null   float64
 13  heartRate         4239 non-null   float64
 14  glucose           3852 non-null   float64
 15  TenYearCHD        4240 non-null   int64
dtypes: float64(9), int64(7)
memory usage: 530.1 KB
```

With that previous block, we can see that all of our data are numerical data.

Data Cleaning

In [5]:

```
df.columns
```

Out[5]:

```
Index(['male', 'age', 'education', 'currentSmoker', 'cigsPerDay', 'BPMeds',
       'prevalentStroke', 'prevalentHyp', 'diabetes', 'totChol', 'sysBP',
       'diaBP', 'BMI', 'heartRate', 'glucose', 'TenYearCHD'],
      dtype='object')
```

In [6]:

```
new_col_names={'male':'gender','currentSmoker':'is_current_smoker','cigsPerDay':'cigs_per
_day','BPMed':'use_blood_pressure_medication','prevalentStroke':'had_stroke',
               'prevalentHyp':'had_hypertension','diabetes':'has_diabetes','totChol':'tot
al_cholesterol','sysBP':'systolic_bp'
               ,'diaBP':'diastolic_bp','BMI':'bmi','heartRate':'heart_rate',
               'TenYearCHD':'ten_year_chd'}
df.rename(columns=new_col_names,inplace=True)
df.columns
```

Out[6]:

```
Index(['gender', 'age', 'education', 'is_current_smoker', 'cigs_per_day',
       'BPMeds', 'had_stroke', 'had_hypertension', 'has_diabetes',
       'total_cholesterol', 'systolic_bp', 'diastolic_bp', 'bmi', 'heart_rate',
       'glucose', 'ten_year_chd'],
      dtype='object')
```

In [7]:

```
#check null values
df.isna().sum()
```

Out[7]:

```
gender                 0
age                    0
education            105
is_current_smoker      0
cigs_per_day          29
BPMeds                53
had_stroke             0
had_hypertension       0
has_diabetes           0
total_cholesterol     50
systolic_bp            0
diastolic_bp           0
bmi                   19
heart_rate             1
glucose              388
ten_year_chd           0
dtype: int64
```

In [8]:

```
df.dropna(inplace=True)
df.isna().sum()
```

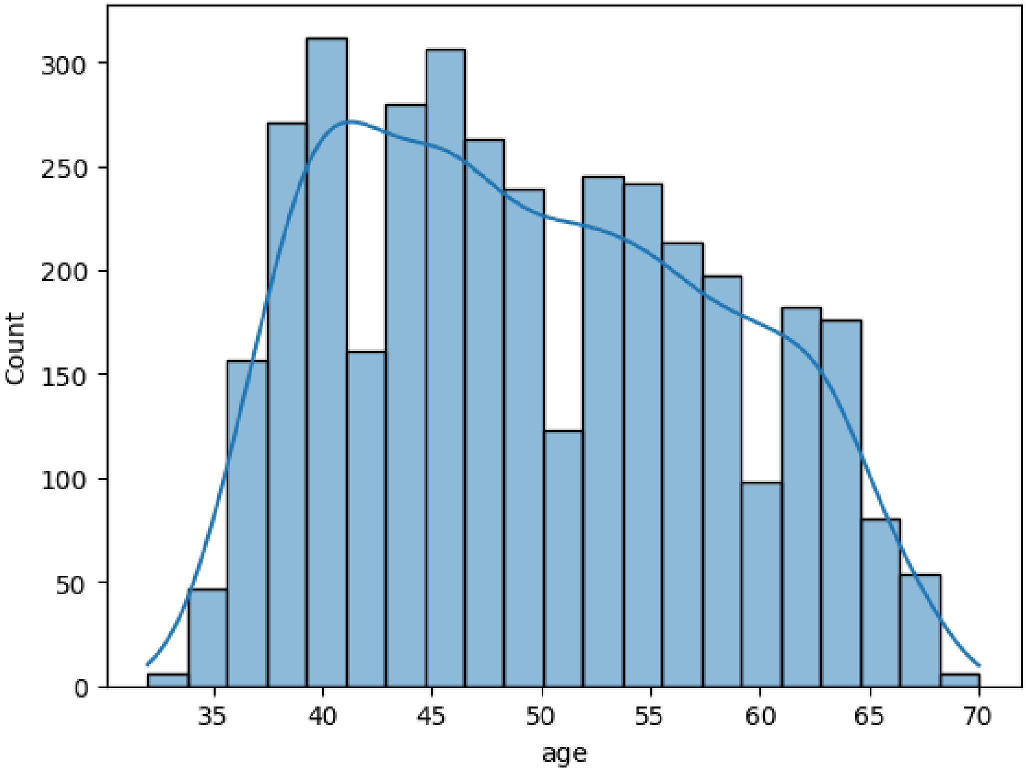
Out[8]:

```
gender               0
age                  0
education            0
is_current_smoker    0
cigs_per_day         0
BPMeds               0
had_stroke           0
had_hypertension     0
has_diabetes         0
total_cholesterol    0
systolic_bp          0
diastolic_bp         0
bmi                  0
heart_rate           0
glucose              0
ten_year_chd         0
dtype: int64
```

Data Visualization

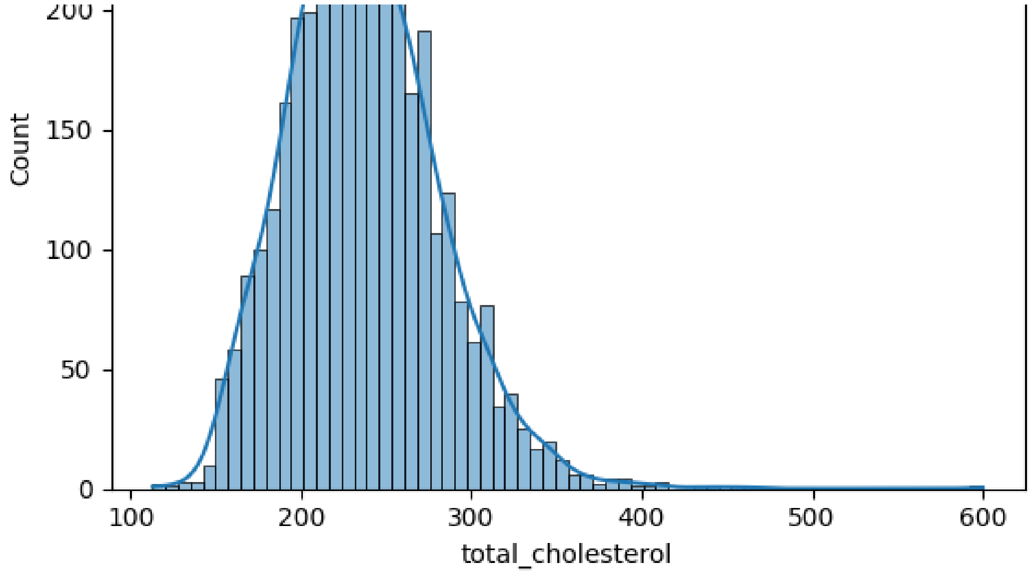
In [9]:

```
sns.histplot(data=df['age'],kde=True)
plt.show()
```



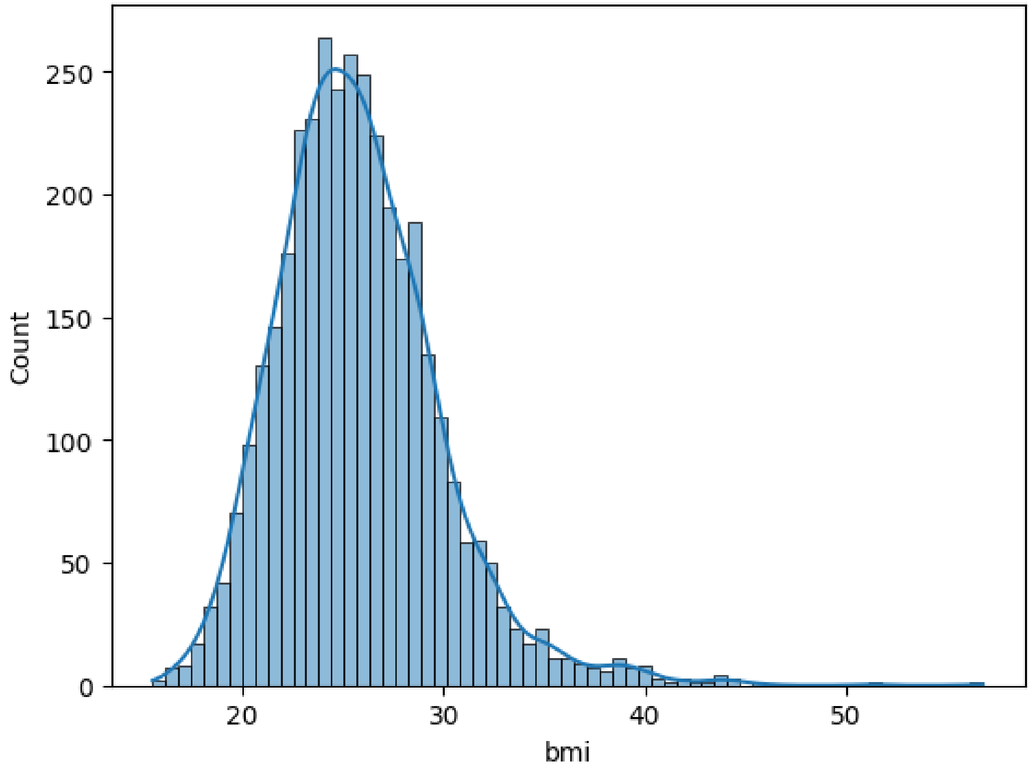
In [10]:

```
sns.histplot(data=df['total_cholesterol'],kde=True)
plt.show()
```



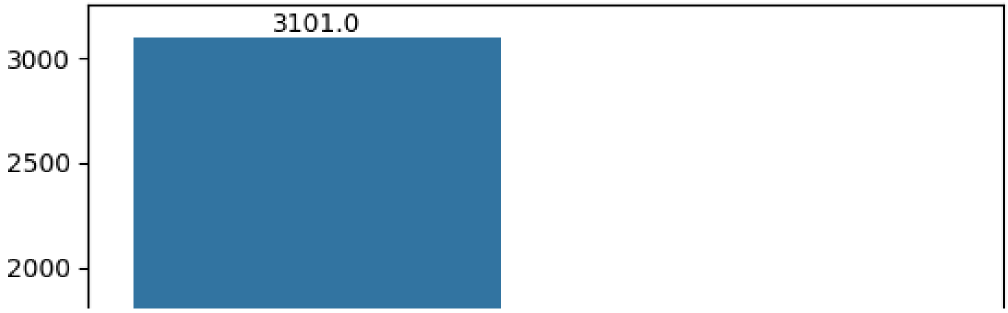
In [11]:

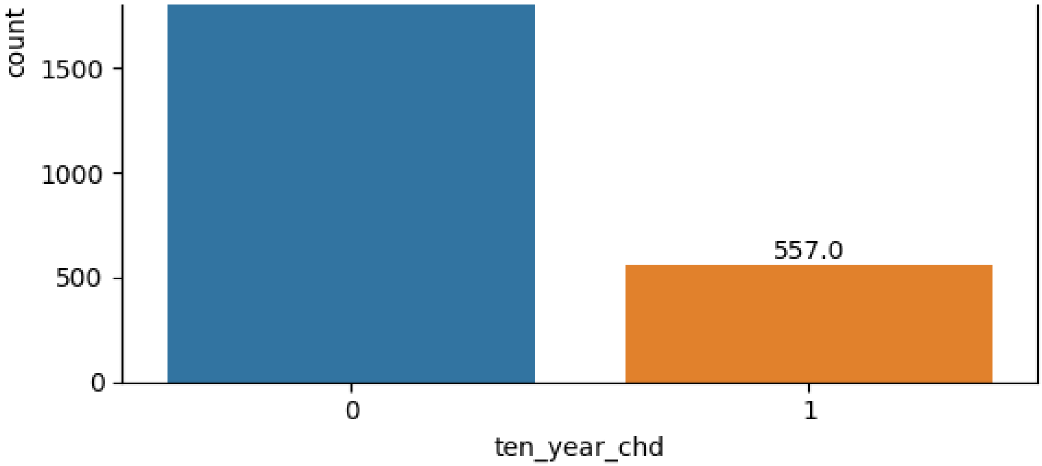
```
sns.histplot(x='bmi',data=df,kde=True)
plt.show()
```



In [12]:

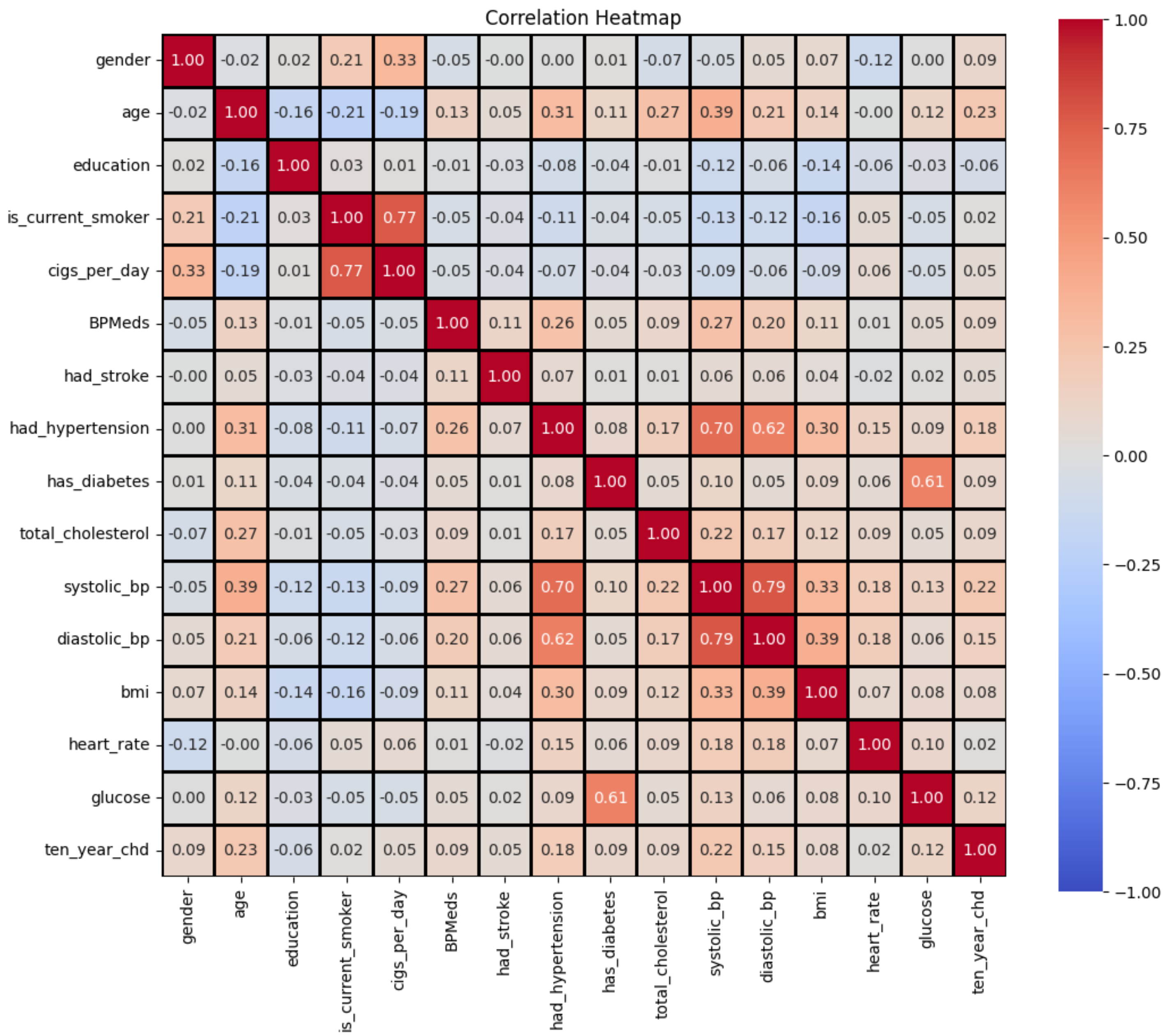
```
ax=sns.countplot(x='ten_year_chd',data=df)
for p in ax.patches:
    ax.annotate(f'{p.get_height()}',
                (p.get_x() + p.get_width() / 2., p.get_height()),
                ha='center',
                va='bottom')
plt.show()
```





In [13]:

```
#correlation heatmap
plt.figure(figsize=(12,10))
correlation_matrix=df.corr()
sns.heatmap(data=correlation_matrix,annot=True,cmap='coolwarm',vmin=-1,vmax=1,linewidths=
2,fmt='.2f',linecolor='black',square=True)
plt.title('Correlation Heatmap')
plt.show()
```



Identifying Outliers

In [14]:

```
# Assuming `df` is your DataFrame
# Calculate IQR
Q1 = df.quantile(0.25)
Q3 = df.quantile(0.75)
IQR = Q3 - Q1

# Define lower and upper bounds
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Cap the outliers while keeping values of 1 unchanged
for column in df.columns:
    df[column] = df[column].where(
        (df[column] >= lower_bound[column]) & (df[column] <= upper_bound[column]) | (df[
column] == 1),
        other=df[column].clip(lower=lower_bound[column], upper=upper_bound[column])
    )

# Display the first 10 rows of the modified DataFrame
df.head(10)
```

Out[14]:

	gender	age	education	is_current_smoker	cigs_per_day	BPMeds	had_stroke	had_hypertension	has_diabetes	total_chole
0	1	39	4.0	0	0.0	0.0	0	0	0	
1	0	46	2.0	0	0.0	0.0	0	0	0	
2	1	48	1.0	1	20.0	0.0	0	0	0	
3	0	61	3.0	1	30.0	0.0	0	1	0	
4	0	46	3.0	1	23.0	0.0	0	0	0	
5	0	43	2.0	0	0.0	0.0	0	1	0	
6	0	63	1.0	0	0.0	0.0	0	0	0	
7	0	45	2.0	1	20.0	0.0	0	0	0	
8	1	52	1.0	0	0.0	0.0	0	1	0	
9	1	43	1.0	1	30.0	0.0	0	1	0	

Normalization

In order to enhance the machine learning algorithm, we are going to use the min-max normalization method. This technique reduces the data to a range of [0, 1].

In [15]:

```
scaler=MinMaxScaler()
normalized_data=pd.DataFrame(scaler.fit_transform(df),columns=df.columns)
normalized_data.head(10)
```

Out[15]:

	gender	age	education	is_current_smoker	cigs_per_day	BPMeds	had_stroke	had_hypertension	has_diabetes	total_
0	1.0	0.184211	1.000000	0.0	0.00	0.0	0.0	0.0	0.0	
1	0.0	0.368421	0.333333	0.0	0.00	0.0	0.0	0.0	0.0	
2	1.0	0.421053	0.000000	1.0	0.40	0.0	0.0	0.0	0.0	
3	0.0	0.763158	0.666667	1.0	0.60	0.0	0.0	1.0	0.0	
4	0.0	0.368421	0.666667	1.0	0.46	0.0	0.0	0.0	0.0	
5	0.0	0.289474	0.333333	0.0	0.00	0.0	0.0	1.0	0.0	

	gender	age	education	is_current_smoker	cigs_per_day	BPMeds	had_stroke	had_hypertension	has_diabetes	total_
6	0.0	0.815789	0.000000	0.0	0.00	0.0	0.0	0.0	0.0	
7	0.0	0.342105	0.333333	1.0	0.40	0.0	0.0	0.0	0.0	
8	1.0	0.526316	0.000000	0.0	0.00	0.0	0.0	1.0	0.0	
9	1.0	0.289474	0.000000	1.0	0.60	0.0	0.0	1.0	0.0	

Train-Test Split

In this section for ML algorithms to learn, we divide the data into train and test sets, but first we select our features and label the data.

In [16]:

```
X=normalized_data.loc[:'ten_year_chd'].drop(columns='ten_year_chd')#features
y=normalized_data['ten_year_chd']#label
print(f'feature names: {X.columns} \n, shape:{X.shape}')
print(f'label name: {y.name}, shape:{y.shape}')
```

```
feature names: Index(['gender', 'age', 'education', 'is_current_smoker', 'cigs_per_day',
       'BPMeds', 'had_stroke', 'had_hypertension', 'has_diabetes',
       'total_cholesterol', 'systolic_bp', 'diastolic_bp', 'bmi', 'heart_rate',
       'glucose'],
      dtype='object')
, shape:(3658, 15)
label name: ten_year_chd, shape:(3658,)
```

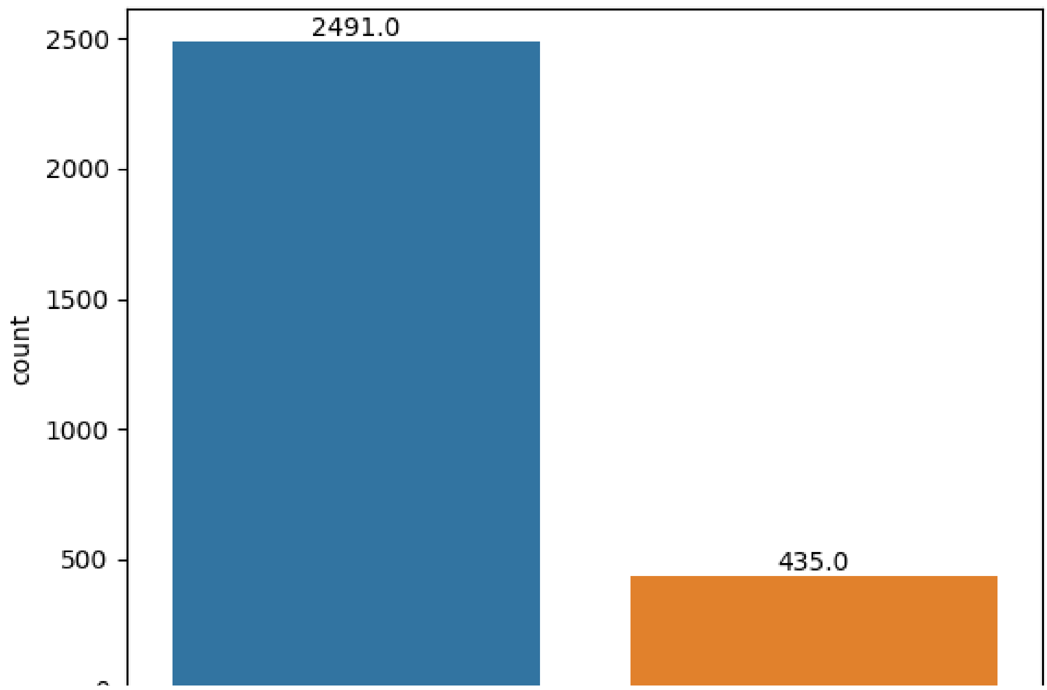
In [17]:

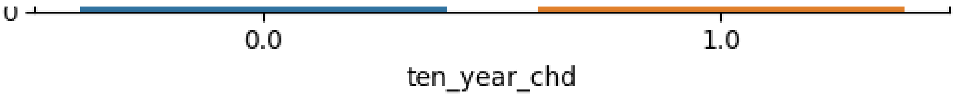
```
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,random_state=42)
print(f'X_train:{X_train.shape}, X_test:{X_test.shape}')
print(f'y_train:{y_train.shape},y_test:{y_test.shape}')
```

```
X_train:(2926, 15), X_test:(732, 15)
y_train:(2926,),y_test:(732,)
```

In [18]:

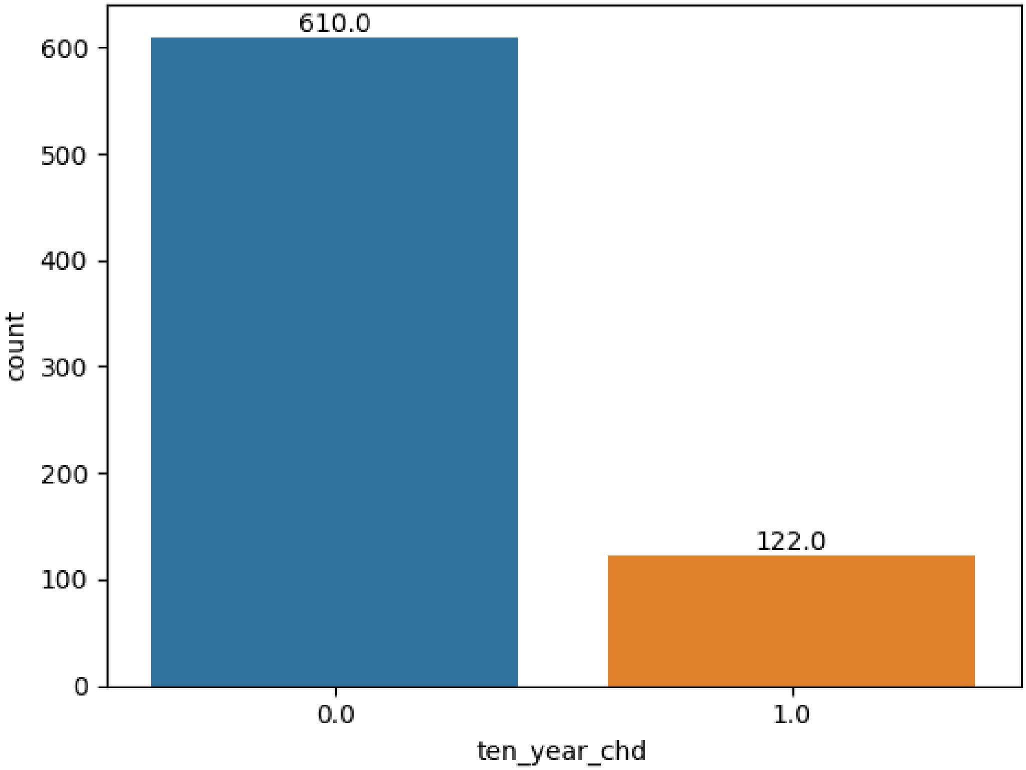
```
ax=sns.countplot(x=y_train)
for p in ax.patches:
    ax.annotate(f'{p.get_height()}',
                (p.get_x() + p.get_width() / 2., p.get_height()),
                ha='center',
                va='bottom')
plt.show()
```





In [19]:

```
ax=sns.countplot(x=y_test)
for p in ax.patches:
    ax.annotate(f'{p.get_height()}',
                (p.get_x() + p.get_width() / 2., p.get_height()),
                ha='center',
                va='bottom')
plt.show()
```



Models

performance metrics plot function

In [20]:

```
def plot_roc_curve(y_test, y_scores):
    """
    Plots the ROC curve and calculates the AUC.

    Parameters:
        y_test (array-like): True labels for the test set.
        y_scores (array-like): Predicted probabilities for the positive class.
    """
    # Calculate ROC curve and AUC
    fpr, tpr, thresholds = roc_curve(y_test, y_scores)
    roc_auc = auc(fpr, tpr)

    # Plot ROC curve
    plt.figure(figsize=(8, 6))
    sns.lineplot(x=fpr, y=tpr, color='red', lw=2, label=f'ROC curve (area = {roc_auc:.2f
})')
    sns.lineplot(x=[0, 1], y=[0, 1], color='gray', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])

    # Set the title and labels
```

```
    plt.title('Receiver Operating Characteristic (ROC) Curve')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')

    # Add legend
    plt.legend(loc='lower right')

    # Add grid for better readability
    plt.grid(linestyle='--', alpha=0.7)

    # Show the plot
    plt.show()
```

In [21]:

```
def plot_confusion_matrix(y_true, y_pred, labels=['No CHD', 'CHD']):
    """
    Plots the confusion matrix.

    Parameters:
        y_true (array-like): True labels.
        y_pred (array-like): Predicted labels.
        labels (list): List of label names for the confusion matrix.
    """
    # Generate the confusion matrix
    conf_matrix = confusion_matrix(y_true, y_pred)

    # Plot the confusion matrix
    plt.figure(figsize=(8, 6))
    sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Reds', xticklabels=labels, ytick
labels=labels)

    # Set the title and labels
    plt.title('Confusion Matrix')
    plt.xlabel('Predicted')
    plt.ylabel('Actual')

    # Show the plot
    plt.show()
```

Logistic Regression

In [22]:

```
# Choosing the hyperparameters
param_grid = {
    'C': [0.01, 0.1, 1, 10, 100],
    'penalty': ['l1', 'l2'],
    'solver': ['liblinear']
}

# Grid search
grid_search = GridSearchCV(LogisticRegression(), param_grid, cv=5, scoring='accuracy', r
eturn_train_score=True)
grid_search.fit(X_train, y_train)

# Best parameters and performance
print("Best parameters:", grid_search.best_params_)

# Show all results
results = grid_search.cv_results_
results_df = pd.DataFrame({
    'Mean Test Score': results['mean_test_score'],
    'Mean Train Score': results['mean_train_score'],
    'Parameters': results['params'],
    'Rank': results['rank_test_score']
})

# Sort the DataFrame by Mean Test Score
results_df = results_df.sort_values(by='Mean Test Score', ascending=False).reset_index(d
```

```
rop=True)

# Print the results DataFrame
print(results_df)
```

```
Best parameters: {'C': 1, 'penalty': 'l1', 'solver': 'liblinear'}
   Mean Test Score  Mean Train Score  \
0         0.852700          0.855178
1         0.851675          0.851931
2         0.851333          0.851333
3         0.851333          0.851333
4         0.851333          0.851333
5         0.850990          0.855092
6         0.849964          0.855434
7         0.849964          0.855434
8         0.849964          0.855178
9         0.849964          0.855178

                                          Parameters  Rank
0    {'C': 1, 'penalty': 'l1', 'solver': 'liblinear'}     1
1  {'C': 0.1, 'penalty': 'l2', 'solver': 'libline...     2
2  {'C': 0.01, 'penalty': 'l1', 'solver': 'liblin...     3
3  {'C': 0.01, 'penalty': 'l2', 'solver': 'liblin...     3
4  {'C': 0.1, 'penalty': 'l1', 'solver': 'libline...     3
5    {'C': 1, 'penalty': 'l2', 'solver': 'liblinear'}     6
6   {'C': 10, 'penalty': 'l1', 'solver': 'liblinear'}     7
7   {'C': 10, 'penalty': 'l2', 'solver': 'liblinear'}     7
8  {'C': 100, 'penalty': 'l1', 'solver': 'libline...     7
9  {'C': 100, 'penalty': 'l2', 'solver': 'libline...     7
```

Based on the preceding results, the ideal hyperparameters for our model will be:

- **C : 1**
- **penalty : l1**
- **solver : liblinear**

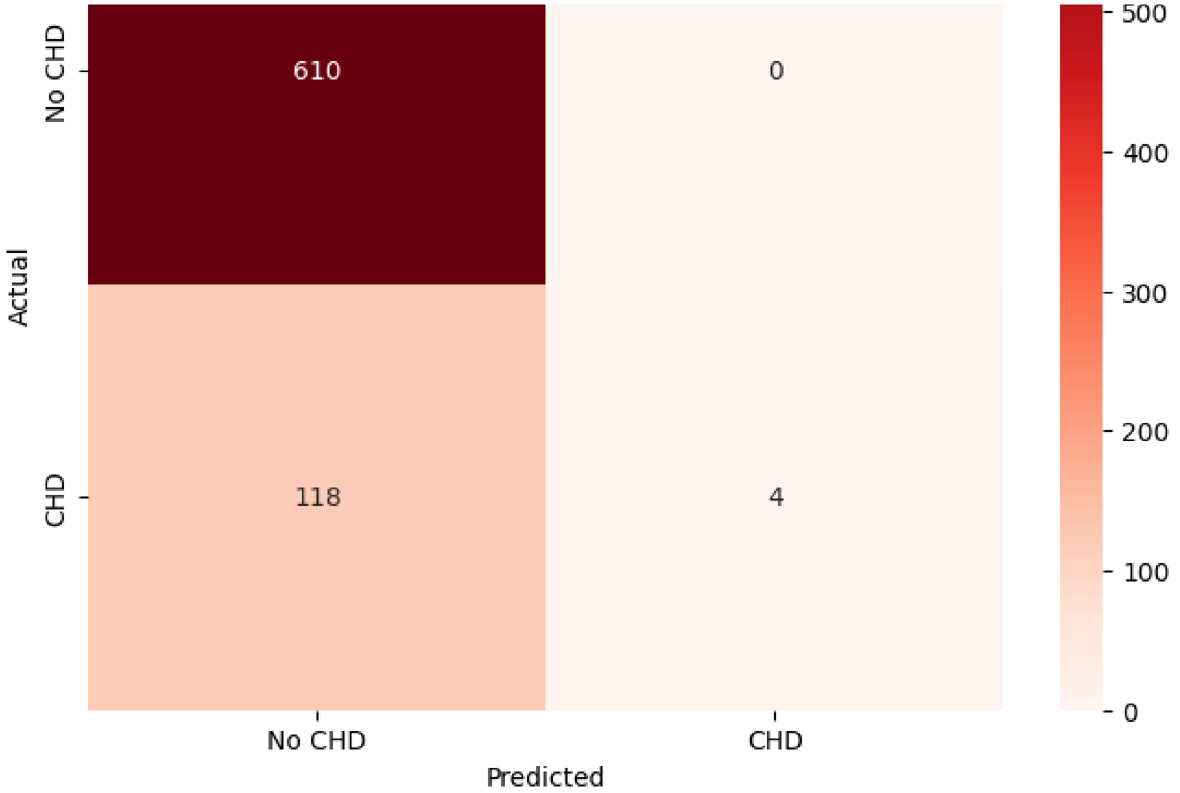
In [23]:

```
# Fitting logistic regression to the training set
logistic_regression_classifier=LogisticRegression(C=1,penalty='l1',solver='liblinear',ran
dom_state=0)
logistic_regression_classifier.fit(X_train,y_train)
#Predicting the test results
y_pred_logistic_regression=logistic_regression_classifier.predict(X_test)
accuracy_log=accuracy_score(y_test,y_pred_logistic_regression)
print('Accuracy Score: ',accuracy_log)
print(classification_report(y_test, y_pred_logistic_regression))
```

```
Accuracy Score:  0.8387978142076503
              precision    recall  f1-score   support

         0.0       0.84      1.00      0.91       610
         1.0       1.00      0.03      0.06       122

    accuracy                           0.84       732
   macro avg       0.92      0.52      0.49       732
weighted avg       0.86      0.84      0.77       732
```

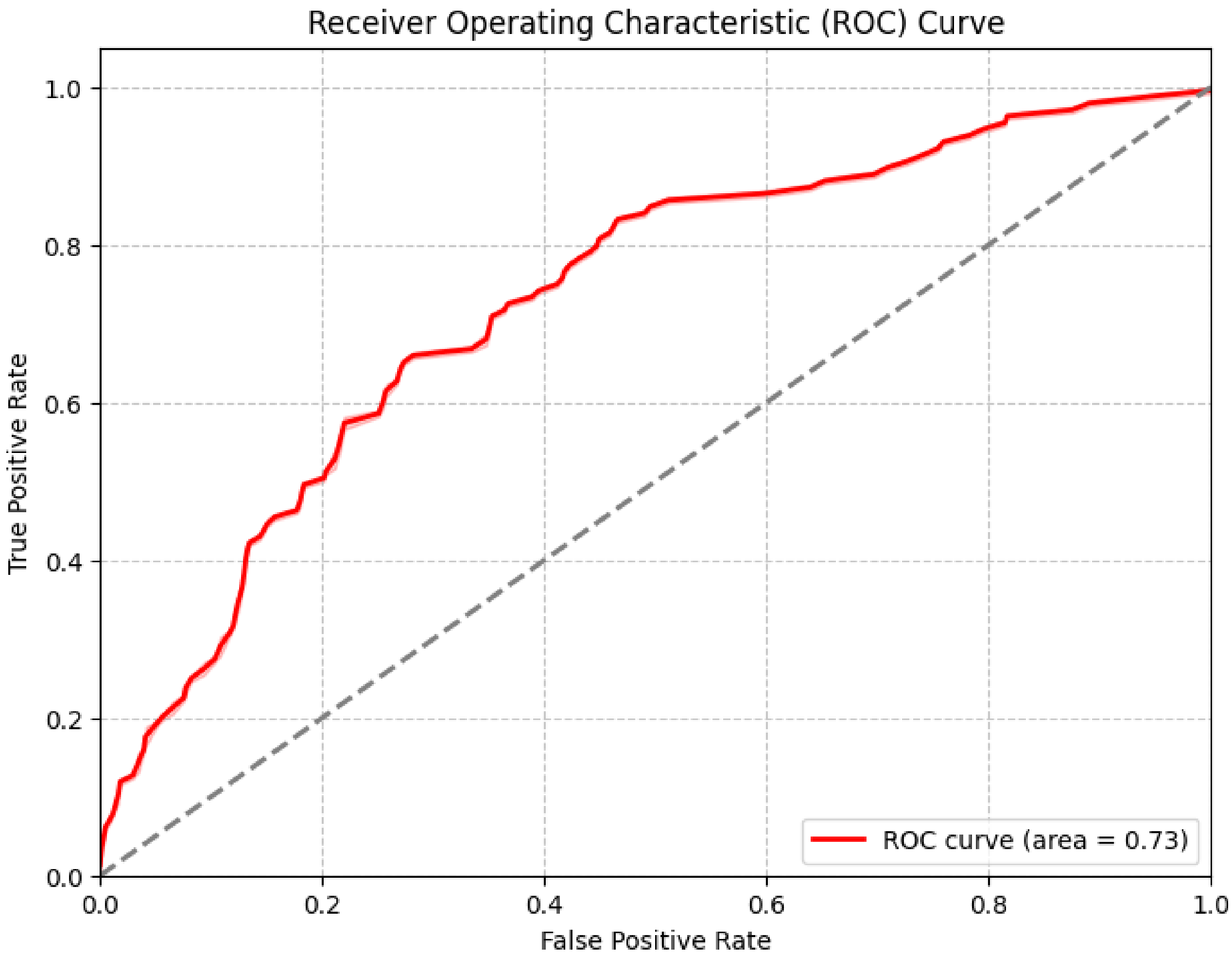
In [24]:

```
#confusion matrix
plot_confusion_matrix(y_test, y_pred_logistic_regression)
```



In [25]:

```
# Extract probabilities for the positive class
y_scores = logistic_regression_classifier.predict_proba(X_test)[:, 1]

# Call the function to plot the ROC curve
plot_roc_curve(y_test, y_scores)
```



Support Vector Machine(SVM)

In [26]:

```
# choosing the hyperparameters
param_grid = [
    {'kernel': ['linear'], 'C': [0.1, 1, 10]},
    {'kernel': ['poly'], 'C': [0.1, 1, 10], 'degree': [2, 3, 4], 'coef0': [0, 1]},
    {'kernel': ['rbf'], 'C': [0.1, 1, 10], 'gamma': [0.01, 0.1, 1]},
]
# Grid Search
grid_search = GridSearchCV(SVC(), param_grid, cv=5, scoring='accuracy', n_jobs=-1,return
_train_score=True)
grid_search.fit(X_train, y_train)

# Best parameters and performance
print("Best parameters:", grid_search.best_params_)

# Show all results
results = grid_search.cv_results_
results_df = pd.DataFrame({
    'Mean Test Score': results['mean_test_score'],
    'Mean Train Score': results['mean_train_score'],
    'Parameters': results['params'],
    'Rank': results['rank_test_score']
})

# Sort the DataFrame by Mean Test Score
results_df = results_df.sort_values(by='Mean Test Score', ascending=False).reset_index(d
rop=True)

# Print the results DataFrame
print(results_df)
```

```
Best parameters: {'C': 0.1, 'coef0': 1, 'degree': 4, 'kernel': 'poly'}
    Mean Test Score  Mean Train Score  \
0          0.851335          0.868763
1          0.851333          0.851333
2          0.851333          0.851333
3          0.851333          0.851333
4          0.851333          0.851333
5          0.851333          0.851333
6          0.851333          0.851333
7          0.851333          0.851333
8          0.851333          0.851333
9          0.851333          0.851333
10         0.850992          0.865174
11         0.850992          0.851418
12         0.850648          0.855263
13         0.850308          0.880554
14         0.849968          0.851418
15         0.849966          0.863978
16         0.849966          0.865174
17         0.849624          0.859450
18         0.849623          0.855947
19         0.849283          0.853554
20         0.849282          0.879785
21         0.848256          0.854494
22         0.848256          0.854494
23         0.847915          0.884826
24         0.847914          0.857826
25         0.847572          0.880297
26         0.847230          0.857826
27         0.840738          0.898752
28         0.839029          0.904477
29         0.834241          0.911313

                                           Parameters  Rank
0   {'C': 0.1, 'coef0': 1, 'degree': 4, 'kernel': ...     1
1                         {'C': 0.1, 'kernel': 'linear'}     2
2              {'C': 1, 'gamma': 0.01, 'kernel': 'rbf'}     2
3               {'C': 0.1, 'gamma': 1, 'kernel': 'rbf'}     2
4             {'C': 0.1, 'gamma': 0.1, 'kernel': 'rbf'}     2
5            {'C': 0.1, 'gamma': 0.01, 'kernel': 'rbf'}     2
```

```
6            {'C': 10, 'gamma': 0.01, 'kernel': 'rbf'}      2
7              {'C': 1, 'gamma': 0.1, 'kernel': 'rbf'}      2
8    {'C': 0.1, 'coef0': 1, 'degree': 2, 'kernel': ...      2
9    {'C': 0.1, 'coef0': 0, 'degree': 2, 'kernel': ...      2
10   {'C': 1, 'coef0': 1, 'degree': 3, 'kernel': 'p...     11
11                        {'C': 1, 'kernel': 'linear'}     12
12   {'C': 0.1, 'coef0': 0, 'degree': 3, 'kernel': ...     13
13   {'C': 1, 'coef0': 0, 'degree': 4, 'kernel': 'p...     14
14                       {'C': 10, 'kernel': 'linear'}     15
15   {'C': 1, 'coef0': 0, 'degree': 3, 'kernel': 'p...     16
16   {'C': 0.1, 'coef0': 0, 'degree': 4, 'kernel': ...     16
17                {'C': 1, 'gamma': 1, 'kernel': 'rbf'}    18
18   {'C': 0.1, 'coef0': 1, 'degree': 3, 'kernel': ...     19
19             {'C': 10, 'gamma': 0.1, 'kernel': 'rbf'}    20
20   {'C': 10, 'coef0': 0, 'degree': 3, 'kernel': '...     21
21   {'C': 1, 'coef0': 0, 'degree': 2, 'kernel': 'p...     22
22   {'C': 1, 'coef0': 1, 'degree': 2, 'kernel': 'p...     22
23   {'C': 1, 'coef0': 1, 'degree': 4, 'kernel': 'p...     24
24   {'C': 10, 'coef0': 1, 'degree': 2, 'kernel': '...     25
25   {'C': 10, 'coef0': 1, 'degree': 3, 'kernel': '...     26
26   {'C': 10, 'coef0': 0, 'degree': 2, 'kernel': '...     27
27               {'C': 10, 'gamma': 1, 'kernel': 'rbf'}    28
28   {'C': 10, 'coef0': 0, 'degree': 4, 'kernel': '...     29
29   {'C': 10, 'coef0': 1, 'degree': 4, 'kernel': '...     30
```

Based on the preceding results, the ideal hyperparameters for our model will be:

- **C : 0.1**
- **coef0 : 1**
- **degree : 4 kernel : Polynomial**

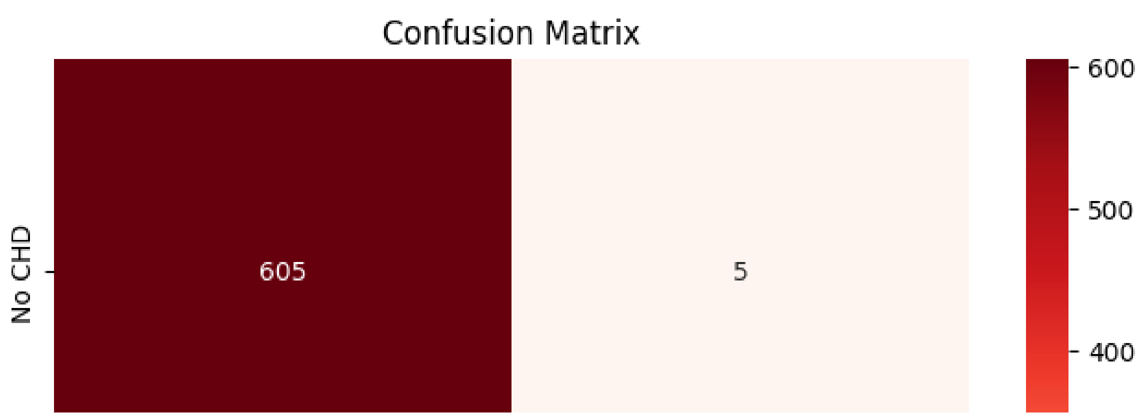
In [27]:

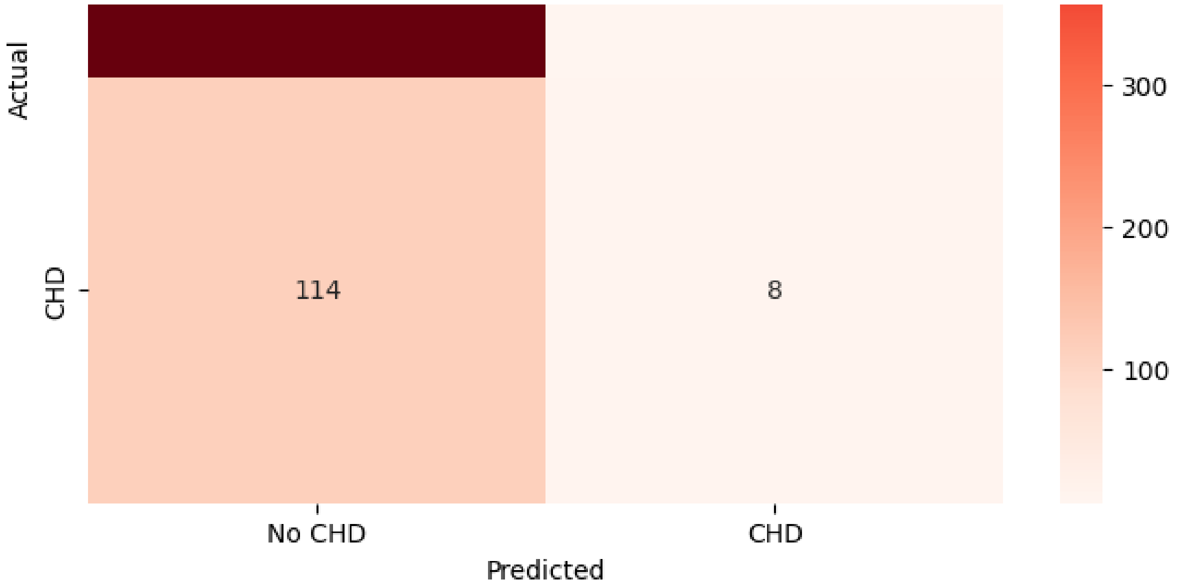
```
svm_classifier=SVC(kernel='poly', C=0.1, degree=4, coef0=1,probability=True ,random_stat
e=42)
svm_classifier.fit(X_train,y_train)
y_pred_svm=svm_classifier.predict(X_test)
accuracy_svm=accuracy_score(y_test,y_pred_svm)
print('Accuracy Score: ',accuracy_svm)
print(classification_report(y_test, y_pred_svm))
```

```
Accuracy Score:  0.837431693989071
              precision    recall  f1-score   support

         0.0       0.84      0.99      0.91       610
         1.0       0.62      0.07      0.12       122

    accuracy                           0.84       732
   macro avg       0.73      0.53      0.51       732
weighted avg       0.80      0.84      0.78       732
```

In [28]:

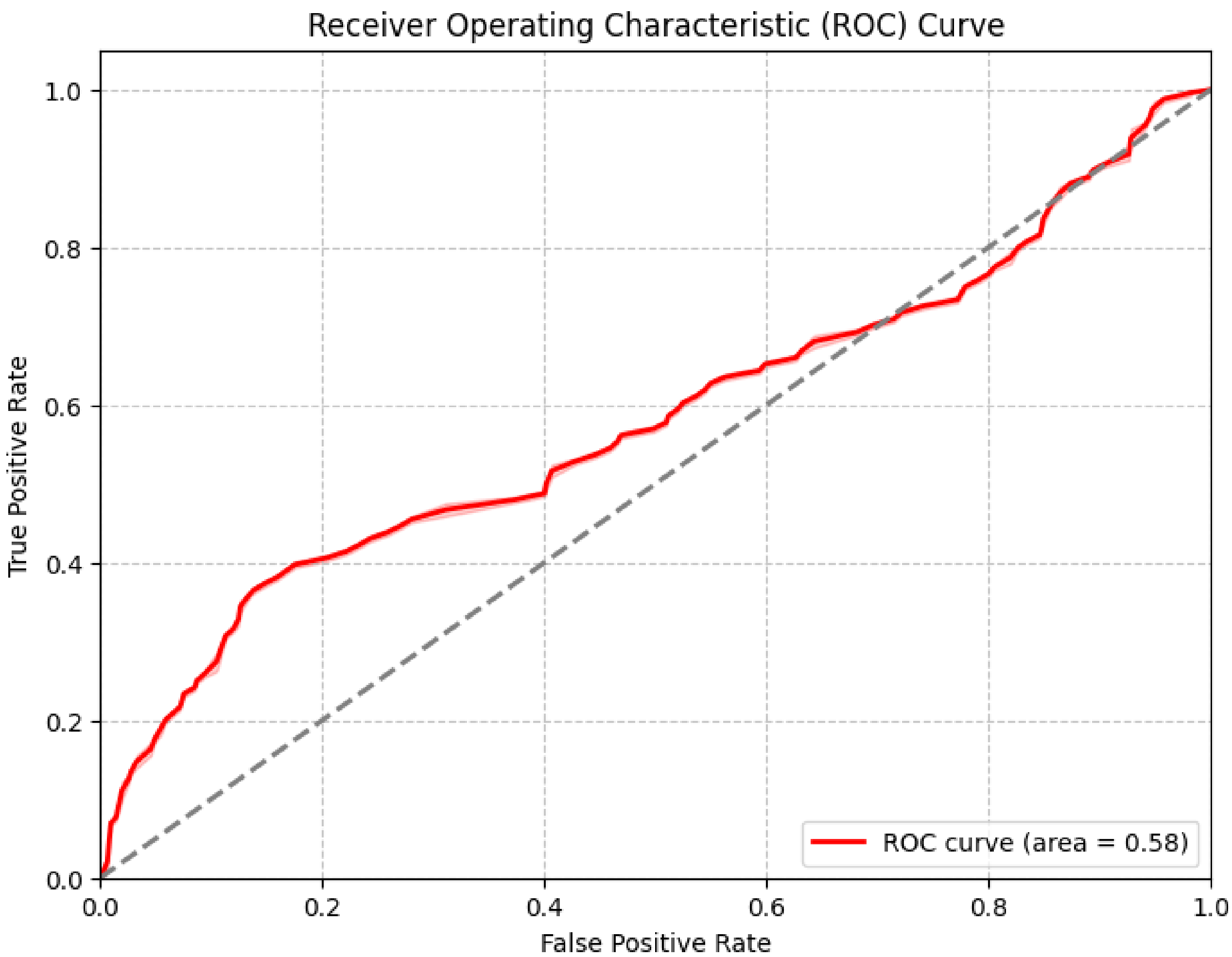
```
#confusion matrix
plot_confusion_matrix(y_test, y_pred_svm)
```





In [29]:

```
# Extract probabilities for the positive class
y_scores = svm_classifier.predict_proba(X_test)[:, 1]

# Call the function to plot the ROC curve
plot_roc_curve(y_test, y_scores)
```



Naive Bayes

In [30]:

```
# Define parameter grid
param_grid = {
    'var_smoothing': np.logspace(0, -9, num=100)  # Exploring a range of values
}
```

```
# Perform Grid Search
grid_search = GridSearchCV(GaussianNB(), param_grid, cv=5, scoring='accuracy',return_tra
in_score=True)
grid_search.fit(X_train, y_train)

# Best parameters and performance
print("Best parameters:", grid_search.best_params_)

# Show all results
results = grid_search.cv_results_
results_df = pd.DataFrame({
    'Mean Test Score': results['mean_test_score'],
    'Mean Train Score': results['mean_train_score'],
    'Parameters': results['params'],
    'Rank': results['rank_test_score']
})

# Sort the DataFrame by Mean Test Score
results_df = results_df.sort_values(by='Mean Test Score', ascending=False).reset_index(d
rop=True)

# Print the results DataFrame
print(results_df)
```

```
Best parameters: {'var_smoothing': 0.657933224657568}
    Mean Test Score  Mean Train Score  \
0          0.851674          0.852358
1          0.851333          0.851333
2          0.851333          0.851675
3          0.850650          0.852615
4          0.848600          0.850393
..              ...               ...
95         0.827409          0.828008
96         0.827409          0.828008
97         0.827409          0.828008
98         0.827409          0.828008
99         0.827409          0.828008

                                   Parameters  Rank
0        {'var_smoothing': 0.657933224657568}     1
1                      {'var_smoothing': 1.0}     2
2       {'var_smoothing': 0.8111308307896871}     2
3        {'var_smoothing': 0.533669923120631}     4
4      {'var_smoothing': 0.43287612810830584}     5
..                                        ...   ...
95    {'var_smoothing': 3.511191734215127e-05}    28
96    {'var_smoothing': 4.328761281083062e-05}    28
97   {'var_smoothing': 5.3366992312063123e-05}    28
98    {'var_smoothing': 6.579332246575683e-05}    28
99                   {'var_smoothing': 1e-09}    28

[100 rows x 4 columns]
```

Based on the preceding results, the ideal hyperparameters for our model will be:

- **var_smoothing: 0.657933224657568**

In [31]:

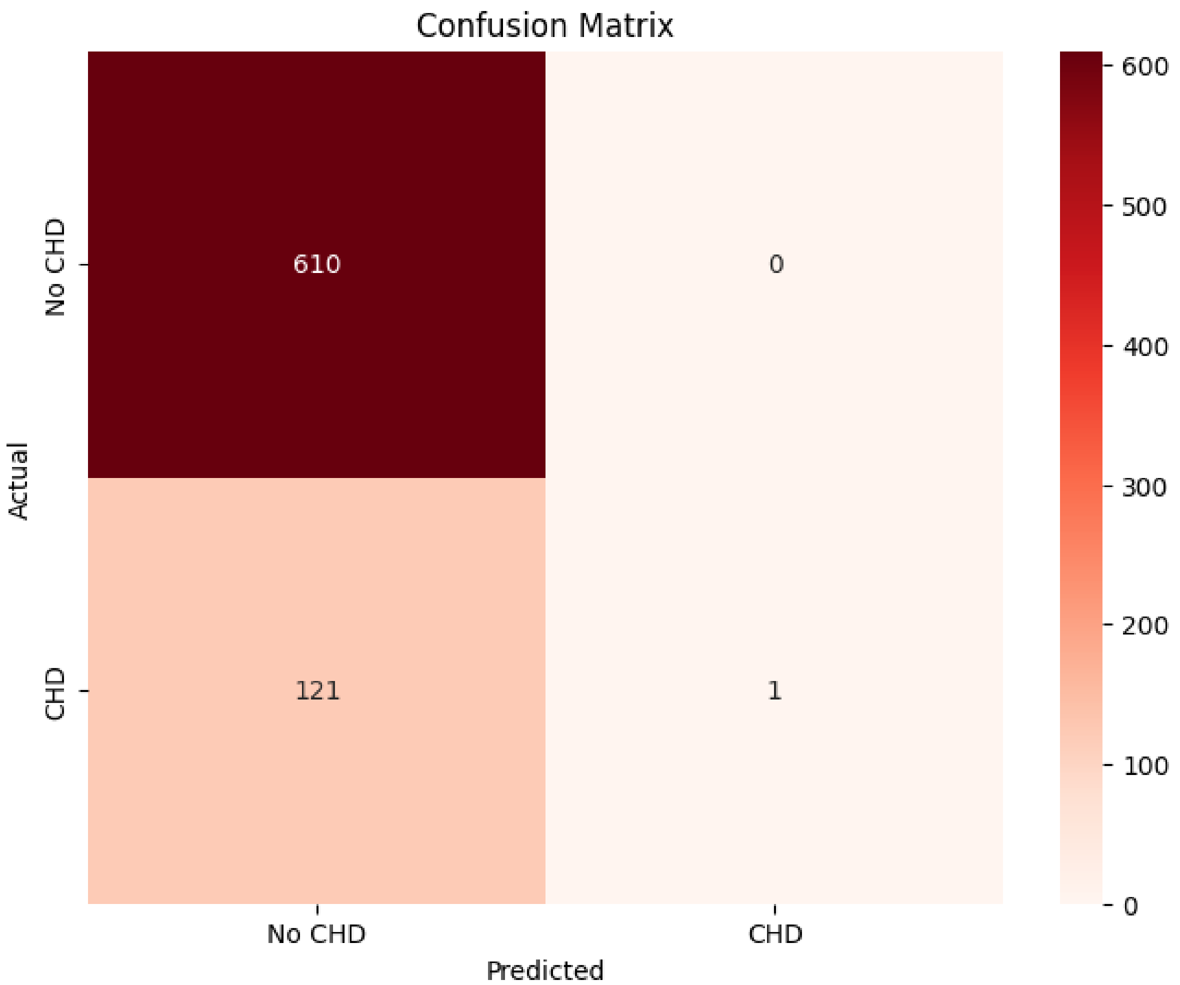
```
nb_classifier=GaussianNB(var_smoothing= 0.657933224657568)
nb_classifier.fit(X_train,y_train)
y_pred_nb=nb_classifier.predict(X_test)
accuracy_nb=accuracy_score(y_test,y_pred_nb)
print('Accuracy Score: ',accuracy_nb)
print(classification_report(y_test, y_pred_nb))
```

```
Accuracy Score:  0.8346994535519126
              precision    recall  f1-score   support

         0.0       0.83      1.00      0.91       610
```

```
         1.0       1.00      0.01      0.02       122

    accuracy                           0.83       732
   macro avg       0.92      0.50      0.46       732
weighted avg       0.86      0.83      0.76       732
```

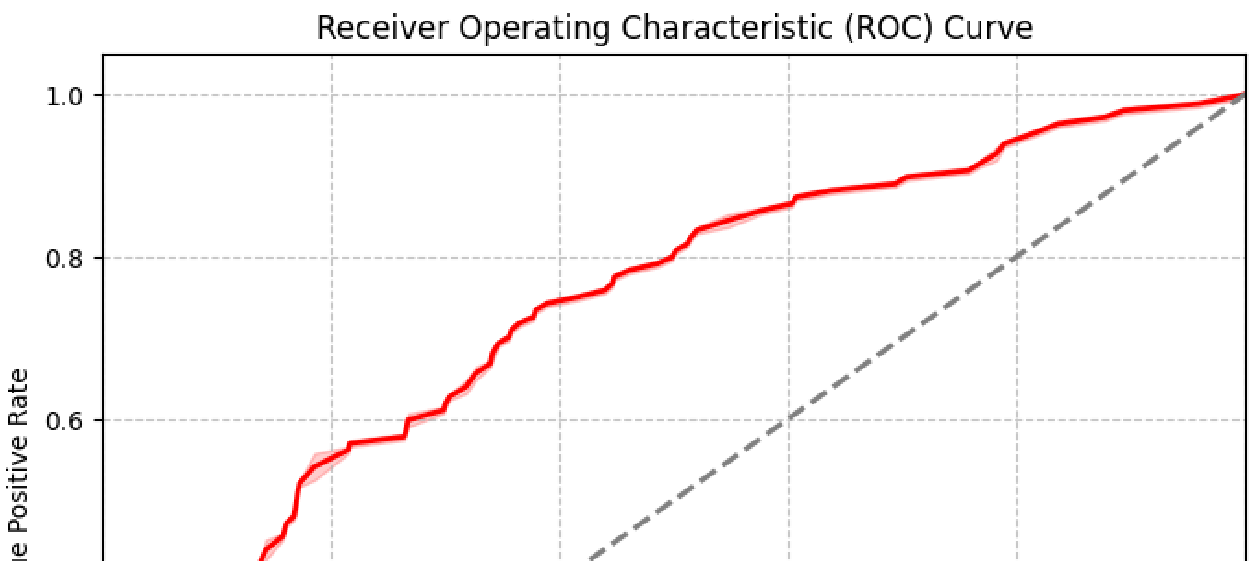
In [32]:

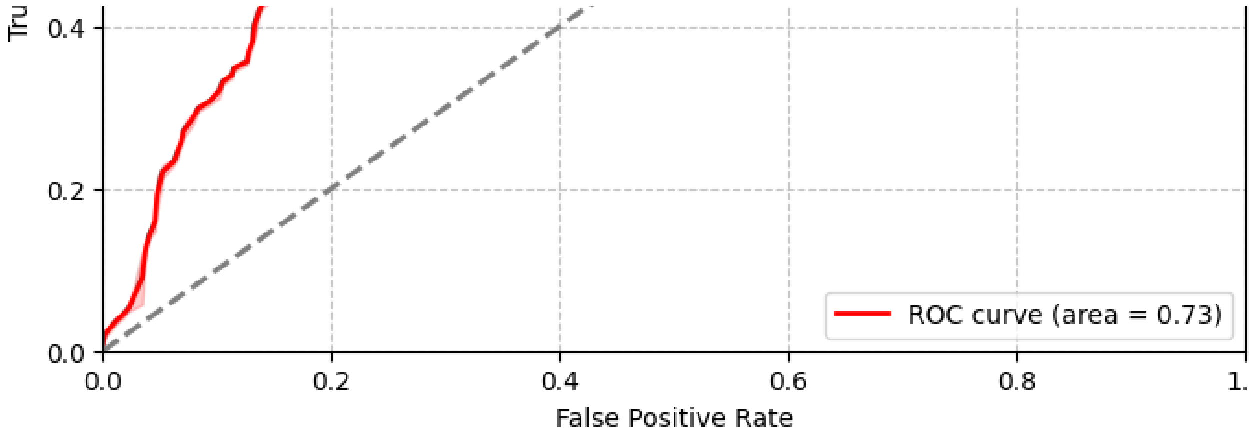
```
#confusion matrix
plot_confusion_matrix(y_test, y_pred_nb)
```



In [33]:

```
# Extract probabilities for the positive class
y_scores = nb_classifier.predict_proba(X_test)[:, 1]

# Call the function to plot the ROC curve
plot_roc_curve(y_test, y_scores)
```





Decision Tree

In [34]:

```python
# choosing the hyperparameters
param_grid = {
    'max_depth': [None, 5, 10, 15, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['auto', 'sqrt', 'log2']
}
#Grid search
grid_search = GridSearchCV(DecisionTreeClassifier(random_state=42), param_grid, cv=5, sc
oring='accuracy',return_train_score=True)
grid_search.fit(X_train, y_train)

# Best parameters and performance
print("Best parameters:", grid_search.best_params_)

# Show all results
results = grid_search.cv_results_
results_df = pd.DataFrame({
    'Mean Test Score': results['mean_test_score'],
    'Mean Train Score': results['mean_train_score'],
    'Parameters': results['params'],
    'Rank': results['rank_test_score']
})

# Sort the DataFrame by Mean Test Score
results_df = results_df.sort_values(by='Mean Test Score', ascending=False).reset_index(d
rop=True)

# Print the results DataFrame
print(results_df)
```

```
Best parameters: {'max_depth': 5, 'max_features': 'auto', 'min_samples_leaf': 2, 'min_sam
ples_split': 5}
     Mean Test Score  Mean Train Score  \
0           0.848939          0.862013
1           0.848939          0.862013
2           0.848939          0.862013
3           0.847573          0.860902
4           0.847573          0.860902
..               ...               ...
130         0.769653          0.996924
131         0.769653          0.996924
132         0.759062          1.000000
133         0.759062          1.000000
134         0.759062          1.000000

                                            Parameters  Rank
0    {'max_depth': 5, 'max_features': 'log2', 'min_...     1
1    {'max_depth': 5, 'max_features': 'sqrt', 'min_...     1
2    {'max_depth': 5, 'max_features': 'auto', 'min_...     1
3    {'max_depth': 5, 'max_features': 'sqrt', 'min_...     4
```

```
4     {'max_depth': 5, 'max_features': 'log2', 'min_...      4
..                                                  ...    ...
130   {'max_depth': 20, 'max_features': 'sqrt', 'min...    130
131   {'max_depth': 20, 'max_features': 'auto', 'min...    130
132   {'max_depth': None, 'max_features': 'sqrt', 'm...    133
133   {'max_depth': None, 'max_features': 'log2', 'm...    133
134   {'max_depth': None, 'max_features': 'auto', 'm...    133

[135 rows x 4 columns]
```

Based on the preceding results, the ideal hyperparameters for our model will be:

- **max_depth : 5**
- **max_features : auto**
- **min_samples_split : 5**
- **min_samples_leaf : 2**

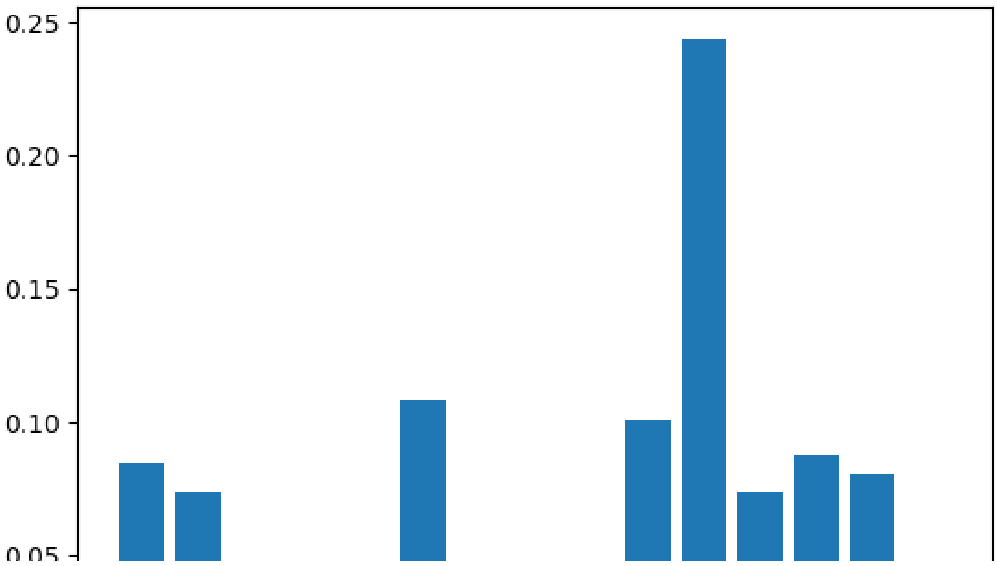
In [35]:

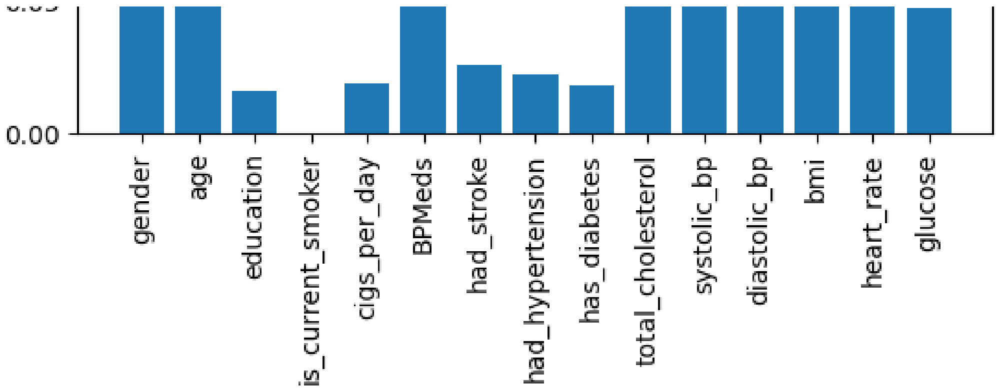
```
# Create and fit the Decision Tree classiresults = grid_search.cv_results_fier
dt_classifier = DecisionTreeClassifier(max_depth=5,max_features='auto',min_samples_leaf=
2,min_samples_split=5)
dt_classifier.fit(X_train, y_train)

# Make predictions on the test set
y_pred_dt = dt_classifier.predict(X_test)  # Use predict() method

# Calculate the accuracy score
accuracy_dt = accuracy_score(y_test, y_pred_dt)

# Print the accuracy score
print('Accuracy Score:', accuracy_dt)
print(classification_report(y_test, y_pred_dt))
```

```
Accuracy Score: 0.8278688524590164
              precision    recall  f1-score   support

         0.0       0.84      0.99      0.91       610
         1.0       0.33      0.03      0.06       122

    accuracy                           0.83       732
   macro avg       0.58      0.51      0.48       732
weighted avg       0.75      0.83      0.76       732
```

In [36]:

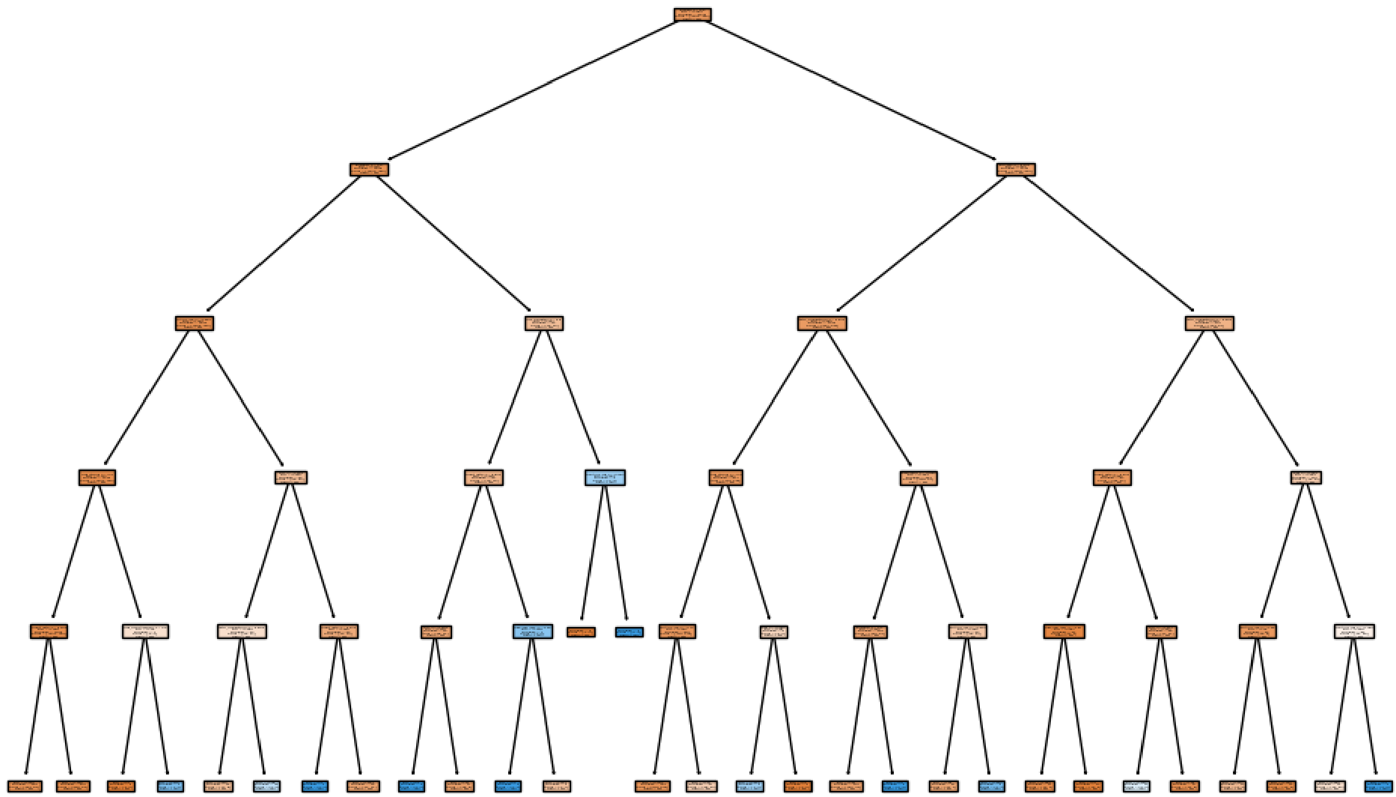
```
feature_importances = dt_classifier.feature_importances_
plt.bar(range(len(feature_importances)), feature_importances)
plt.xticks(range(len(feature_importances)), X.columns, rotation=90)
plt.show()
```





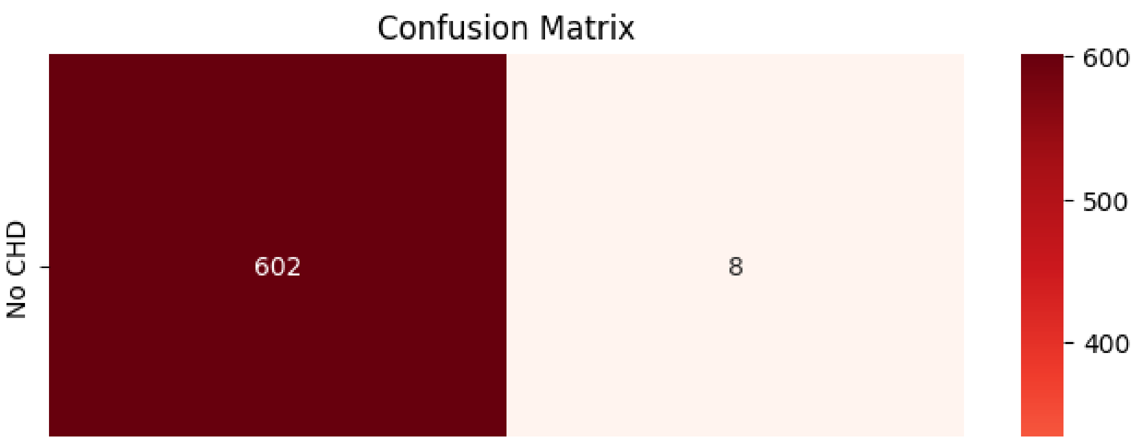
In [37]:

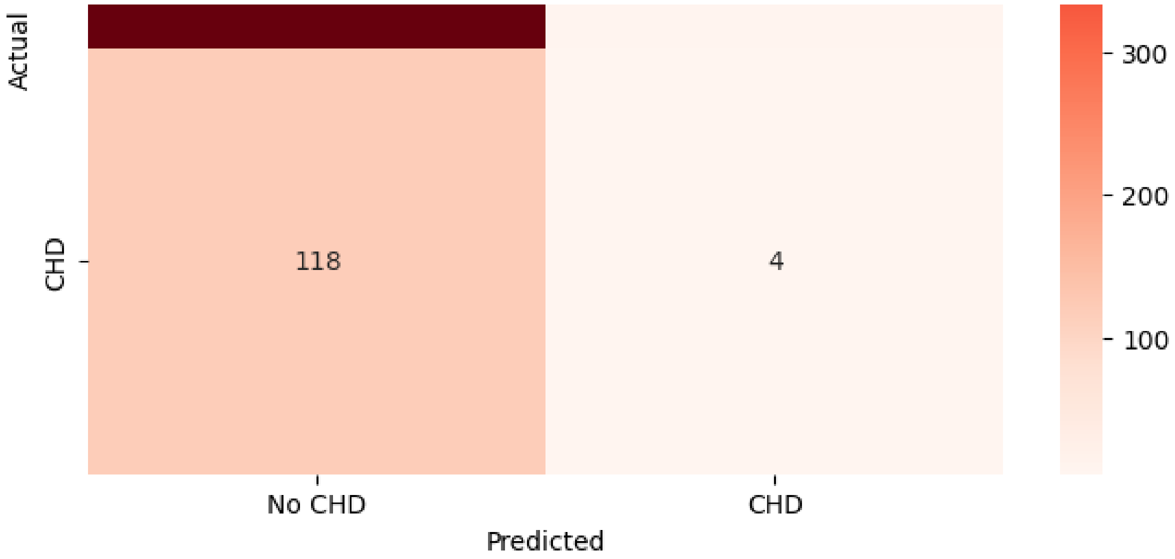
```
plt.figure(figsize=(12, 8))
plot_tree(dt_classifier, filled=True, feature_names=X.columns, class_names=['No', 'Yes']
)
plt.show()
```



In [38]:

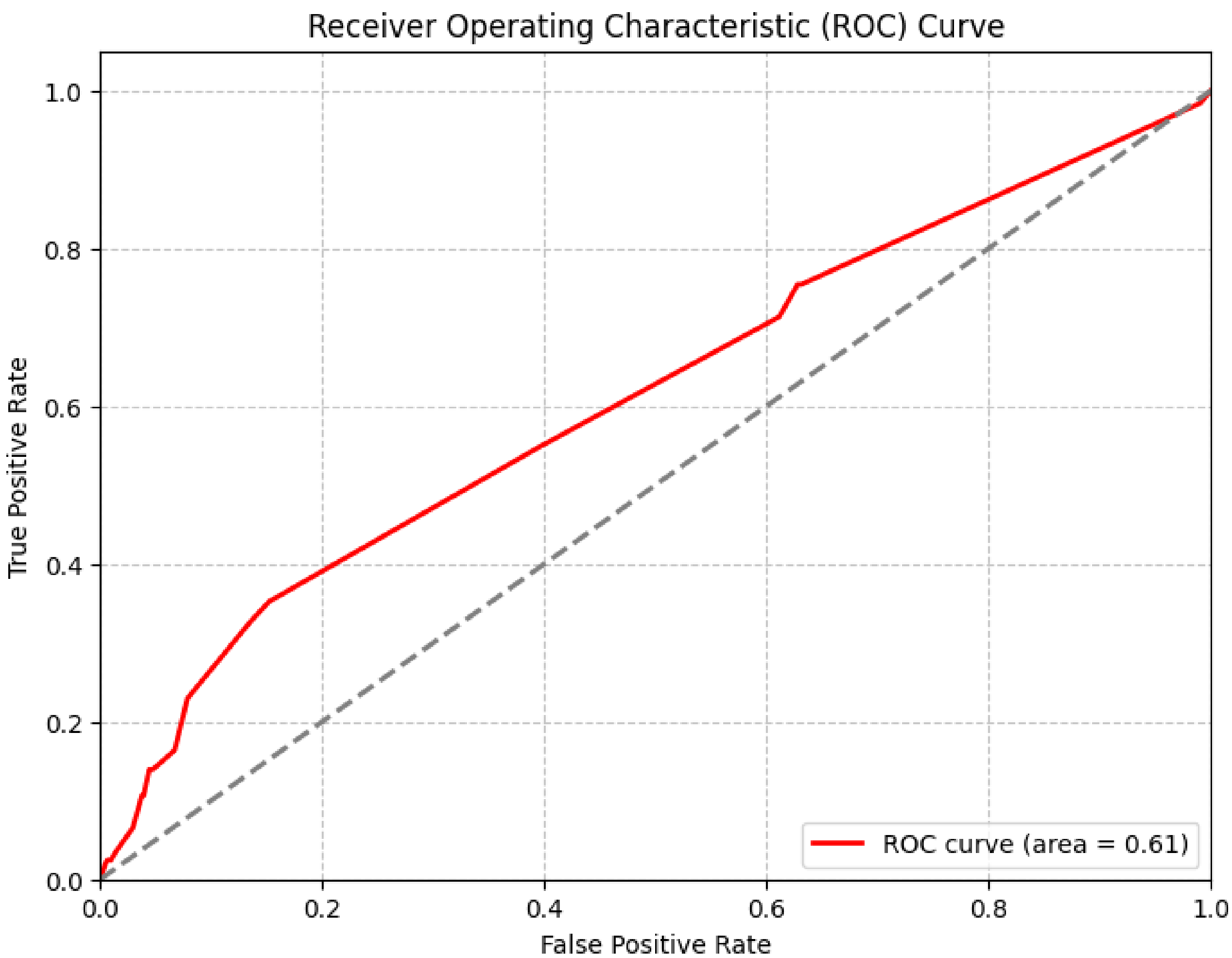
```
#confusion matrix
plot_confusion_matrix(y_test, y_pred_dt)
```





In [39]:

```
# Extract probabilities for the positive class
y_scores = dt_classifier.predict_proba(X_test)[:, 1]

# Call the function to plot the ROC curve
plot_roc_curve(y_test, y_scores)
```



Gradient Boosting

In [40]:

```
# Define parameter grid
param_grid = {
    'n_estimators': [100, 200],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 5, 7],
```

```python
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2],
    'subsample': [0.8, 1.0]
}

# Perform Grid Search
grid_search = GridSearchCV(GradientBoostingClassifier(), param_grid, cv=5, scoring='accu
racy',return_train_score=True)
grid_search.fit(X_train, y_train)

# Best parameters and performance
print("Best parameters:", grid_search.best_params_)

# Show all results
results = grid_search.cv_results_
results_df = pd.DataFrame({
    'Mean Test Score': results['mean_test_score'],
    'Mean Train Score': results['mean_train_score'],
    'Parameters': results['params'],
    'Rank': results['rank_test_score']
})

# Sort the DataFrame by Mean Test Score
results_df = results_df.sort_values(by='Mean Test Score', ascending=False).reset_index(d
rop=True)

# Print the results DataFrame
print(results_df)
```

```
Best parameters: {'learning_rate': 0.01, 'max_depth': 5, 'min_samples_leaf': 2, 'min_samp
les_split': 2, 'n_estimators': 200, 'subsample': 1.0}
     Mean Test Score  Mean Train Score  \
0           0.854749          0.879187
1           0.854406          0.880297
2           0.853724          0.880383
3           0.853723          0.881152
4           0.853722          0.880041
..               ...               ...
139         0.828776          0.950701
140         0.828775          0.990687
141         0.828433          0.989747
142         0.828097          0.950444
143         0.825017          0.951299

                                            Parameters  Rank
0    {'learning_rate': 0.01, 'max_depth': 5, 'min_s...     1
1    {'learning_rate': 0.01, 'max_depth': 5, 'min_s...     2
2    {'learning_rate': 0.01, 'max_depth': 5, 'min_s...     3
3    {'learning_rate': 0.01, 'max_depth': 5, 'min_s...     4
4    {'learning_rate': 0.01, 'max_depth': 5, 'min_s...     5
..                                                 ...   ...
139  {'learning_rate': 0.2, 'max_depth': 3, 'min_sa...   140
140  {'learning_rate': 0.2, 'max_depth': 5, 'min_sa...   141
141  {'learning_rate': 0.2, 'max_depth': 5, 'min_sa...   142
142  {'learning_rate': 0.2, 'max_depth': 3, 'min_sa...   143
143  {'learning_rate': 0.2, 'max_depth': 3, 'min_sa...   144

[144 rows x 4 columns]
```

Based on the preceding results, the ideal hyperparameters for our model will be:

- **max_depth : 5**
- **min_samples_split : 2**
- **min_samples_leaf : 2**
- **n_estimators : 200**
- **learning_rate : 0.01**
- **subsample=0.8**

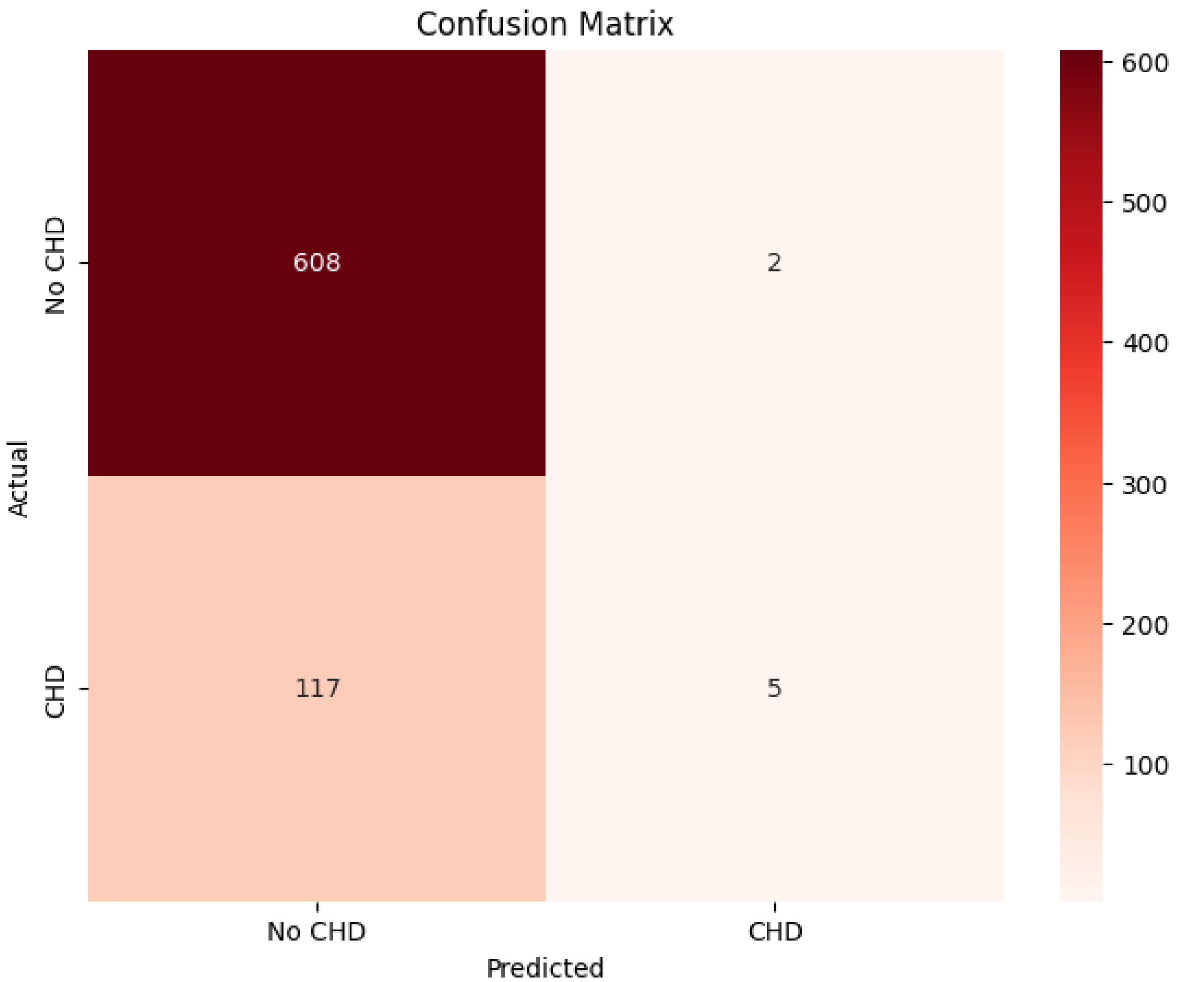
In [41]:

```
# Create the Gradient Boosting classifier
gb_classifier = GradientBoostingClassifier(learning_rate=0.01,max_depth=5,min_samples_lea
f=2,min_samples_split=2,n_estimators=200,subsample=0.8)

# Fit the model to the training data
gb_classifier.fit(X_train, y_train)

# Make predictions on the test set
y_pred_gb = gb_classifier.predict(X_test)

# Calculate accuracy
accuracy_gb = accuracy_score(y_test, y_pred_gb)
print('Accuracy Score:', accuracy_gb)
print(classification_report(y_test, y_pred_gb))
```

```
Accuracy Score: 0.837431693989071
              precision    recall  f1-score   support

         0.0       0.84      1.00      0.91       610
         1.0       0.71      0.04      0.08       122

    accuracy                           0.84       732
   macro avg       0.78      0.52      0.49       732
weighted avg       0.82      0.84      0.77       732
```

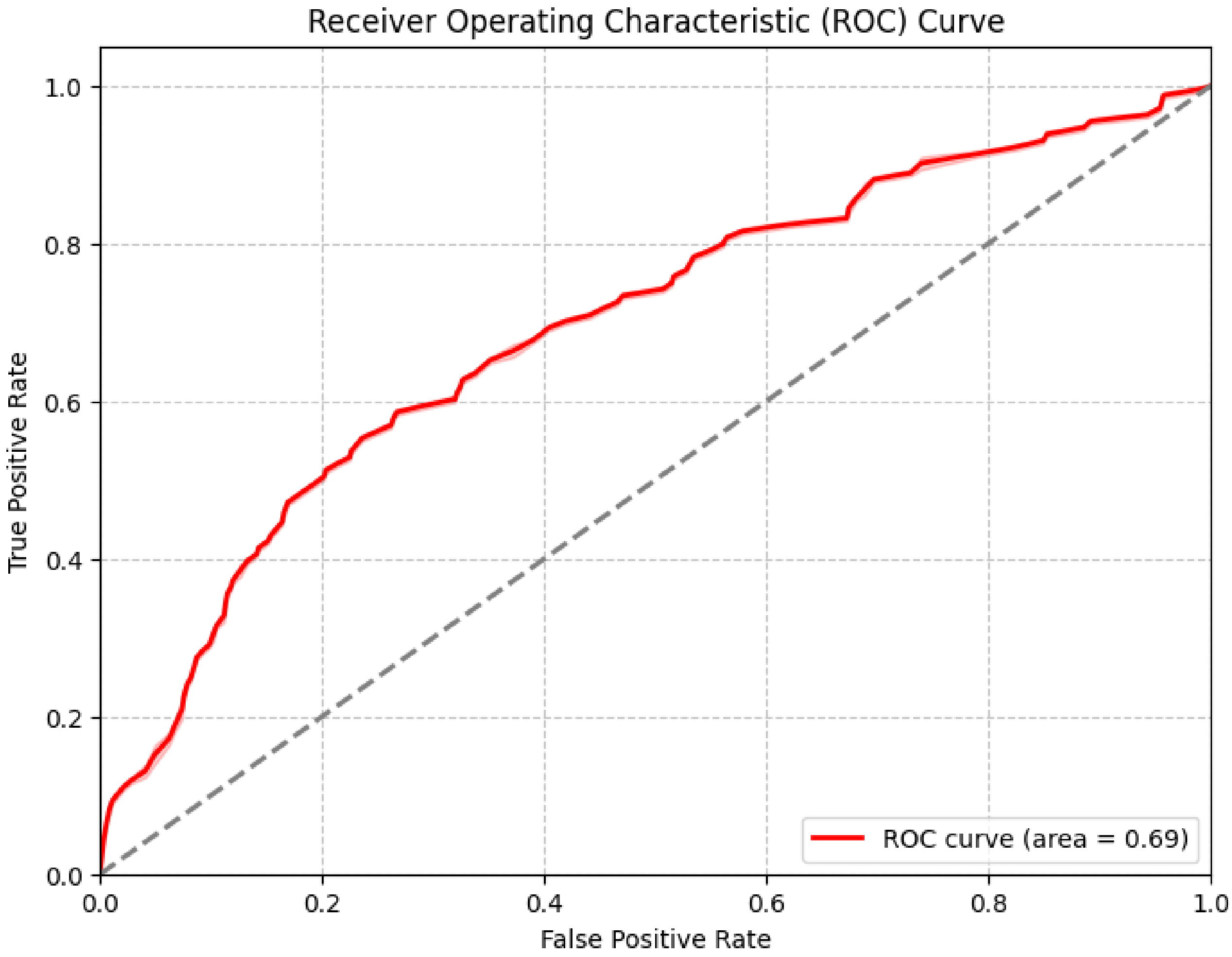
In [42]:

```
#confusion matrix
plot_confusion_matrix(y_test, y_pred_gb)
```



In [43]:

```
# Extract probabilities for the positive class
y_scores = gb_classifier.predict_proba(X_test)[:, 1]

# Call the function to plot the ROC curve
plot_roc_curve(y_test, y_scores)
```



Model Comperision

In [44]:

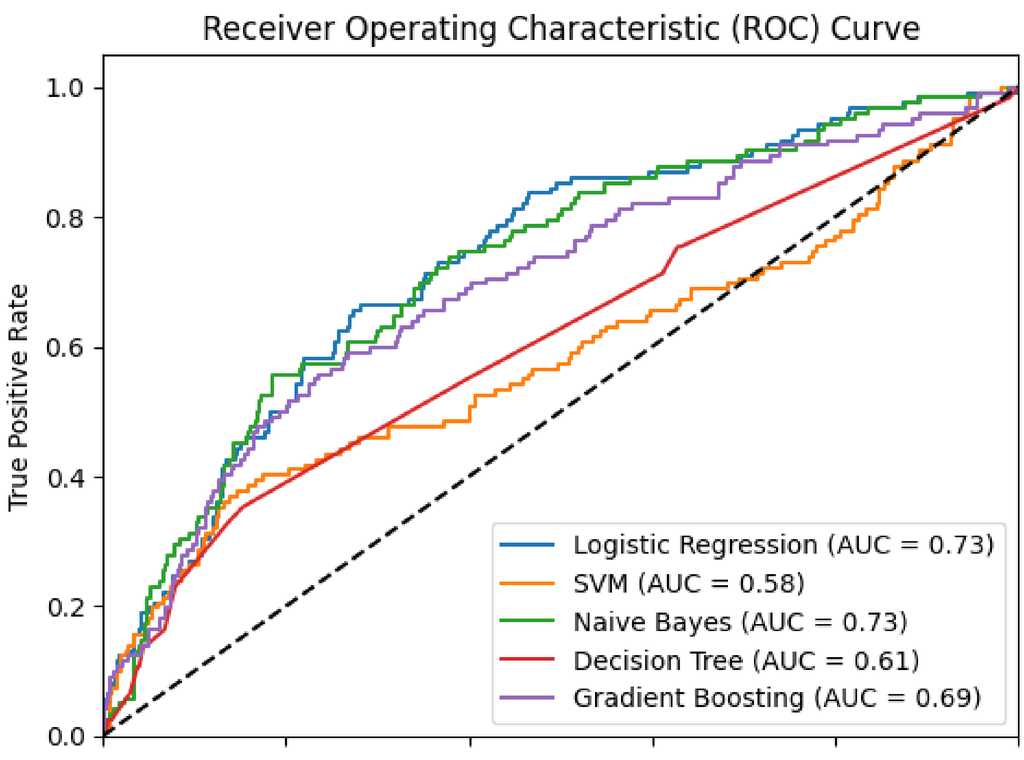
```
# List of models and their corresponding names
models = [
    ('Logistic Regression', logistic_regression_classifier),
    ('SVM', svm_classifier),
    ('Naive Bayes', nb_classifier),
    ('Decision Tree', dt_classifier),
    ('Gradient Boosting', gb_classifier),
]
```

In [45]:

```
results = {}

# Loop through each model to calculate metrics
for model_name, model in models:
    # Make predictions
    y_pred = model.predict(X_test)

    # Calculate metrics
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='weighted')
    recall = recall_score(y_test, y_pred, average='weighted')
    f1 = f1_score(y_test, y_pred, average='weighted')

    # Store the metrics
    results[model_name] = {
        'Accuracy': accuracy,
        'Precision': precision,
        'Recall': recall,
        'F1 Score': f1,
    }
```

```
    # If the model supports probability predictions, calculate ROC AUC and plot ROC curve
    if hasattr(model, "predict_proba"):
        y_scores = model.predict_proba(X_test)[:, 1]  # Get probabilities for the positiv
e class
        roc_auc = roc_auc_score(y_test, y_scores)
        fpr, tpr, _ = roc_curve(y_test, y_scores)

        # Calculate the average FPR
        avg_fpr = np.mean(fpr)

        # Store ROC AUC and average FPR score
        results[model_name]['ROC AUC'] = roc_auc
        results[model_name]['Avg FPR'] = avg_fpr

        # Plot ROC curve
        plt.plot(fpr, tpr, label=f'{model_name} (AUC = {roc_auc:.2f})')

# Convert results to a DataFrame for better visualization
results_df = pd.DataFrame(results).T  # Transpose for better readability

# Print the results as a table
print(results_df)

# Finalize the ROC plot
plt.plot([0, 1], [0, 1], 'k--')  # Diagonal line
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc='lower right')
plt.show()
```

```
                     Accuracy  Precision    Recall  F1 Score   ROC AUC  \
Logistic Regression  0.838798   0.864927  0.838798  0.770423  0.730744
SVM                  0.837432   0.803769  0.837432  0.778469  0.580798
Naive Bayes          0.834699   0.862061  0.834699  0.760851  0.725060
Decision Tree        0.827869   0.752315  0.827869  0.764336  0.611657
Gradient Boosting    0.837432   0.817898  0.837432  0.771971  0.694209

                      Avg FPR
Logistic Regression  0.316761
SVM                  0.446055
Naive Bayes          0.331751
Decision Tree        0.234738
Gradient Boosting    0.337737
```



The table compares classifiers based on their accuracy, precision, recall, F1 score, ROC AUC, and average false positive rate (Avg FPR). Logistic Regression has the best accuracy (83.88%) and precision, followed by Gradient Boosting and SVM. The Decision Tree classifier scores poorly across most parameters, suggesting overfitting or insufficient pattern recognition skills, while Logistic regression exhibits balanced performance with the lowest Avg FPR (31.68%). With Logistic Regression and Naive Bayes outperforming the rest, ROC AUC values demonstrate the models' discriminatory capability.

Model Analysis

The best accuracy, precision, and ROC AUC are attained by logistic regression, which makes it a compelling option for use in this circumstance. Despite achieving comparable accuracy, SVM is less dependable for medical data due to its lower ROC AUC (58.08%) and higher Avg FPR. With the second-best ROC AUC (72.51%) and lowest Avg FPR, Naive Bayes strikes a good balance between performance metrics, showing robustness in classification. Because it either overfits or isn't sufficiently accurate for the dataset, Decision Tree performs noticeably worse. As demonstrated by its higher average FPR and lower ROC AUC (69.39%), gradient boosting performs slightly poorly in discrimination power than logistic regression, but it performs similarly in accuracy and recall.

Conclusion

Logistic Regression is the best model, with the highest accuracy (83.88%) and precision, as well as a high ROC AUC (73.07%) and a low average FPR (31.68%). However, performance measures across all models point to dataset limitations, such as insufficient size or class imbalance. For healthcare uses, these restrictions could affect prediction reliability. While Logistic Regression outperforms the other models, the results show that more data and improved dataset preparation are required to guarantee applicability for medical purposes. Without resolving these constraints, applying these models in real-life situations would be inappropriate and possibly harmful. In order to meet the high clinical standards, accurate validation and calibration are required.