In [130…

```
## Importing Libraries

import numpy as np  # Linear algebra operations
import pandas as pd  # Data processing and analysis
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf

from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix, roc_auc
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn import tree
from sklearn.svm import SVC
from tensorflow import keras
from tensorflow.keras import layers, Sequential
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.layers import Dense, Dropout, BatchNormalization, LSTM
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.naive_bayes import GaussianNB

import warnings
warnings.filterwarnings("ignore")
```

In [3]:

```
## Upload dataset

df = pd.read_csv('/Users/serenaygoler/heart disease.csv')

df.head() # Displays the first 5 rows.
```

Out[3]:

	Age	Sex	ChestPainType	RestingBP	Cholesterol	FastingBS	RestingECG	MaxHR
0	40	M	ATA	140	289	0	Normal	172
1	49	F	NAP	160	180	0	Normal	156
2	37	M	ATA	130	283	0	ST	98
3	48	F	ASY	138	214	0	Normal	108
4	54	M	NAP	150	195	0	Normal	122

In [6]:

```
df.tail() # Display the last 5 rows.
```

Out[6]:

	Age	Sex	ChestPainType	RestingBP	Cholesterol	FastingBS	RestingECG	MaxH
913	45	M	TA	110	264	0	Normal	13
914	68	M	ASY	144	193	1	Normal	14
915	57	M	ASY	130	131	0	Normal	1
916	57	F	ATA	130	236	0	LVH	17
917	38	M	NAP	138	175	0	Normal	17

In [8]:
```
df.info() # Prints name and type of variables, number of observations, and c
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 918 entries, 0 to 917
Data columns (total 12 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   Age             918 non-null    int64  
 1   Sex             918 non-null    object 
 2   ChestPainType   918 non-null    object 
 3   RestingBP       918 non-null    int64  
 4   Cholesterol     918 non-null    int64  
 5   FastingBS       918 non-null    int64  
 6   RestingECG      918 non-null    object 
 7   MaxHR           918 non-null    int64  
 8   ExerciseAngina  918 non-null    object 
 9   Oldpeak         918 non-null    float64
 10  ST_Slope        918 non-null    object 
 11  HeartDisease    918 non-null    int64  
dtypes: float64(1), int64(6), object(5)
memory usage: 86.2+ KB
```

In [10]:
```
df.shape # Displays the number of rows and columns in the dataset.
```

Out[10]: (918, 12)

In [12]:
```
df.isna().sum() # Counts missing values in each column.
```

Out[12]:
```
Age               0
Sex               0
ChestPainType     0
RestingBP         0
Cholesterol       0
FastingBS         0
RestingECG        0
MaxHR             0
ExerciseAngina    0
Oldpeak           0
ST_Slope          0
HeartDisease      0
dtype: int64
```

In [14]:
```
df.duplicated().sum() # Counts the number of duplicate rows.
```

Out[14]: 0

In [16]:
```
## Provides summary statistics for numeric columns, rounded to 2 decimals ar
df.describe().round(2).T
```

Out[16]:

	count	mean	std	min	25%	50%	75%	max
Age	918.0	53.51	9.43	28.0	47.00	54.0	60.0	77.0
RestingBP	918.0	132.40	18.51	0.0	120.00	130.0	140.0	200.0
Cholesterol	918.0	198.80	109.38	0.0	173.25	223.0	267.0	603.0
FastingBS	918.0	0.23	0.42	0.0	0.00	0.0	0.0	1.0
MaxHR	918.0	136.81	25.46	60.0	120.00	138.0	156.0	202.0
Oldpeak	918.0	0.89	1.07	-2.6	0.00	0.6	1.5	6.2
HeartDisease	918.0	0.55	0.50	0.0	0.00	1.0	1.0	1.0

In [18]:
```
# Count how many Cholesterol values are zero
chol_zero_count = (df["Cholesterol"] == 0).sum()

# Count how many RestingBP values are zero
bp_zero_count = (df["RestingBP"] == 0).sum()

print(f"Number of Cholesterol values equal to 0: {chol_zero_count}")
print(f"Number of RestingBP values equal to 0: {bp_zero_count}")
```

```
Number of Cholesterol values equal to 0: 172
Number of RestingBP values equal to 0: 1
```

In [20]:
```
# Cross-tabulate Cholesterol = 0 with HeartDisease status
import pandas as pd

zero_chol = df[df["Cholesterol"] == 0]
ct = pd.crosstab(zero_chol["HeartDisease"], zero_chol["Cholesterol"])
print(ct)
```

```
Cholesterol     0
HeartDisease
0              20
1             152
```

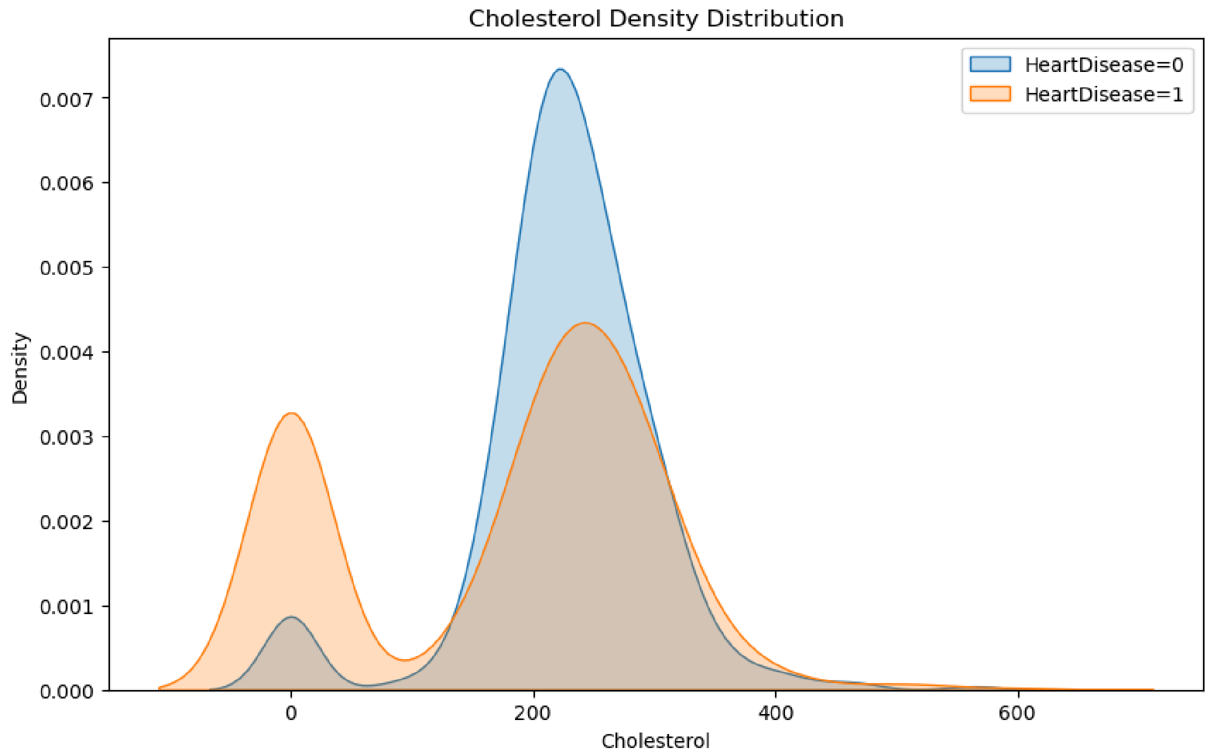
In [22]:
```
## Filters out rows where Cholesterol equals zero and returns summary statis

print(df[df["Cholesterol"] != 0]["Cholesterol"].describe())
```

```
count    746.000000
mean     244.635389
std       59.153524
min       85.000000
25%      207.250000
50%      237.000000
75%      275.000000
max      603.000000
Name: Cholesterol, dtype: float64
```

In [24]:
```
# With zeros included
print("=== With Zero values Included ===")
print(df.groupby("HeartDisease")["Cholesterol"].describe())

# Zeros removed
print("\n===  With zero values removed ===")
print(df[df["Cholesterol"] != 0].groupby("HeartDisease")["Cholesterol"].desc
```

```
=== With Zero values Included ===
              count        mean         std  min     25%    50%     75%    m
ax
HeartDisease
0             410.0  227.121951   74.634659  0.0  197.25  227.0  266.75  56
4.0
1             508.0  175.940945  126.391398  0.0    0.00  217.0  267.00  60
3.0

===  With zero values removed ===
              count        mean        std    min    25%    50%     75%    m
ax
HeartDisease
0             390.0  238.769231  55.394617   85.0  203.0  231.5  269.00  56
4.0
1             356.0  251.061798  62.462713  100.0  212.0  246.0  283.25  60
3.0
```

In [26]:
```
# Plot the cholesterol distribution for HeartDisease=0 and HeartDisease=1 us
plt.figure(figsize=(10,6))
sns.kdeplot(df[df["HeartDisease"]==0]["Cholesterol"], label="HeartDisease=0"
sns.kdeplot(df[df["HeartDisease"]==1]["Cholesterol"], label="HeartDisease=1"
plt.legend()
plt.title("Cholesterol Density Distribution")
plt.show()
```

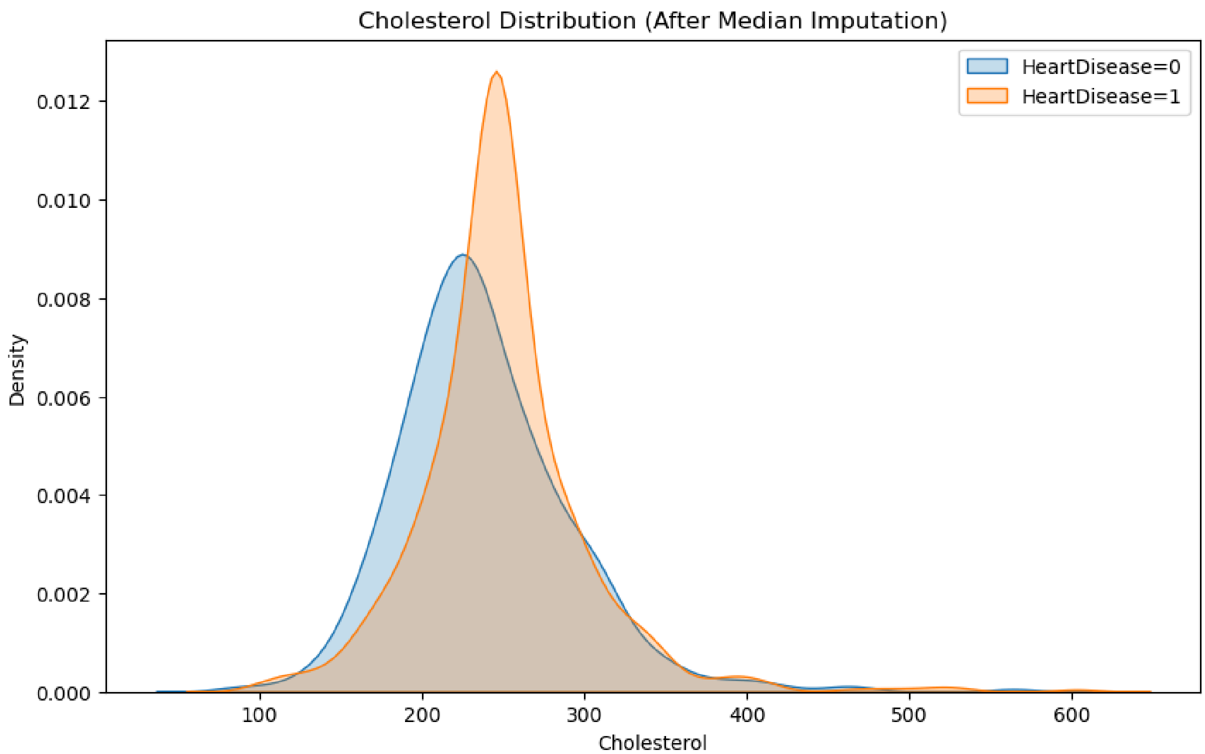


In [28]:

```
# This block cleans the dataset by:
# 1. Removing rows where RestingBP = 0 (unrealistic values).
# 2. Calculating group-wise medians of Cholesterol (by HeartDisease) excludi
# 3. Replacing Cholesterol values of zero with the corresponding group media
# 4. Checking that no zero values remain.
# 5. Displaying summary statistics of Cholesterol by HeartDisease after clea

df_clean = df.copy()
df_clean = df_clean[df_clean["RestingBP"] != 0].copy()

medians = (
    df_clean[df_clean["Cholesterol"] != 0]
    .groupby("HeartDisease")["Cholesterol"]
    .median()
)

mask_zero = df_clean["Cholesterol"] == 0
df_clean["Cholesterol"] = df_clean["Cholesterol"].astype(float)
df_clean.loc[mask_zero, "Cholesterol"] = (
    df_clean.loc[mask_zero, "HeartDisease"].map(medians)
)

print("Remaining zeros:", (df_clean["Cholesterol"] == 0).sum())
print(df_clean.groupby("HeartDisease")["Cholesterol"].describe())
```

```
Remaining zeros: 0
              count        mean        std    min    25%    50%     75%    m
ax
HeartDisease
0             410.0  238.414634  54.045994   85.0  204.0  231.5  266.75   56
4.0
1             507.0  249.554241  52.370323  100.0  225.0  246.0  267.00   60
3.0
```

In [30]:
```
# KDE plot — distribution comparison after median imputation
plt.figure(figsize=(10,6))
sns.kdeplot(df_clean[df_clean["HeartDisease"]==0]["Cholesterol"], label="Hea
sns.kdeplot(df_clean[df_clean["HeartDisease"]==1]["Cholesterol"], label="Hea
plt.title("Cholesterol Distribution (After Median Imputation)")
plt.xlabel("Cholesterol")
plt.ylabel("Density")
plt.legend()
plt.show()
```



In [32]:
```
## Provides summary statistics for numeric columns for clean data, rounded t
df_clean.describe().round(2).T
```

Out[32]:

	count	mean	std	min	25%	50%	75%	max
Age	917.0	53.51	9.44	28.0	47.0	54.0	60.0	77.0
RestingBP	917.0	132.54	18.00	80.0	120.0	130.0	140.0	200.0
Cholesterol	917.0	244.57	53.39	85.0	214.0	246.0	267.0	603.0
FastingBS	917.0	0.23	0.42	0.0	0.0	0.0	0.0	1.0
MaxHR	917.0	136.79	25.47	60.0	120.0	138.0	156.0	202.0
Oldpeak	917.0	0.89	1.07	-2.6	0.0	0.6	1.5	6.2
HeartDisease	917.0	0.55	0.50	0.0	0.0	1.0	1.0	1.0

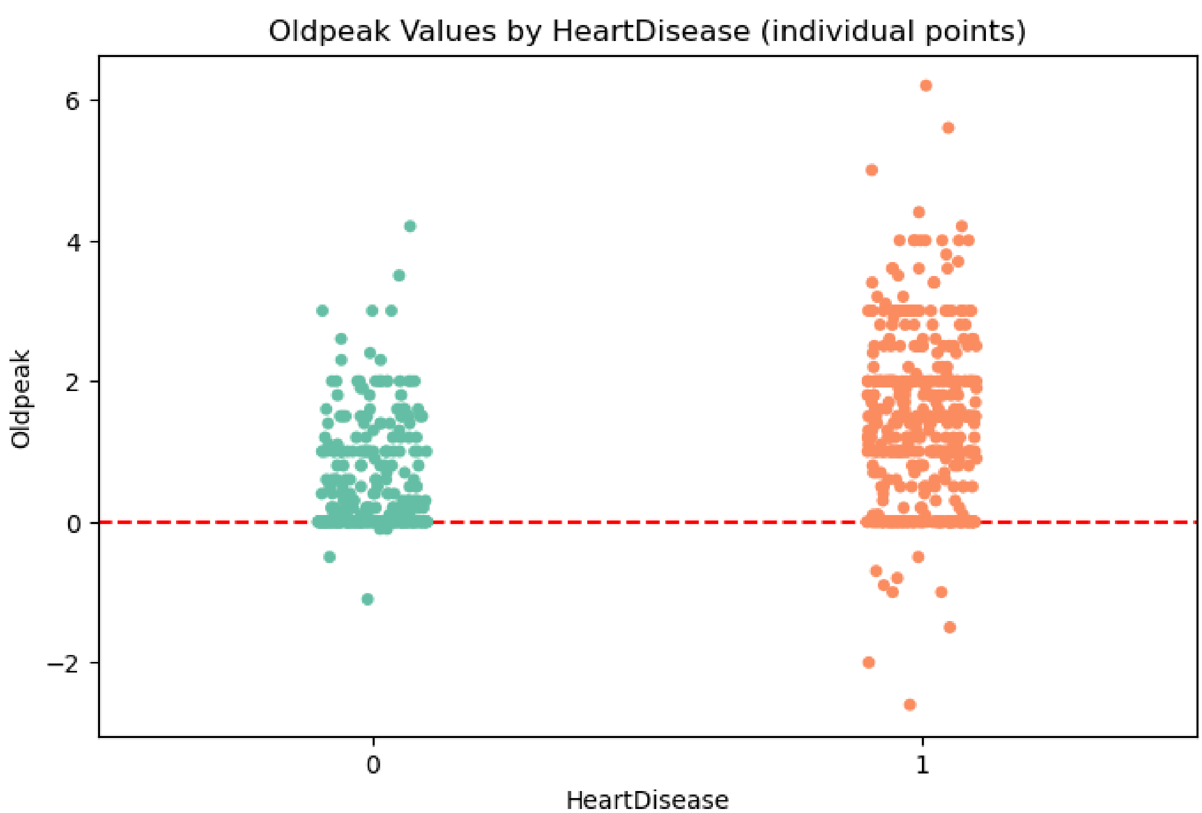
In [34]:

```
# Count negative Oldpeak values

neg_oldpeak_count = (df["Oldpeak"] < 0).sum()
print(f"Number of negative Oldpeak values: {neg_oldpeak_count}")
```

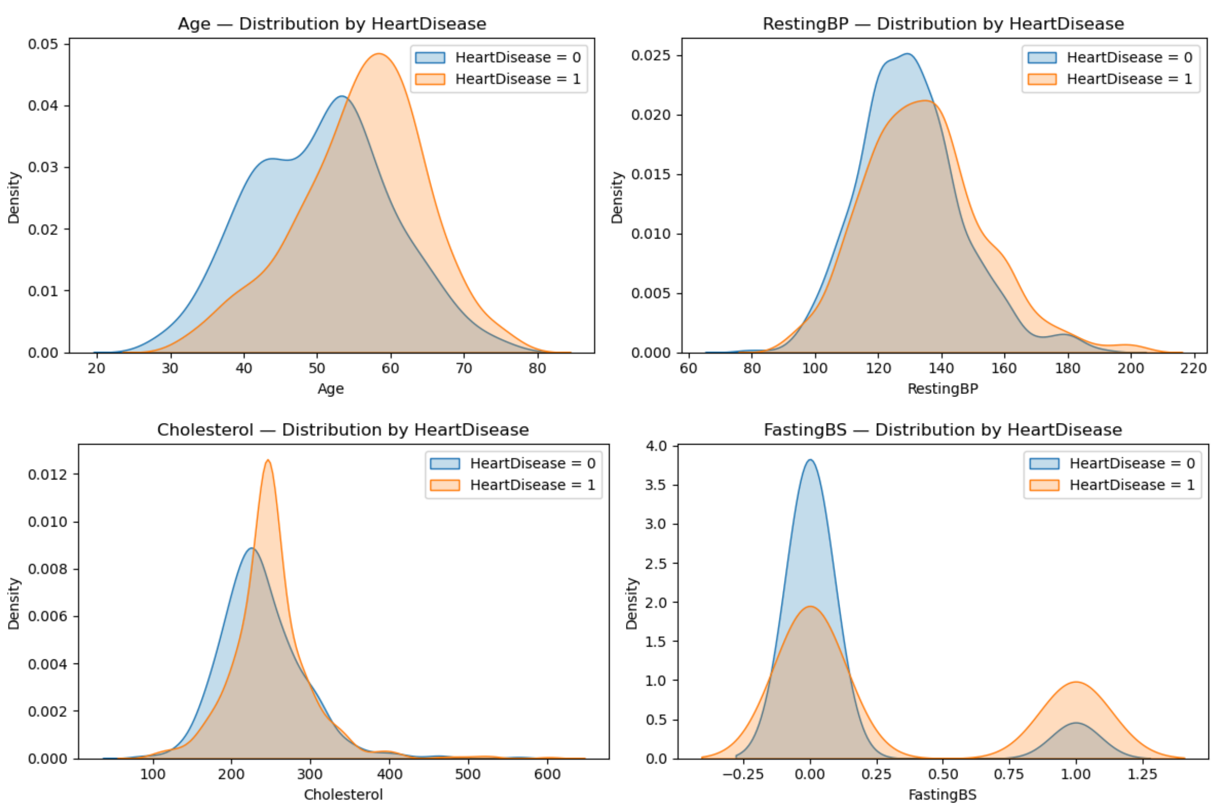
```
Number of negative Oldpeak values: 13
```

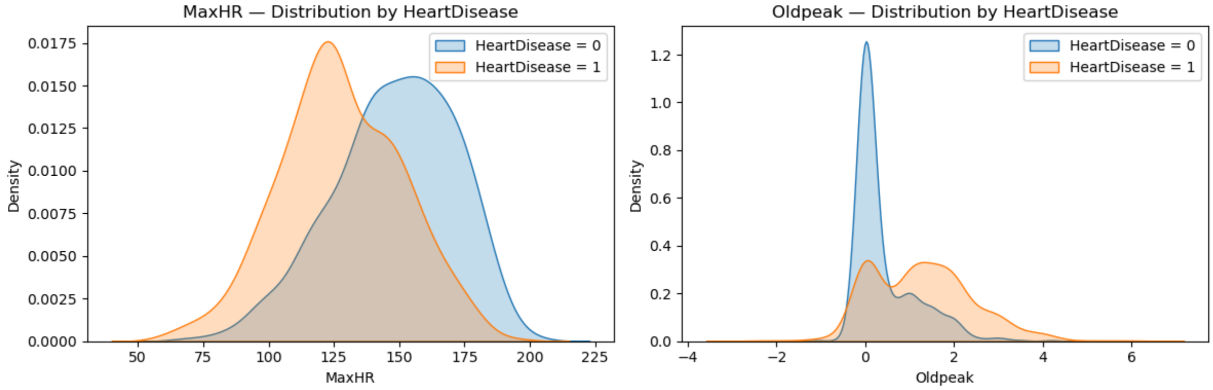
In [36]:

```
# Stripplot showing distribution of Oldpeak values by HeartDisease, with ref

plt.figure(figsize=(8,5))
sns.stripplot(x="HeartDisease", y="Oldpeak", data=df, jitter=True, palette="
plt.axhline(0, color="red", linestyle="--")
plt.title("Oldpeak Values by HeartDisease (individual points)")
plt.show()
```

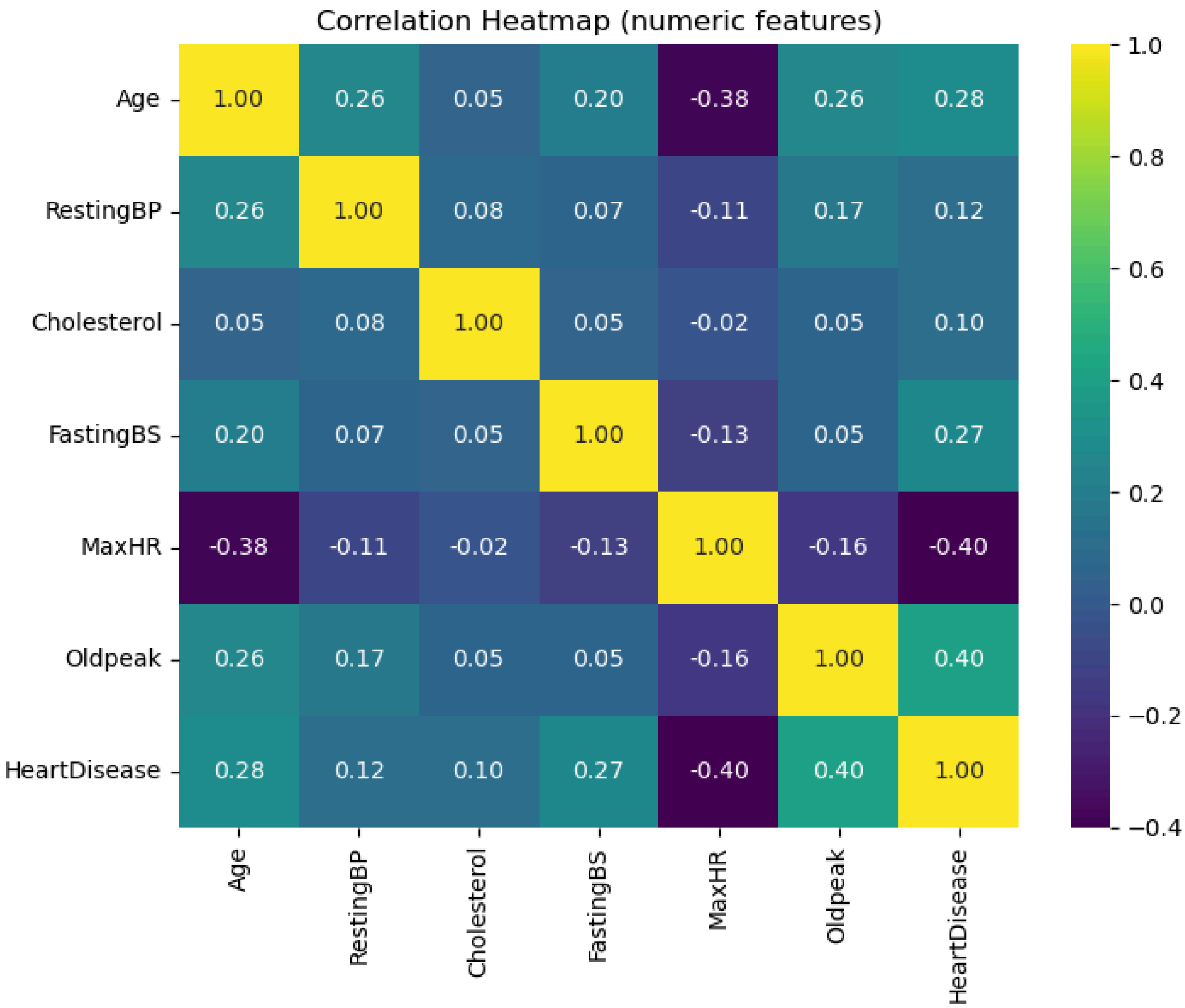


In [38]:
```python
# Plot numeric feature distributions by target, two-at-a-time
num_cols = df_clean.select_dtypes(include="number").columns.drop("HeartDisea
cols = list(num_cols)

for i in range(0, len(cols), 2):
    pair = cols[i:i+2]  # up to 2 columns per figure

    fig, axes = plt.subplots(1, len(pair), figsize=(12, 4))
    if len(pair) == 1:
        axes = [axes]  # make iterable if only one axis

    for ax, col in zip(axes, pair):
        sns.kdeplot(
            df_clean.loc[df_clean["HeartDisease"] == 0, col].dropna(),
            label="HeartDisease = 0", fill=True, ax=ax
        )
        sns.kdeplot(
            df_clean.loc[df_clean["HeartDisease"] == 1, col].dropna(),
            label="HeartDisease = 1", fill=True, ax=ax
        )
        ax.set_title(f"{col} — Distribution by HeartDisease")
        ax.set_xlabel(col); ax.set_ylabel("Density")
        ax.legend()



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



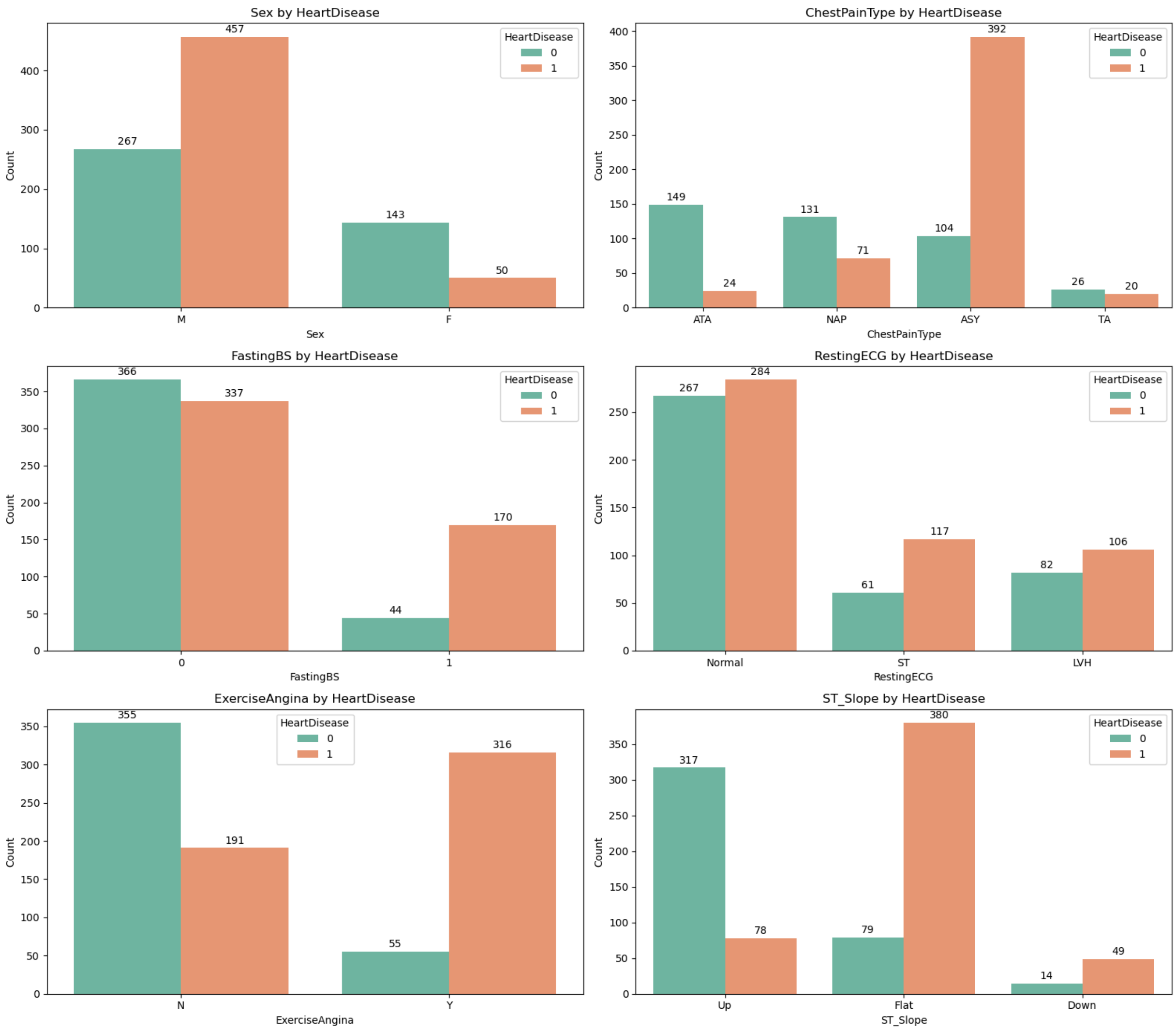


In [40]:
```python
# Select only numerical columns and to check correlation
num_cols = df_clean.select_dtypes(include=[np.number]).columns

plt.figure(figsize=(8,6))
sns.heatmap(df_clean[num_cols].corr(), annot=True, cmap="viridis", fmt=".2f"
plt.title("Correlation Heatmap (numeric features)")
plt.show()
```



In [42]:
```python
# Distribution of categorical variables by the target variable
cat_cols = ["Sex", "ChestPainType", "FastingBS", "RestingECG", "ExerciseAngi
fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(16, 14))
axes = axes.flatten()
```

```
for ax, col in zip(axes, cat_cols):
    g = sns.countplot(data=df_clean, x=col, hue="HeartDisease", palette="Set
    ax.set_title(f"{col} by HeartDisease")
    ax.set_xlabel(col); ax.set_ylabel("Count")
    # label name
    for c in g.containers:
        g.bar_label(c, padding=2, fmt="%.0f")

# Remove extra axes
for ax in axes[len(cat_cols):]:
    fig.delaxes(ax)

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



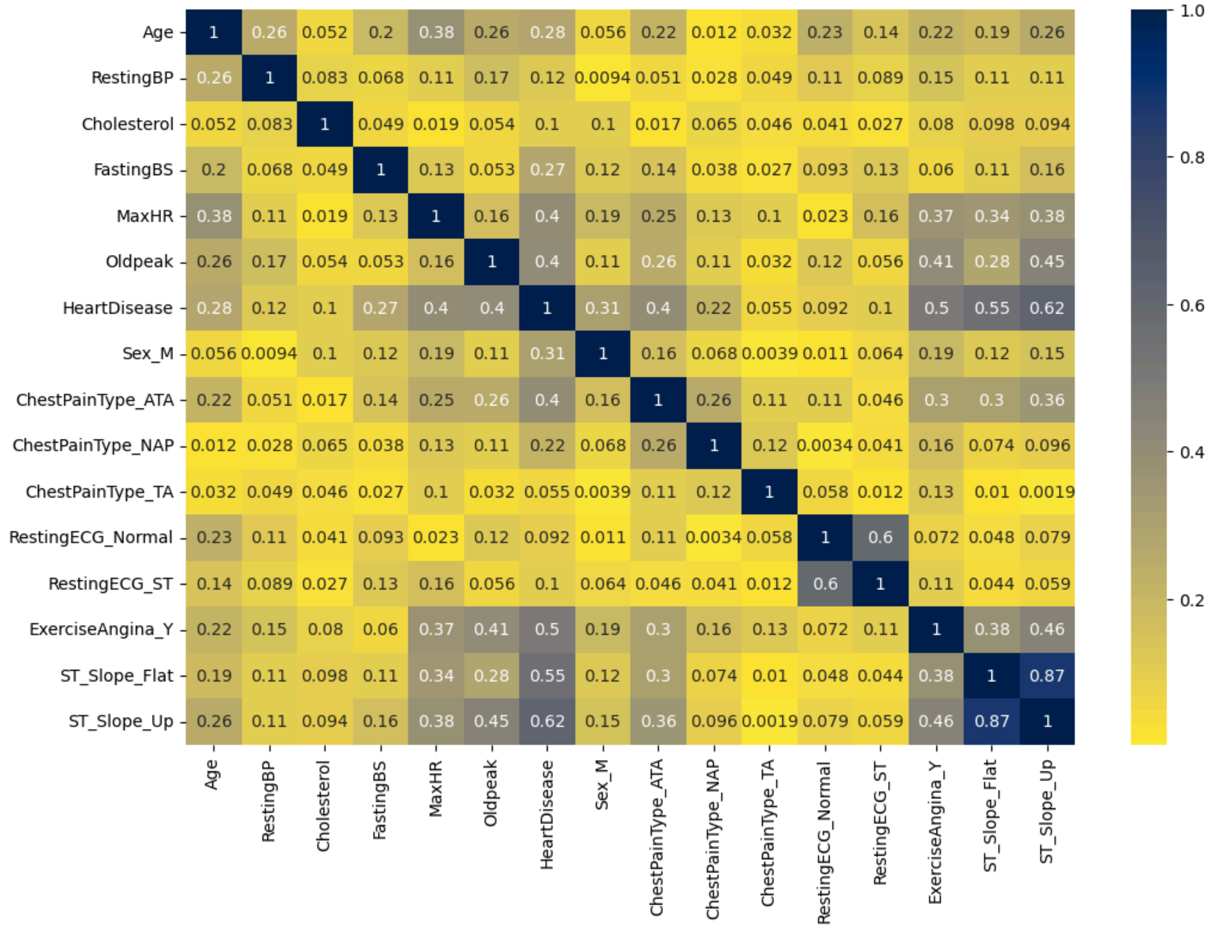
In [44]:
```
# One-hot encoding was applied to transform categorical variables into dummy
DUMMY = pd.get_dummies(df_clean, drop_first=True)
DUMMY.head()
```

Out[44]:

	Age	RestingBP	Cholesterol	FastingBS	MaxHR	Oldpeak	HeartDisease	Sex_M
0	40	140	289.0	0	172	0.0	0	True
1	49	160	180.0	0	156	1.0	1	False
2	37	130	283.0	0	98	0.0	0	True
3	48	138	214.0	0	108	1.5	1	False
4	54	150	195.0	0	122	0.0	0	True

In [46]:

```
## Compute absolute pairwise correlations (after one-hot encoding) and visua
correlations = abs(DUMMY.corr())
plt.figure(figsize=(12,8))
sns.heatmap(correlations, annot=True, cmap="cividis_r")
plt.show()
```



In [48]:

```
# Create a copy of the cleaned dataset
codedf = df_clean.copy()

# 1) Convert binary categorical columns into 0/1 format
if codedf['Sex'].dtype == 'object':
    codedf['Sex'] = codedf['Sex'].str.strip().map({'F': 0, 'M': 1}).astype('

if codedf['ExerciseAngina'].dtype == 'object':
    codedf['ExerciseAngina'] = codedf['ExerciseAngina'].str.strip().map({'N'
```

```python
# (If they are already boolean True/False, convert them to integers)
for col in ['Sex', 'ExerciseAngina']:
    if codedf[col].dtype == 'bool':
        codedf[col] = codedf[col].astype(int)

# 2) Apply one-hot encoding for multi-class categorical columns
to_onehot = ['ChestPainType', 'RestingECG', 'ST_Slope']
codedf = pd.get_dummies(codedf, columns=to_onehot, drop_first=True)

# Convert any remaining boolean columns into 0/1 integers
for col in codedf.select_dtypes(include='bool').columns:
    codedf[col] = codedf[col].astype(int)

codedf.dtypes
```

Out[48]:

```
Age                    int64
Sex                    Int64
RestingBP              int64
Cholesterol          float64
FastingBS              int64
MaxHR                  int64
ExerciseAngina         Int64
Oldpeak              float64
HeartDisease           int64
ChestPainType_ATA      int64
ChestPainType_NAP      int64
ChestPainType_TA       int64
RestingECG_Normal      int64
RestingECG_ST          int64
ST_Slope_Flat          int64
ST_Slope_Up            int64
dtype: object
```

In [50]:

```python
# Standardize continuous variables (mean = 0, std = 1)
# This ensures that all numeric predictors are on the same scale,
# which is especially important for distance-based algorithms (e.g., KNN, SV
numcolsc = ['Age','RestingBP','Cholesterol','MaxHR','Oldpeak']
scaler = StandardScaler()
codedf[numcolsc] = scaler.fit_transform(codedf[numcolsc])

codedf.head()
```

Out[50]:

	Age	Sex	RestingBP	Cholesterol	FastingBS	MaxHR	ExerciseAngina	Ol
0	-1.432206	1	0.414627	0.832639	0	1.383339	0	-0.8
1	-0.478057	0	1.526360	-1.210238	0	0.754736	0	0.1
2	-1.750256	1	-0.141240	0.720187	0	-1.523953	0	-0.8
3	-0.584074	0	0.303453	-0.573010	0	-1.131075	1	0.5
4	0.052026	1	0.970493	-0.929108	0	-0.581047	0	-0.8

Machine Learning

In [53]:
```
# Split the dataset into features (X) and target (y)
X = codedf.drop(columns=["HeartDisease"])
y = codedf["HeartDisease"]

# Train-test split: 80% training, 20% testing
# Stratify ensures the target class distribution (0/1) is preserved in both
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)
```

In [55]:
```
X_test.shape , y_test.shape # Check the shape of the test sets
```

Out[55]:
```
((184, 15), (184,))
```

Logistic Regression

In [58]:
```
# Logistic Regression Model
# max_iter=1000 ensures convergence during optimization
logreg = LogisticRegression(max_iter=1000)

# Train the model on the training set
logreg.fit(X_train, y_train)

# Make predictions on the test set
y_pred = logreg.predict(X_test)

# Calculate accuracy of the model
logregAcc = accuracy_score(y_test, y_pred)
logregAcc
```

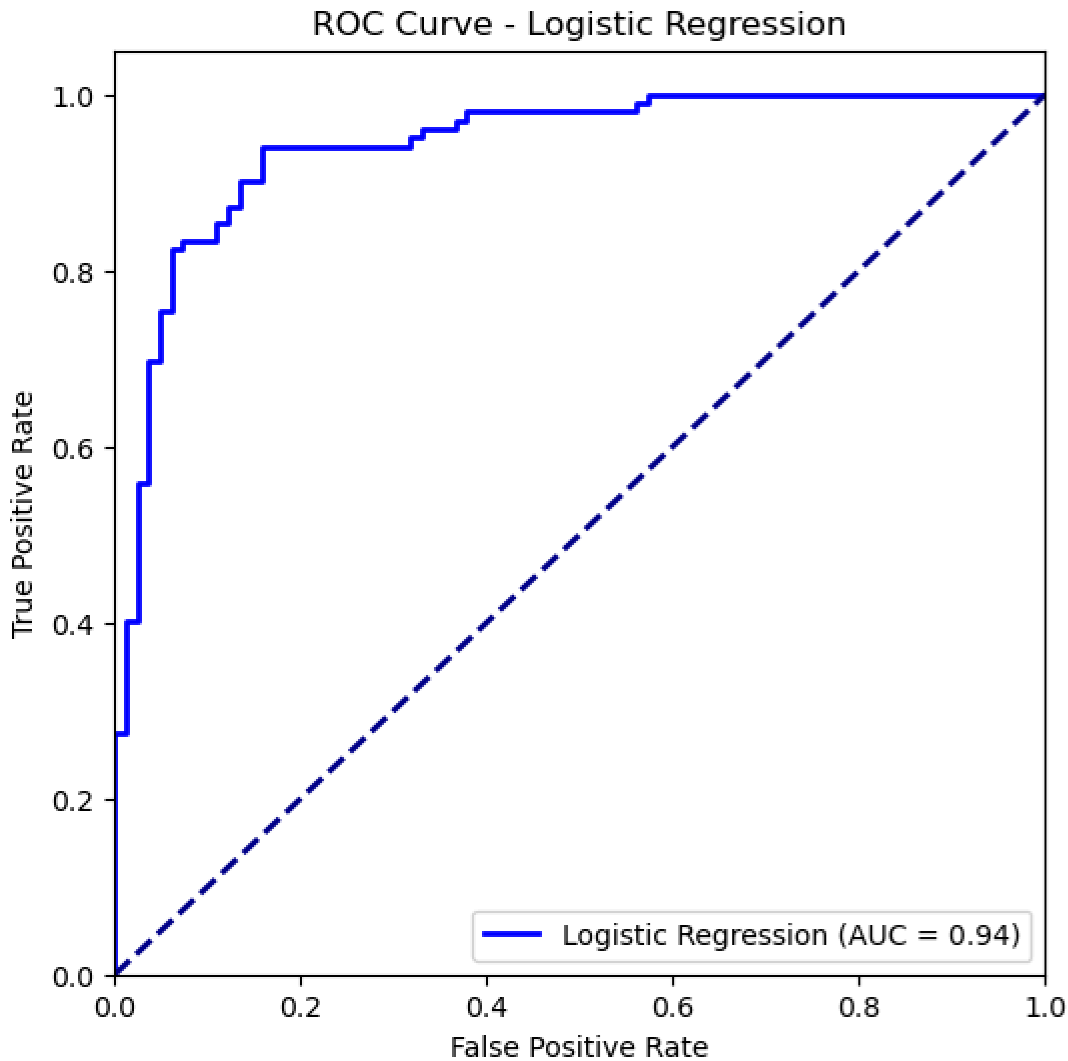
Out[58]:
```
0.8858695652173914
```

In [60]:
```
# Generate a detailed classification report
# Includes precision, recall, f1-score, and support for each class
print("\nClassification Report:\n", classification_report(y_test, y_pred))

# Predict probabilities for the positive class (1 = Heart Disease)
y_proba = logreg.predict_proba(X_test)[:, 1]

# Calculate the ROC AUC score to evaluate the model's discriminative ability
print("ROC AUC:", roc_auc_score(y_test, y_proba))
```

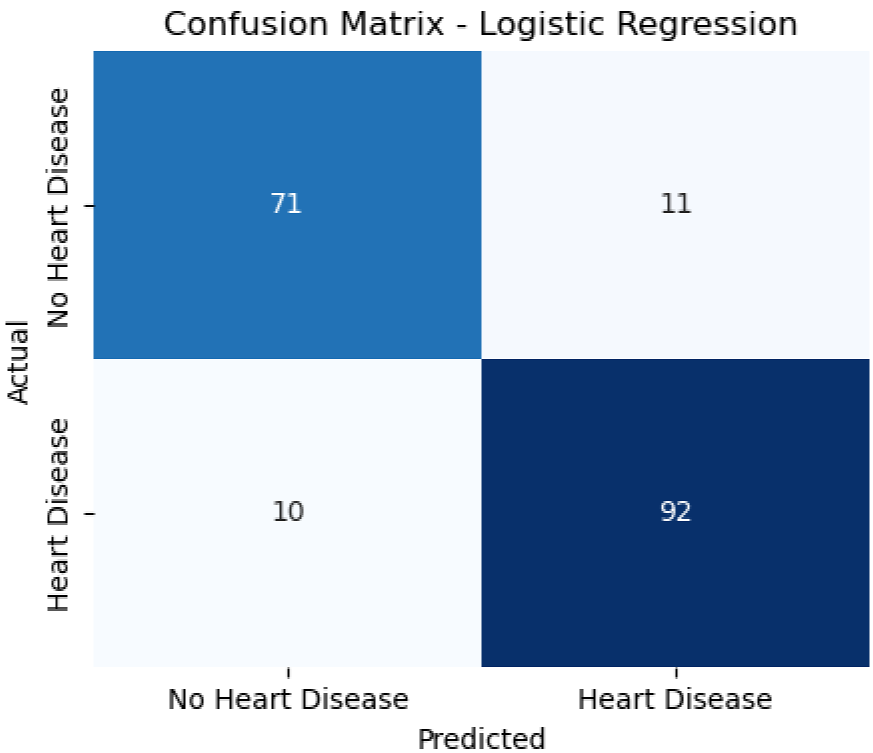
```
Classification Report:
               precision    recall  f1-score   support

           0       0.88      0.87      0.87        82
           1       0.89      0.90      0.90       102

    accuracy                           0.89       184
   macro avg       0.88      0.88      0.88       184
weighted avg       0.89      0.89      0.89       184

ROC AUC: 0.9423720707795313
```

In [62]:

```
from sklearn.metrics import roc_curve, roc_auc_score

# Probability predictions for positive class
y_proba = logreg.predict_proba(X_test)[:,1]

# ROC curve values
fpr, tpr, thresholds = roc_curve(y_test, y_proba)
roc_auc = roc_auc_score(y_test, y_proba)

# ROC curve plot
plt.figure(figsize=(6, 6))
plt.plot(fpr, tpr, color='blue', lw=2,
         label='Logistic Regression (AUC = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='darkblue', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve - Logistic Regression')
plt.legend(loc="lower right")
plt.show()
```



In [64]:
```
## Confusion Matrix
cm = confusion_matrix(y_test, y_pred)

plt.figure(figsize=(5,4))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", cbar=False,
            xticklabels=['No Heart Disease','Heart Disease'],
            yticklabels=['No Heart Disease','Heart Disease'])
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix - Logistic Regression")
plt.show()
```



In [66]:
```
# Create a dataframe of Logistic Regression coefficients
# This shows the direction (+/−) and relative magnitude of each feature's ef
# Positive coefficients → increase likelihood of heart disease
# Negative coefficients → decrease likelihood of heart disease

coefficients = pd.DataFrame({
    'Feature': X_train.columns,
    'Coefficient': logreg.coef_[0]
}).sort_values(by='Coefficient', ascending=False)

coefficients
```

Out[66]:

	Feature	Coefficient
1	Sex	1.358389
4	FastingBS	1.156245
13	ST_Slope_Flat	0.948000
6	ExerciseAngina	0.828883
7	Oldpeak	0.322847
3	Cholesterol	0.074785
0	Age	0.039656
2	RestingBP	-0.008738
12	RestingECG_ST	-0.173987
5	MaxHR	-0.268010
11	RestingECG_Normal	-0.309394
14	ST_Slope_Up	-1.280401
10	ChestPainType_TA	-1.329603
9	ChestPainType_NAP	-1.485096
8	ChestPainType_ATA	-1.554887

SUPPORT VECTOR MACHINE

In [69]:
```
# Linear SVM
svm_linear = SVC(kernel="linear", probability=True, random_state=1)
svm_linear.fit(X_train, y_train)
svm_linearAcc = accuracy_score(y_test, svm_linear.predict(X_test))
print("Linear SVM Accuracy:", svm_linearAcc)

# RBF SVM (non-linear)
svm_rbf = SVC(kernel="rbf", probability=True, random_state=1)
svm_rbf.fit(X_train, y_train)
svm_rbfAcc = accuracy_score(y_test, svm_rbf.predict(X_test))
print("RBF SVM Accuracy:", svm_rbfAcc)
```

```
Linear SVM Accuracy: 0.8586956521739131
RBF SVM Accuracy: 0.8858695652173914
```

In [71]:
```
print("\nClassification Report:\n", classification_report(y_test, y_pred))

# Note:
# The ROC curve and confusion matrix visuals are not repeated here for SVM,
# as their performance and outputs were nearly identical to Logistic Regress
```

```
Classification Report:
               precision    recall  f1-score   support

           0       0.88      0.87      0.87        82
           1       0.89      0.90      0.90       102

    accuracy                           0.89       184
   macro avg       0.88      0.88      0.88       184
weighted avg       0.89      0.89      0.89       184
```

DECISION TREE

In [74]:
```
# Build and train a Decision Tree model
clf = tree.DecisionTreeClassifier(random_state=0)  # reproducibility ensured
clf.fit(X_train, y_train)  # fit the model to training data

# Make predictions on the test set
y_pred = clf.predict(X_test)

# Calculate accuracy on the test set
clfAcc = accuracy_score(y_test, y_pred)
clfAcc
```

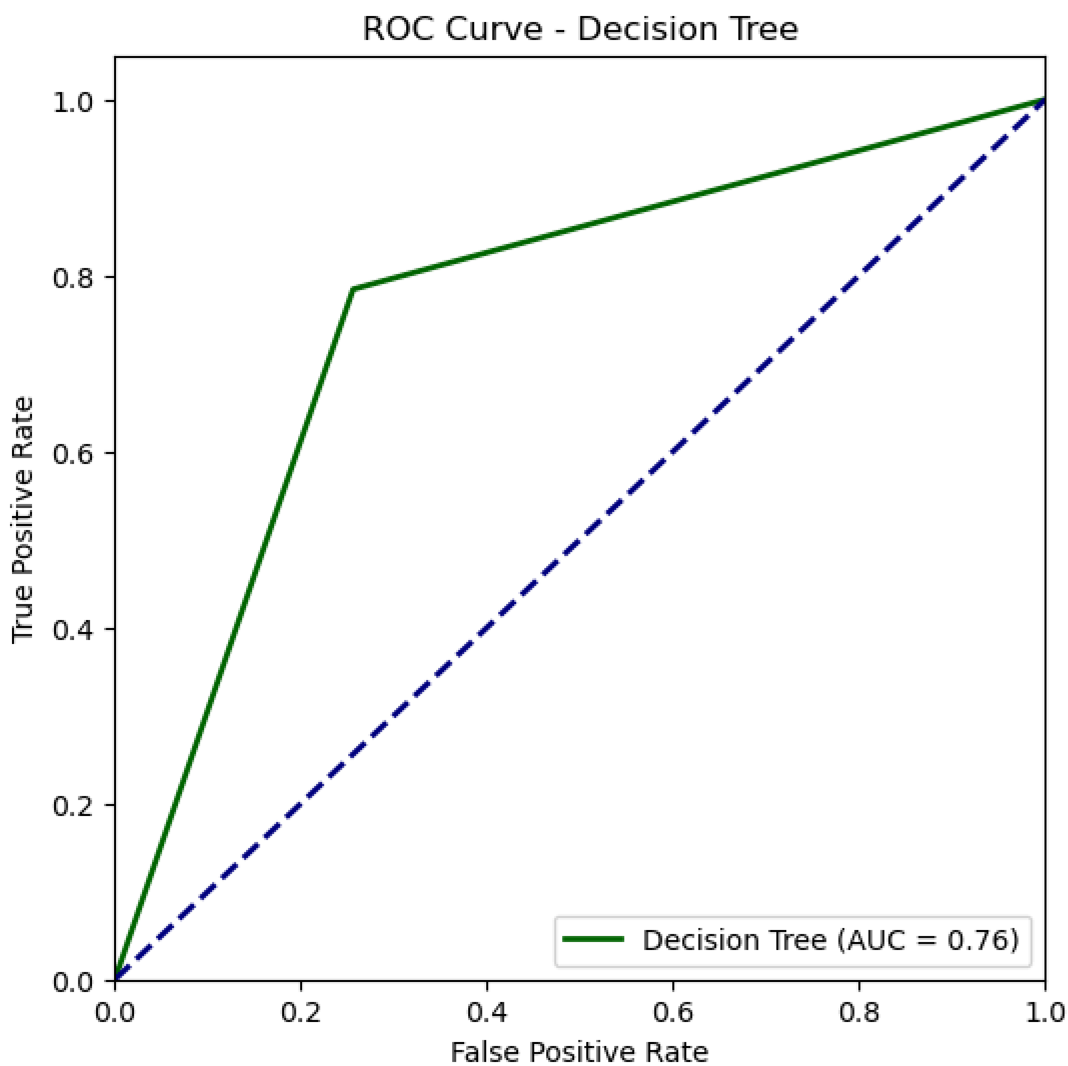
Out[74]: 0.7663043478260869

In [76]:
```
# Print classification metrics and ROC AUC for the Decision Tree
print("\nClassification Report:\n", classification_report(y_test, y_pred))

# Predict probabilities for the positive class (heart disease = 1)
y_proba = clf.predict_proba(X_test)[:, 1]

# Calculate and print ROC AUC score
print("ROC AUC:", roc_auc_score(y_test, y_proba))
```

```
Classification Report:
               precision    recall  f1-score   support

           0       0.73      0.74      0.74        82
           1       0.79      0.78      0.79       102

    accuracy                           0.77       184
   macro avg       0.76      0.76      0.76       184
weighted avg       0.77      0.77      0.77       184

ROC AUC: 0.7641080822572931
```

In [78]:
```
# Predict probability estimates for the positive class (heart disease = 1)
y_proba_tree = clf.predict_proba(X_test)[:, 1]

# Compute ROC curve values
fpr, tpr, thresholds = roc_curve(y_test, y_proba_tree)

# Calculate AUC (Area Under the Curve)
roc_auc_tree = roc_auc_score(y_test, y_proba_tree)
```

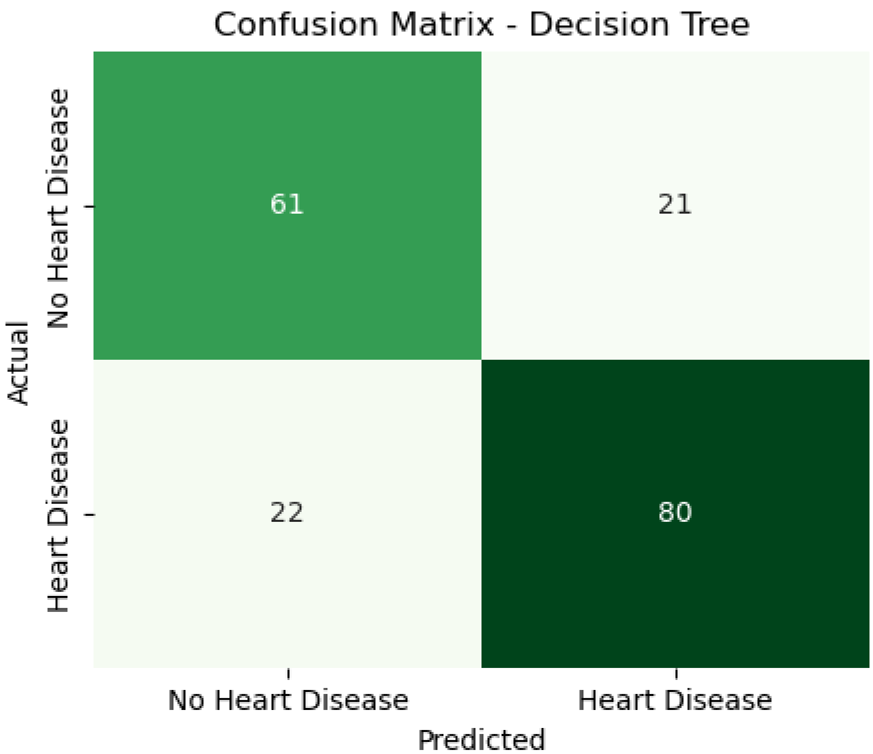
```
# Plot ROC curve for Decision Tree
plt.figure(figsize=(6, 6))
plt.plot(fpr, tpr, color='darkgreen', lw=2,
         label='Decision Tree (AUC = %0.2f)' % roc_auc_tree)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve - Decision Tree')
plt.legend(loc="lower right")
plt.show()

# Print final AUC value
print("ROC AUC (Decision Tree):", roc_auc_tree)
```




```
ROC AUC (Decision Tree): 0.7641080822572931
```

In [80]:
```
# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)

plt.figure(figsize=(5,4))
sns.heatmap(cm, annot=True, fmt="d", cmap="Greens", cbar=False,
```

```
            xticklabels=['No Heart Disease','Heart Disease'],
            yticklabels=['No Heart Disease','Heart Disease'])
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix - Decision Tree")
plt.show()
```



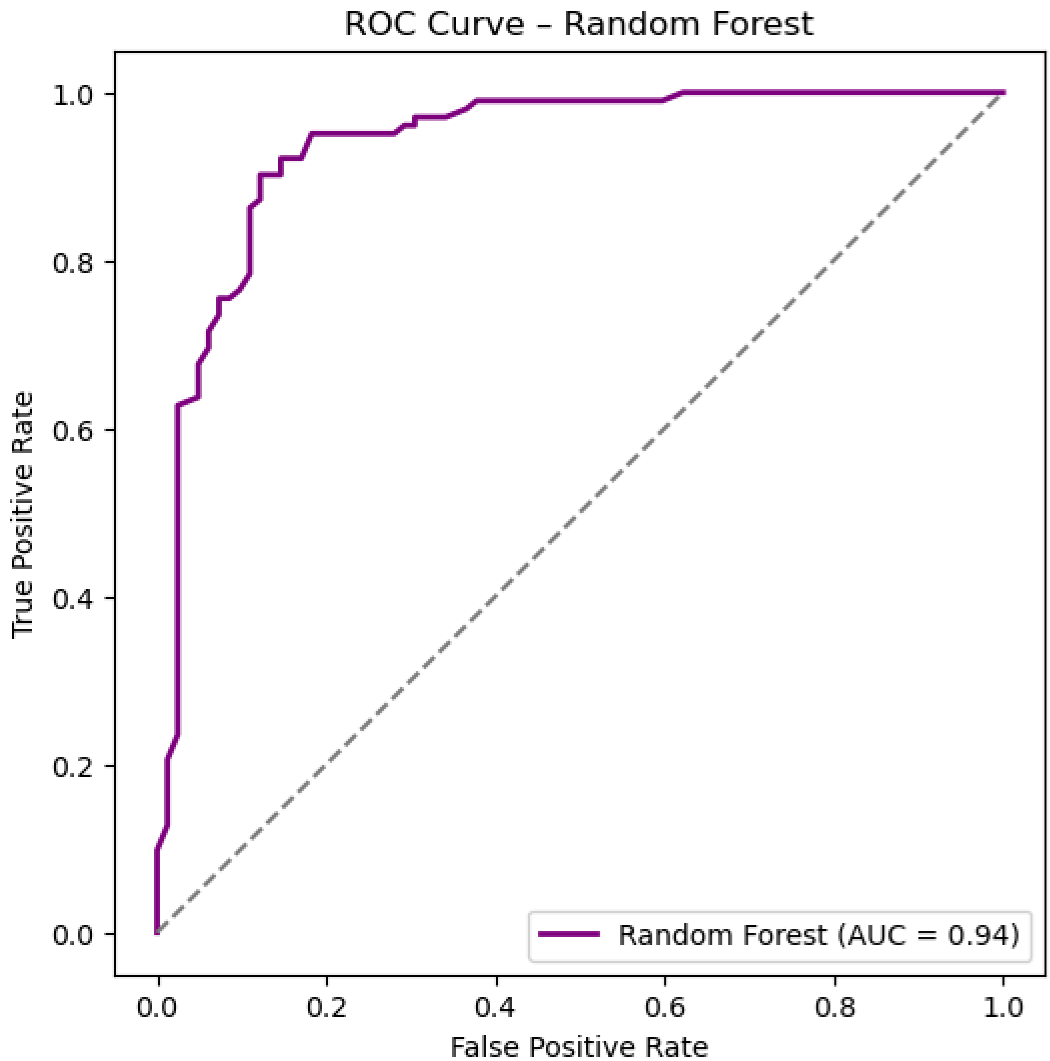
RANDOM FOREST

In [83]:
```
# 1) Build and train the model
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)

# 2) Predictions on the test set
y_pred = rf.predict(X_test)
y_proba = rf.predict_proba(X_test)[:, 1]  # probability for the positive cla

# 3) Evaluation
rfAcc = accuracy_score(y_test, y_pred)          # store accuracy
roc_auc_rf = roc_auc_score(y_test, y_proba)     # store AUC
report = classification_report(y_test, y_pred) # precision/recall/F1 per cla

print("Random Forest Accuracy:", rfAcc)
print("\nClassification Report:\n", report)
print("ROC AUC:", roc_auc_rf)
```

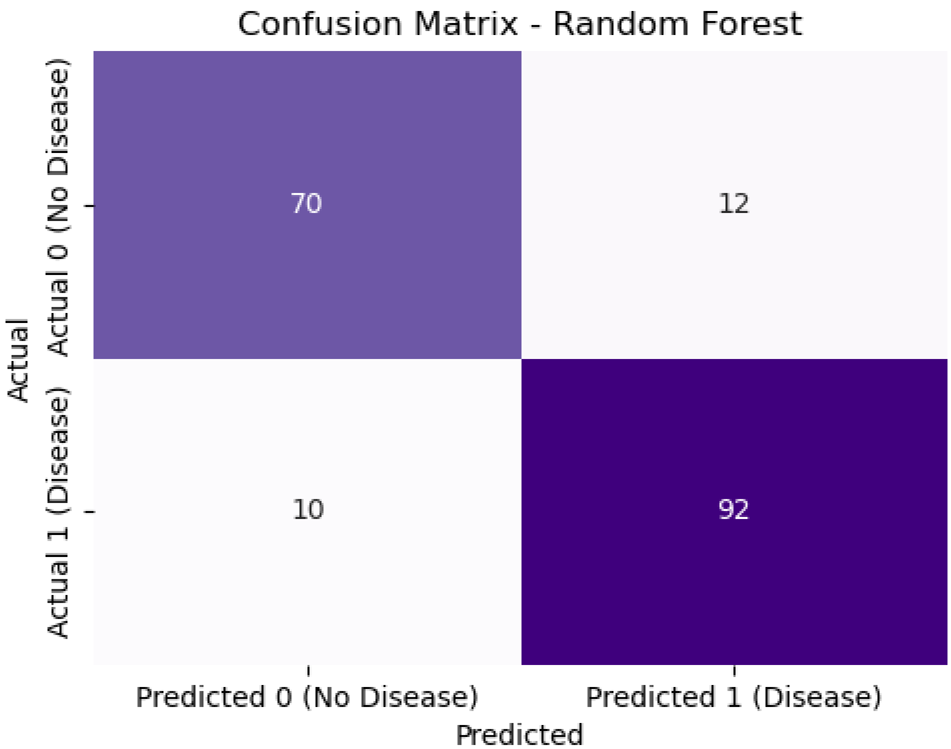
```
Random Forest Accuracy: 0.8804347826086957

Classification Report:
               precision    recall  f1-score   support

           0       0.88      0.85      0.86        82
           1       0.88      0.90      0.89       102

    accuracy                           0.88       184
   macro avg       0.88      0.88      0.88       184
weighted avg       0.88      0.88      0.88       184

ROC AUC: 0.93824725011956
```

In [85]:

```
# ROC Curve for Random Forest

y_proba_rf = rf.predict_proba(X_test)[:, 1]
fpr, tpr, _ = roc_curve(y_test, y_proba_rf)
auc_rf = roc_auc_score(y_test, y_proba_rf)

plt.figure(figsize=(6,6))
plt.plot(fpr, tpr, color="purple", lw=2, label=f"Random Forest (AUC = {auc_r
plt.plot([0,1],[0,1],'--', color='gray')
plt.xlabel("False Positive Rate"); plt.ylabel("True Positive Rate")
plt.title("ROC Curve – Random Forest"); plt.legend(loc="lower right")
plt.show()
```

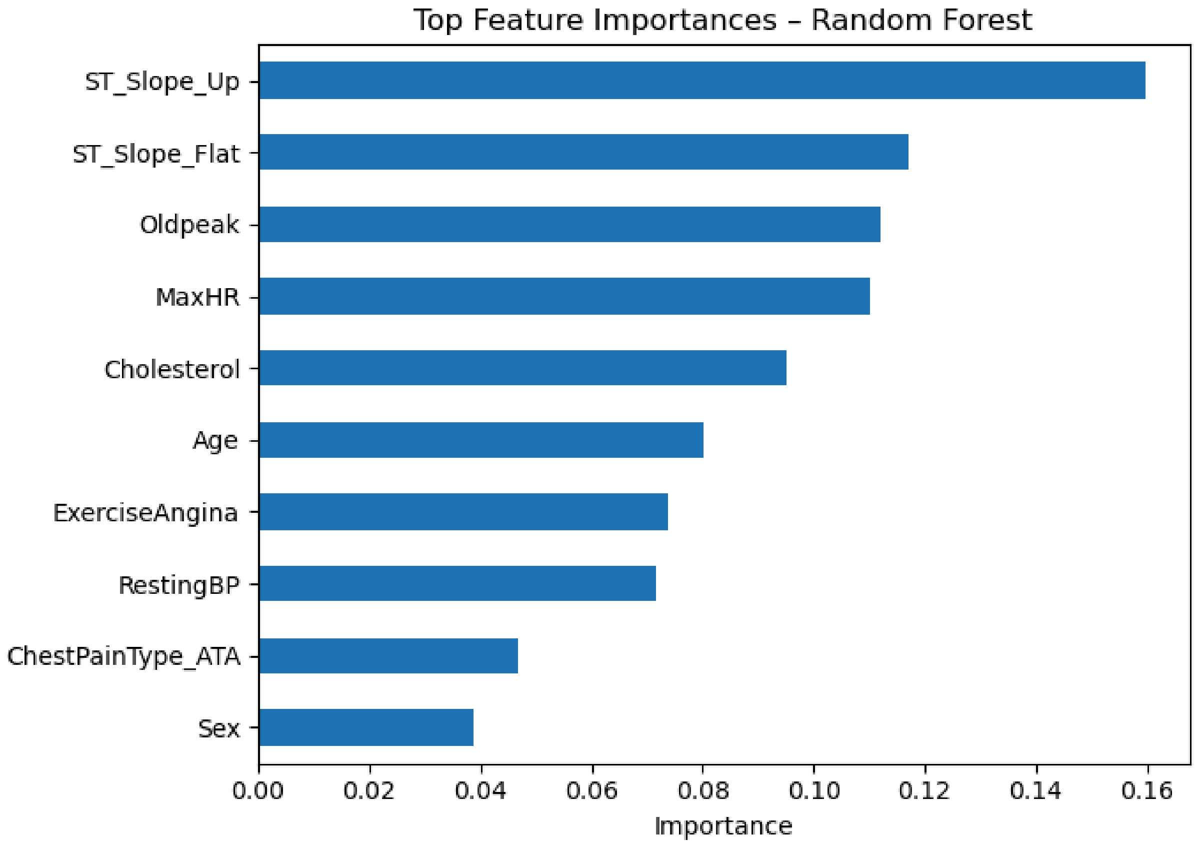


In [87]:
```
# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)

plt.figure(figsize=(5,4))
sns.heatmap(cm, annot=True, fmt="d", cmap="Purples", cbar=False,
            xticklabels=["Predicted 0 (No Disease)", "Predicted 1 (Disease)"
            yticklabels=["Actual 0 (No Disease)", "Actual 1 (Disease)"])

plt.title("Confusion Matrix – Random Forest")
plt.ylabel("Actual")
plt.xlabel("Predicted")
plt.tight_layout()
plt.show()
```



In [89]:

```
# Extract and sort feature importances from the trained Random Forest model
fi = pd.Series(rf.feature_importances_, index=X_train.columns).sort_values()

# Select the top 10 most important features
top = fi.tail(10)

# Plot horizontal bar chart of feature importances
plt.figure(figsize=(7,5))
top.plot(kind="barh")
plt.title("Top Feature Importances – Random Forest")
plt.xlabel("Importance")
plt.tight_layout()
plt.show()
```



KNN

In [92]:
```
# Define the range of k values to test
k_values = range(1, 21)
cv_scores = []

# Loop through each k and perform 5-fold cross-validation
for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn, X_train, y_train, cv=5, scoring='accuracy'
    cv_scores.append(scores.mean())

# Identify the k with the highest mean accuracy
best_k = k_values[cv_scores.index(max(cv_scores))]

print("Best k:", best_k)
```

```
Best k: 10
```

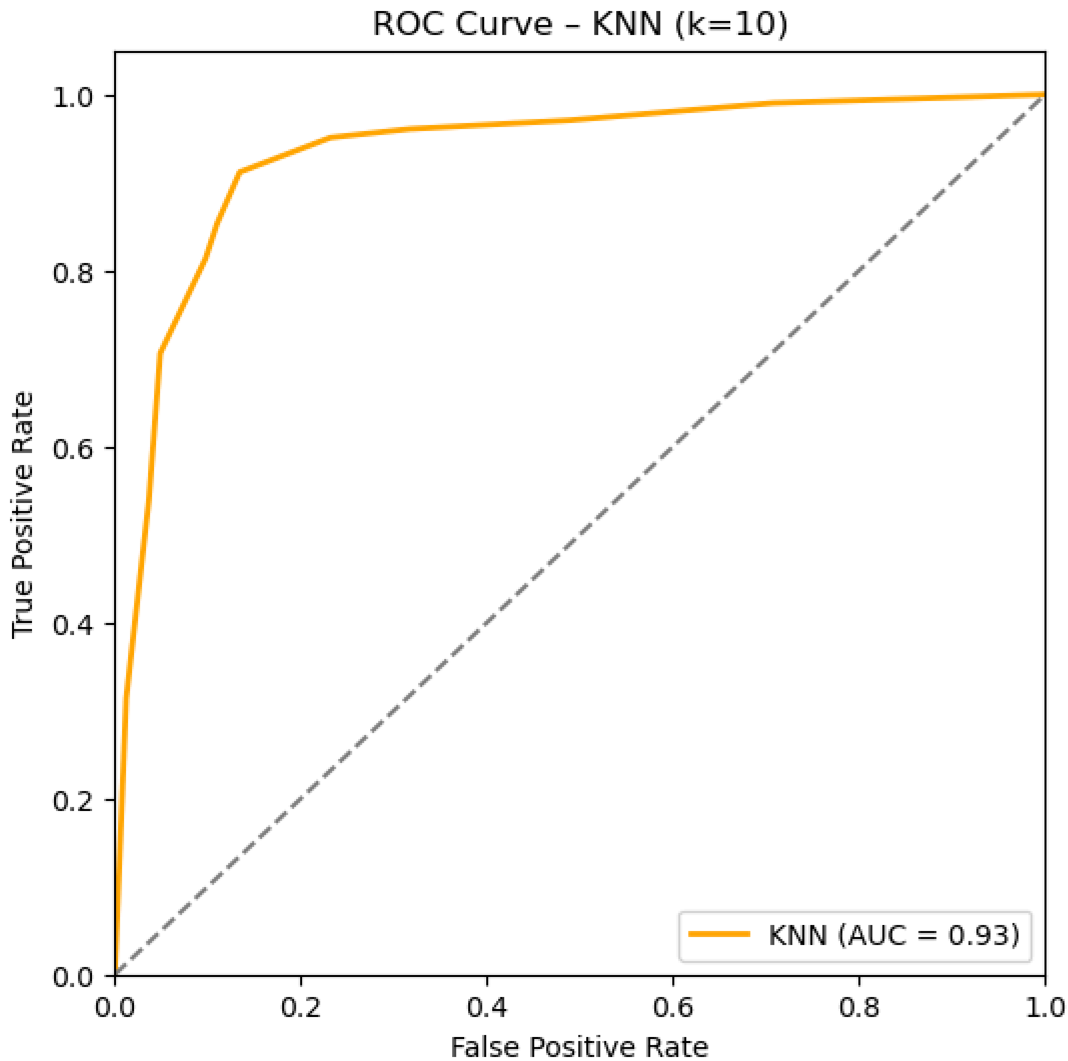
In [94]:
```
# Building a model using KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors = 10)
knn.fit(X_train, y_train)

y_pred = knn.predict(X_test)
knnAcc = accuracy_score(y_test,y_pred)
knnAcc
```

Out[94]: 0.8695652173913043

In [96]:
```
# Generate probability predictions for the positive class
y_proba = knn.predict_proba(X_test)[:, 1]

# Evaluate the model with ROC AUC and classification report
print("ROC AUC:", roc_auc_score(y_test, y_proba))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

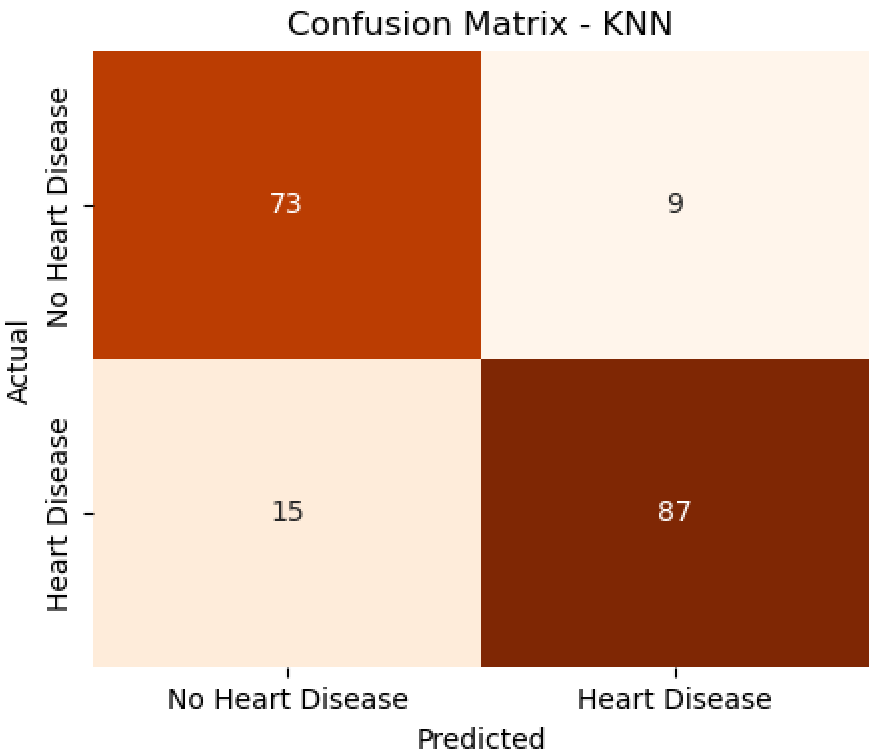
```
ROC AUC: 0.9324485891917742

Classification Report:
               precision    recall  f1-score   support

           0       0.83      0.89      0.86        82
           1       0.91      0.85      0.88       102

    accuracy                           0.87       184
   macro avg       0.87      0.87      0.87       184
weighted avg       0.87      0.87      0.87       184
```

In [98]:
```
# Probability predictions for the positive class
y_proba_knn = knn.predict_proba(X_test)[:, 1]

# ROC curve and AUC
fpr, tpr, _ = roc_curve(y_test, y_proba_knn)
auc_knn = roc_auc_score(y_test, y_proba_knn)

# Plot ROC curve
plt.figure(figsize=(6,6))
plt.plot(fpr, tpr, color='orange', lw=2, label="KNN (AUC = %0.2f)" % auc_knn
plt.plot([0,1],[0,1],'--', color='gray')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve – KNN (k=10)")
plt.legend(loc="lower right")
plt.show()
```



In [100…

```
# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)

plt.figure(figsize=(5,4))
sns.heatmap(cm, annot=True, fmt="d", cmap="Oranges", cbar=False,
            xticklabels=['No Heart Disease','Heart Disease'],
            yticklabels=['No Heart Disease','Heart Disease'])
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix – KNN")
plt.show()
```



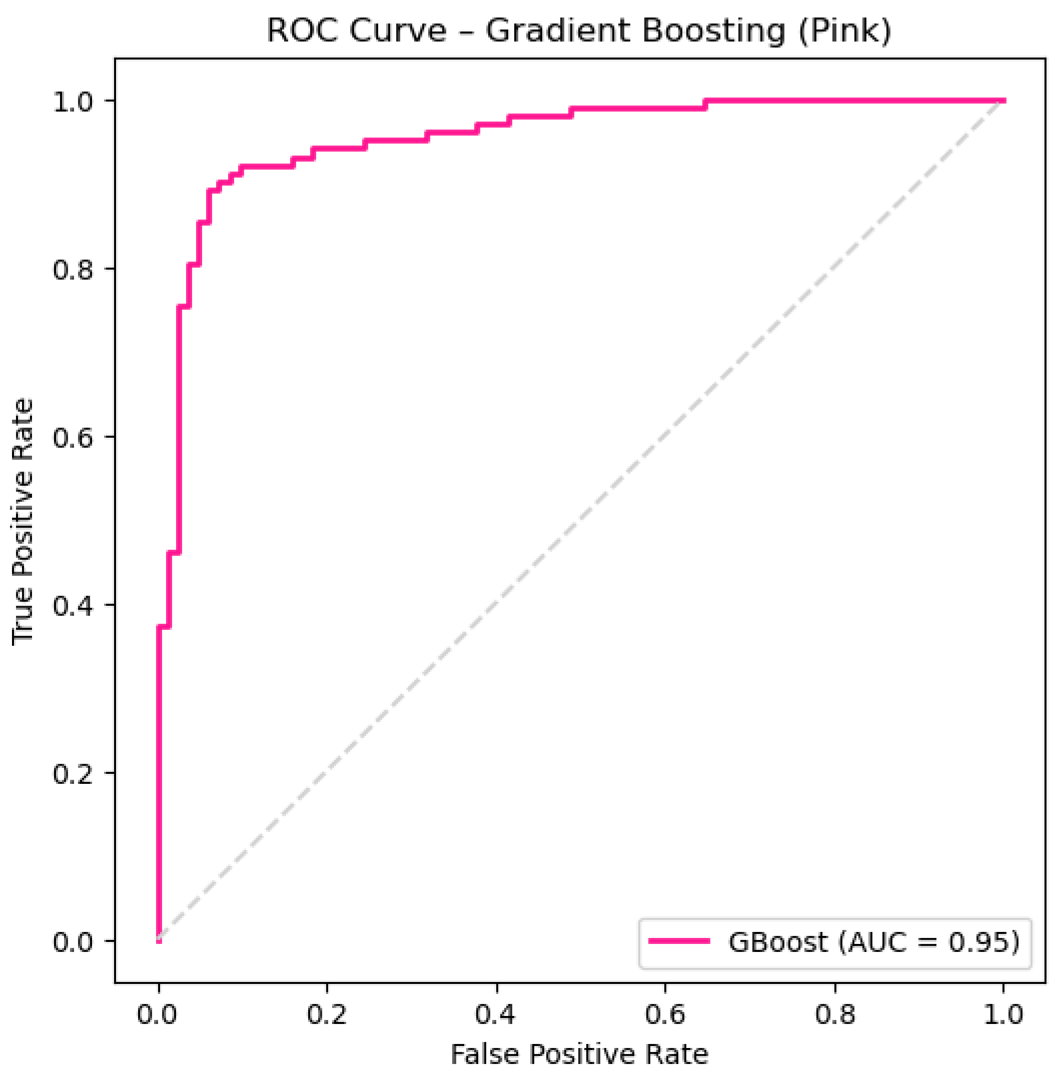
GRADIENT BOOSTING CLASSIFIER

In [103…

```
# 1) Train
gboost = GradientBoostingClassifier(
    random_state=42,      # reproducibility
    learning_rate=0.05,   # mild shrinkage
    n_estimators=300,     # a bit larger to stabilize
    max_depth=2           # shallow trees (prevents overfitting on small dat
)
gboost.fit(X_train, y_train)

# 2) Predict class labels and probabilities
y_pred_gb = gboost.predict(X_test)
y_proba_gb = gboost.predict_proba(X_test)[:, 1]

# 3) Metrics (keep them all!)
gboostAcc = accuracy_score(y_test, y_pred_gb)
gboostAUC = roc_auc_score(y_test, y_proba_gb)

print("Gradient Boosting Accuracy:", gboostAcc)
print("Gradient Boosting ROC AUC:", gboostAUC)
print("\nClassification Report (GBoost):\n", classification_report(y_test, y
print("Confusion Matrix (GBoost):\n", confusion_matrix(y_test, y_pred_gb))

# 4) ROC curve
fpr, tpr, _ = roc_curve(y_test, y_proba_gb)
plt.figure(figsize=(6,6))
plt.plot(fpr, tpr, lw=2, color="deeppink",
         label=f"GBoost (AUC = {gboostAUC:.2f})")
plt.plot([0,1], [0,1], '--', color='lightgray')
plt.xlabel("False Positive Rate")
```

```
plt.ylabel("True Positive Rate")
plt.title("ROC Curve – Gradient Boosting (Pink)")
plt.legend(loc="lower right")
plt.show()
```

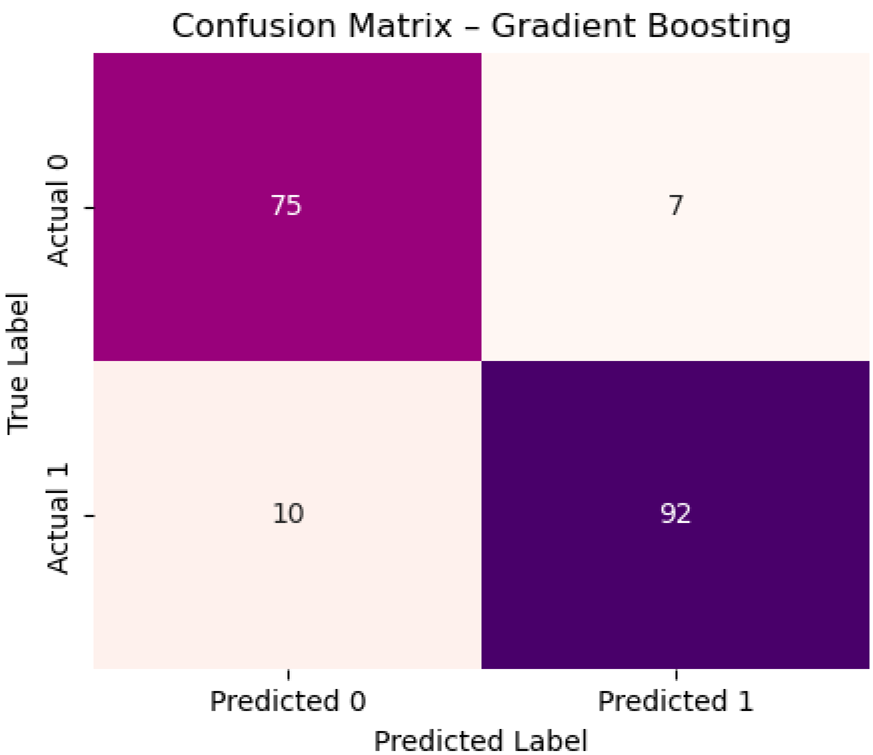
```
Gradient Boosting Accuracy: 0.907608695652174
Gradient Boosting ROC AUC: 0.9549258727881397

Classification Report (GBoost):
              precision    recall  f1-score   support

           0       0.88      0.91      0.90        82
           1       0.93      0.90      0.92       102

    accuracy                           0.91       184
   macro avg       0.91      0.91      0.91       184
weighted avg       0.91      0.91      0.91       184

Confusion Matrix (GBoost):
 [[75  7]
 [10 92]]
```



In [105…

```
# --- Confusion Matrix Visualization for GBoost (Pink tones) ---
cm = confusion_matrix(y_test, y_pred_gb)
```

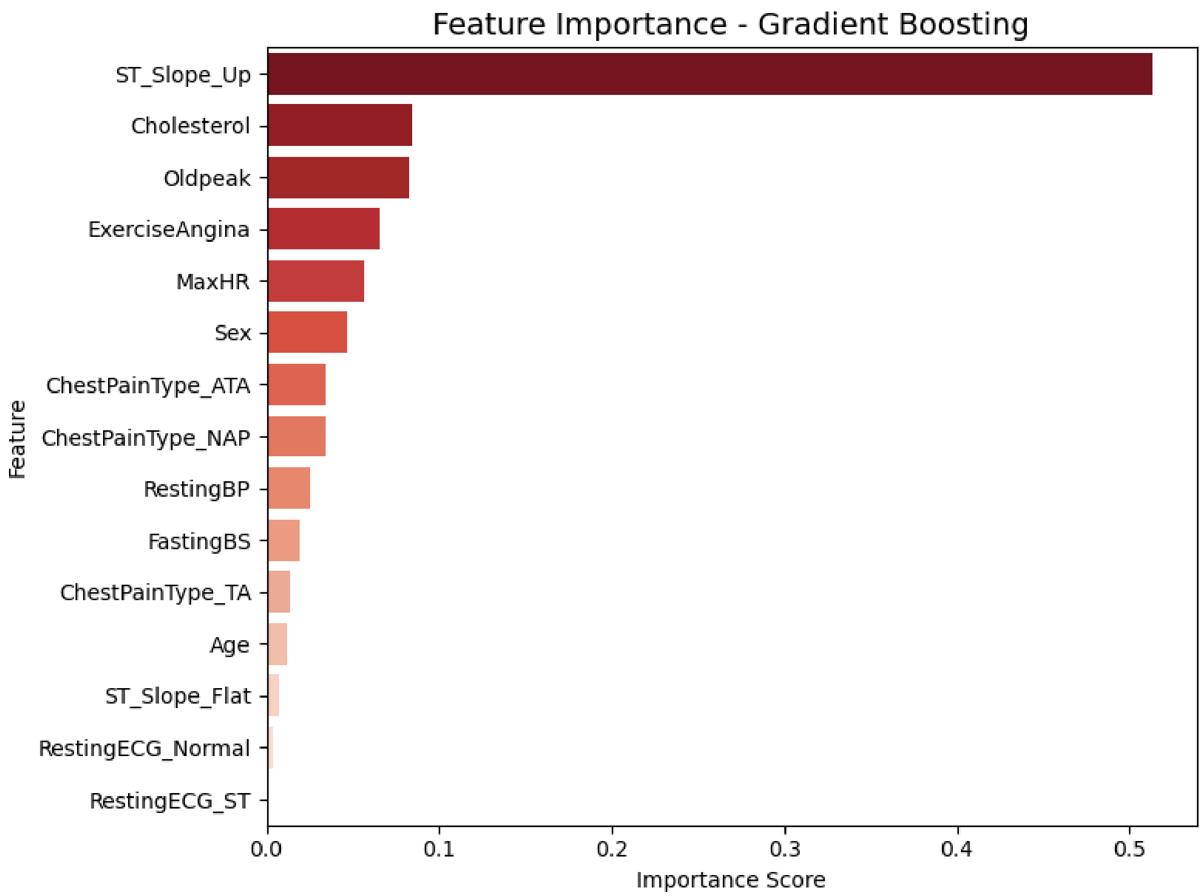
```
plt.figure(figsize=(5,4))
sns.heatmap(cm, annot=True, fmt="d", cmap="RdPu", cbar=False,
            xticklabels=["Predicted 0", "Predicted 1"],
            yticklabels=["Actual 0", "Actual 1"])
plt.title("Confusion Matrix – Gradient Boosting")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()
```



In [107…

```
# 1) Get feature importances from trained Gradient Boosting model
importances = gboost.feature_importances_

# 2) Put into a DataFrame with feature names
feat_imp = pd.DataFrame({
    "Feature": X_train.columns,
    "Importance": importances
}).sort_values(by="Importance", ascending=False)

print(feat_imp)

# 3) Plot feature importance (bar chart)
plt.figure(figsize=(8,6))
sns.barplot(x="Importance", y="Feature", data=feat_imp, palette="Reds_r")
plt.title("Feature Importance - Gradient Boosting", fontsize=14)
plt.xlabel("Importance Score")
plt.ylabel("Feature")
plt.tight_layout()
plt.show()
```

	Feature	Importance
14	ST_Slope_Up	0.513421
3	Cholesterol	0.084141
7	Oldpeak	0.082765
6	ExerciseAngina	0.065509
5	MaxHR	0.056675
1	Sex	0.047185
8	ChestPainType_ATA	0.034654
9	ChestPainType_NAP	0.034181
2	RestingBP	0.025429
4	FastingBS	0.018967
10	ChestPainType_TA	0.013954
0	Age	0.012171
13	ST_Slope_Flat	0.007031
11	RestingECG_Normal	0.003777
12	RestingECG_ST	0.000138



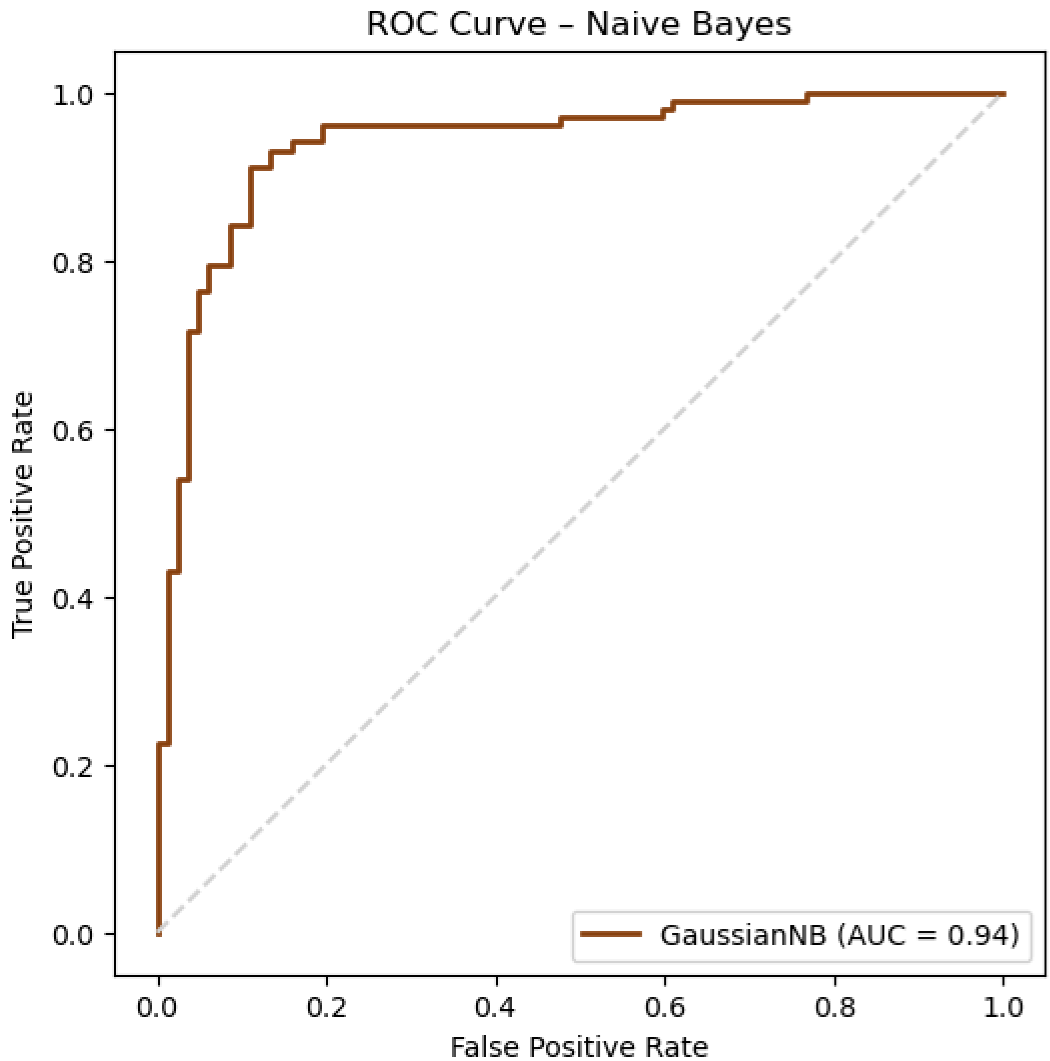
NAIVE BAYES (GAUSSIAN)

In [110…

```
# 1) Train
gnb = GaussianNB()
gnb.fit(X_train, y_train)

# 2) Predict
y_pred_nb = gnb.predict(X_test)
y_proba_nb = gnb.predict_proba(X_test)[:, 1]

# 3) Metrics (keep report & metrics, drop confusion matrix numbers)
```

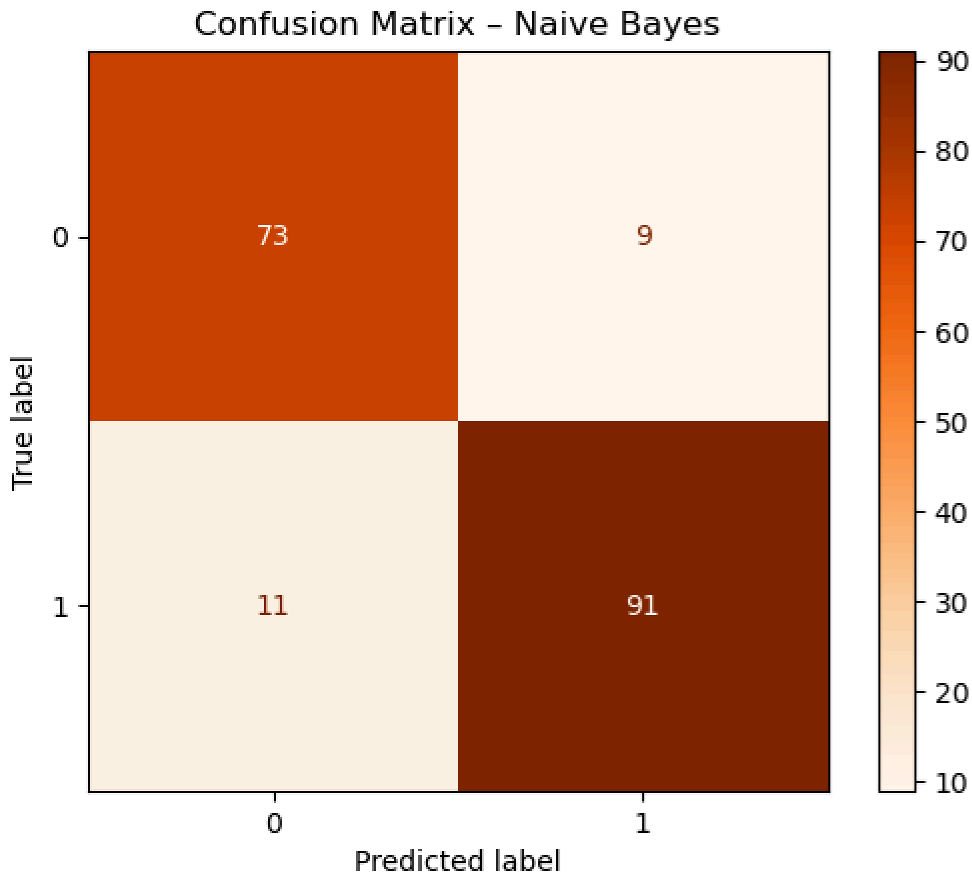
```python
nbAcc = accuracy_score(y_test, y_pred_nb)
nbAUC = roc_auc_score(y_test, y_proba_nb)

print("Naive Bayes (Gaussian) Accuracy:", nbAcc)
print("Naive Bayes (Gaussian) ROC AUC:", nbAUC)
print("\nClassification Report (GaussianNB):\n", classification_report(y_tes

# 4) ROC curve (brown tones)
fpr, tpr, _ = roc_curve(y_test, y_proba_nb)
plt.figure(figsize=(6,6))
plt.plot(fpr, tpr, lw=2, color="saddlebrown",
         label=f"GaussianNB (AUC = {nbAUC:.2f})")
plt.plot([0,1], [0,1], '--', color='lightgray')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve — Naive Bayes")
plt.legend(loc="lower right")
plt.show()
```

```
Naive Bayes (Gaussian) Accuracy: 0.8913043478260869
Naive Bayes (Gaussian) ROC AUC: 0.9404591104734578

Classification Report (GaussianNB):
               precision    recall  f1-score   support

           0       0.87      0.89      0.88        82
           1       0.91      0.89      0.90       102

    accuracy                           0.89       184
   macro avg       0.89      0.89      0.89       184
weighted avg       0.89      0.89      0.89       184
```



In [112…

```
# Confusion Matrix
cm = confusion_matrix(y_test, y_pred_nb)
disp = ConfusionMatrixDisplay(confusion_matrix=cm)
disp.plot(cmap="Oranges")
plt.title("Confusion Matrix – Naive Bayes")
plt.show()
```

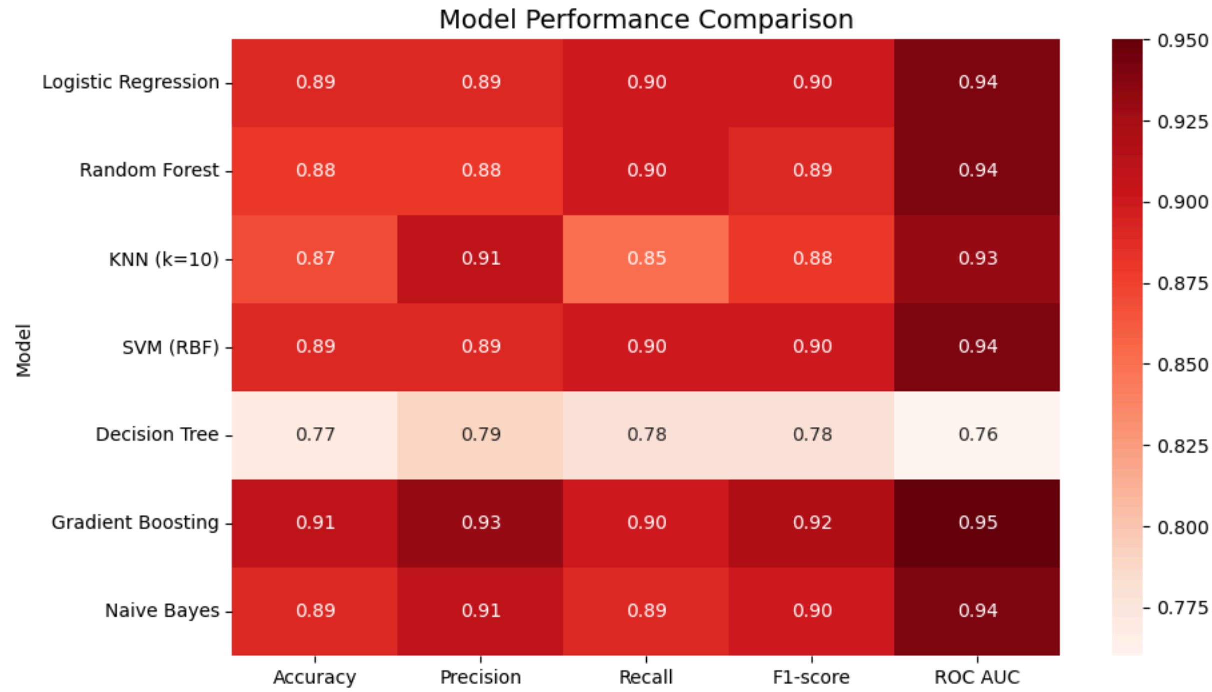


In [114…

```
# Step 1: Create a dictionary with model performance results
data = {
    "Model": [
        "Logistic Regression",
        "Random Forest",
        "KNN (k=10)",
        "SVM (RBF)",
        "Decision Tree",
        "Gradient Boosting",
        "Naive Bayes"
    ],
    "Accuracy": [0.89, 0.88, 0.87, 0.89, 0.77, 0.91, 0.89],
    "Precision": [0.89, 0.88, 0.91, 0.89, 0.79, 0.93, 0.91],
    "Recall": [0.90, 0.90, 0.85, 0.90, 0.78, 0.90, 0.89],
    "F1-score": [0.90, 0.89, 0.88, 0.90, 0.78, 0.92, 0.90],
    "ROC AUC": [0.94, 0.94, 0.93, 0.94, 0.76, 0.95, 0.94]
}

# Step 2: Convert to DataFrame
df_perf = pd.DataFrame(data)

# Step 3: Print table
print(df_perf)

# Step 4: Heatmap visualization
plt.figure(figsize=(10, 6))
sns.heatmap(df_perf.set_index("Model"), annot=True, cmap="Reds", fmt=".2f",
plt.title("Model Performance Comparison", fontsize=14)
```

```
plt.yticks(rotation=0)
plt.show()
```

```
                  Model  Accuracy  Precision  Recall  F1-score  ROC AUC
0   Logistic Regression      0.89       0.89    0.90      0.90     0.94
1         Random Forest      0.88       0.88    0.90      0.89     0.94
2            KNN (k=10)      0.87       0.91    0.85      0.88     0.93
3             SVM (RBF)      0.89       0.89    0.90      0.90     0.94
4         Decision Tree      0.77       0.79    0.78      0.78     0.76
5     Gradient Boosting      0.91       0.93    0.90      0.92     0.95
6           Naive Bayes      0.89       0.91    0.89      0.90     0.94
```



Deep Learning

BASELINE MLP

In [118…

```
# (only Dense layers, no regularization or callbacks)

# (Optional) set seeds for reproducibility
np.random.seed(42)
tf.random.set_seed(42)

# 1) Model architecture (very simple)
model_baseline = tf.keras.Sequential([
    tf.keras.layers.Dense(32, activation="relu", input_shape=(X_train.shape[
    tf.keras.layers.Dense(16, activation="relu"),
    tf.keras.layers.Dense(1, activation="sigmoid")   # output layer for bina
])

# 2) Compile the model
#    - Optimizer: Adam
#    - Loss: Binary crossentropy (since target is 0/1)
#    - Metrics: Accuracy and AUC
```

```
model_baseline.compile(
    optimizer="adam",
    loss="binary_crossentropy",
    metrics=["accuracy", tf.keras.metrics.AUC(name="auc")]
)

# 3) Training
#    - Uses 20% of the training set for validation (validation_split=0.2)
#    - Trains for 100 epochs with batch size = 32
#    - No early stopping or learning rate scheduling
history_base = model_baseline.fit(
    X_train, y_train,
    epochs=100,
    batch_size=32,
    validation_split=0.2,
    verbose=1
)

# 4) Evaluation on the test set
#    - Predictions are probabilities, converted to class labels at 0.5 thres
proba_base = model_baseline.predict(X_test).ravel()
y_pred_base = (proba_base >= 0.5).astype(int)

print("Test Accuracy (Baseline MLP):", accuracy_score(y_test, y_pred_base))
print("Test AUC (Baseline MLP):", roc_auc_score(y_test, proba_base))
print("\nClassification Report (Baseline MLP):\n", classification_report(y_t
print("Confusion Matrix (Baseline MLP):\n", confusion_matrix(y_test, y_pred_
```

```
Epoch 1/100
```

```
2025-08-29 17:36:05.921806: I metal_plugin/src/device/metal_device.cc:1154]
Metal device set to: Apple M3 Pro
2025-08-29 17:36:05.921903: I metal_plugin/src/device/metal_device.cc:296] s
ystemMemory: 18.00 GB
2025-08-29 17:36:05.921924: I metal_plugin/src/device/metal_device.cc:313] m
axCacheSize: 6.00 GB
2025-08-29 17:36:05.921975: I tensorflow/core/common_runtime/pluggable_devic
e/pluggable_device_factory.cc:305] Could not identify NUMA node of platform
GPU ID 0, defaulting to 0. Your kernel may not have been built with NUMA sup
port.
2025-08-29 17:36:05.922011: I tensorflow/core/common_runtime/pluggable_devic
e/pluggable_device_factory.cc:271] Created TensorFlow device (/job:localhos
t/replica:0/task:0/device:GPU:0 with 0 MB memory) -> physical PluggableDevic
e (device: 0, name: METAL, pci bus id: <undefined>)
2025-08-29 17:36:06.229703: I tensorflow/core/grappler/optimizers/custom_gra
ph_optimizer_registry.cc:117] Plugin optimizer for device_type GPU is enable
d.
2025-08-29 17:36:06.230842: E tensorflow/core/grappler/optimizers/meta_optim
izer.cc:961] PluggableGraphOptimizer failed: INVALID_ARGUMENT: Failed to des
erialize the `graph_buf`.
```

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 41ms/step - accuracy: 0.5853 - auc: 0.7332 - l
oss: 0.6408 - val_accuracy: 0.6871 - val_auc: 0.8036 - val_loss: 0.6008
Epoch 2/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.7474 - auc: 0.8732 - l
oss: 0.5550 - val_accuracy: 0.7551 - val_auc: 0.8416 - val_loss: 0.5469
Epoch 3/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8242 - auc: 0.8971 - l
oss: 0.4898 - val_accuracy: 0.7755 - val_auc: 0.8423 - val_loss: 0.5074
Epoch 4/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8464 - auc: 0.9060 - l
oss: 0.4382 - val_accuracy: 0.8027 - val_auc: 0.8471 - val_loss: 0.4823
Epoch 5/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8567 - auc: 0.9121 - l
oss: 0.4011 - val_accuracy: 0.8095 - val_auc: 0.8495 - val_loss: 0.4678
Epoch 6/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8618 - auc: 0.9171 - l
oss: 0.3761 - val_accuracy: 0.8095 - val_auc: 0.8537 - val_loss: 0.4601
Epoch 7/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8635 - auc: 0.9216 - l
oss: 0.3592 - val_accuracy: 0.8027 - val_auc: 0.8557 - val_loss: 0.4561
Epoch 8/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8635 - auc: 0.9264 - l
oss: 0.3465 - val_accuracy: 0.8027 - val_auc: 0.8575 - val_loss: 0.4541
Epoch 9/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8720 - auc: 0.9300 - l
oss: 0.3362 - val_accuracy: 0.8095 - val_auc: 0.8603 - val_loss: 0.4528
Epoch 10/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8720 - auc: 0.9334 - l
oss: 0.3278 - val_accuracy: 0.8095 - val_auc: 0.8611 - val_loss: 0.4520
Epoch 11/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8720 - auc: 0.9365 - l
oss: 0.3204 - val_accuracy: 0.8095 - val_auc: 0.8605 - val_loss: 0.4515
Epoch 12/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8754 - auc: 0.9392 - l
oss: 0.3136 - val_accuracy: 0.8027 - val_auc: 0.8620 - val_loss: 0.4518
Epoch 13/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8754 - auc: 0.9417 - l
oss: 0.3075 - val_accuracy: 0.8027 - val_auc: 0.8616 - val_loss: 0.4527
Epoch 14/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8771 - auc: 0.9436 - l
oss: 0.3019 - val_accuracy: 0.8095 - val_auc: 0.8620 - val_loss: 0.4538
Epoch 15/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8805 - auc: 0.9457 - l
oss: 0.2968 - val_accuracy: 0.8163 - val_auc: 0.8624 - val_loss: 0.4550
Epoch 16/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8823 - auc: 0.9474 - l
oss: 0.2922 - val_accuracy: 0.8231 - val_auc: 0.8618 - val_loss: 0.4560
Epoch 17/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8840 - auc: 0.9491 - l
oss: 0.2876 - val_accuracy: 0.8299 - val_auc: 0.8624 - val_loss: 0.4570
Epoch 18/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8857 - auc: 0.9509 - l
oss: 0.2834 - val_accuracy: 0.8299 - val_auc: 0.8630 - val_loss: 0.4580
Epoch 19/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8891 - auc: 0.9521 - l
oss: 0.2795 - val_accuracy: 0.8299 - val_auc: 0.8626 - val_loss: 0.4593
```

```
Epoch 20/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8874 - auc: 0.9532 - l
oss: 0.2760 - val_accuracy: 0.8299 - val_auc: 0.8635 - val_loss: 0.4602
Epoch 21/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8908 - auc: 0.9544 - l
oss: 0.2727 - val_accuracy: 0.8299 - val_auc: 0.8642 - val_loss: 0.4614
Epoch 22/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8908 - auc: 0.9557 - l
oss: 0.2696 - val_accuracy: 0.8231 - val_auc: 0.8655 - val_loss: 0.4628
Epoch 23/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8959 - auc: 0.9568 - l
oss: 0.2667 - val_accuracy: 0.8231 - val_auc: 0.8657 - val_loss: 0.4639
Epoch 24/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8959 - auc: 0.9577 - l
oss: 0.2639 - val_accuracy: 0.8231 - val_auc: 0.8654 - val_loss: 0.4654
Epoch 25/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8976 - auc: 0.9584 - l
oss: 0.2614 - val_accuracy: 0.8299 - val_auc: 0.8662 - val_loss: 0.4672
Epoch 26/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8976 - auc: 0.9592 - l
oss: 0.2587 - val_accuracy: 0.8299 - val_auc: 0.8670 - val_loss: 0.4689
Epoch 27/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8959 - auc: 0.9601 - l
oss: 0.2564 - val_accuracy: 0.8299 - val_auc: 0.8675 - val_loss: 0.4703
Epoch 28/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8942 - auc: 0.9607 - l
oss: 0.2540 - val_accuracy: 0.8299 - val_auc: 0.8661 - val_loss: 0.4725
Epoch 29/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8942 - auc: 0.9612 - l
oss: 0.2520 - val_accuracy: 0.8163 - val_auc: 0.8655 - val_loss: 0.4740
Epoch 30/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8959 - auc: 0.9620 - l
oss: 0.2497 - val_accuracy: 0.8163 - val_auc: 0.8656 - val_loss: 0.4753
Epoch 31/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8959 - auc: 0.9626 - l
oss: 0.2477 - val_accuracy: 0.8095 - val_auc: 0.8663 - val_loss: 0.4775
Epoch 32/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8976 - auc: 0.9630 - l
oss: 0.2459 - val_accuracy: 0.8095 - val_auc: 0.8664 - val_loss: 0.4785
Epoch 33/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.8976 - auc: 0.9636 - l
oss: 0.2438 - val_accuracy: 0.8095 - val_auc: 0.8667 - val_loss: 0.4801
Epoch 34/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.8976 - auc: 0.9642 - l
oss: 0.2420 - val_accuracy: 0.8095 - val_auc: 0.8655 - val_loss: 0.4811
Epoch 35/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.8993 - auc: 0.9649 - l
oss: 0.2400 - val_accuracy: 0.8095 - val_auc: 0.8661 - val_loss: 0.4827
Epoch 36/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9010 - auc: 0.9655 - l
oss: 0.2383 - val_accuracy: 0.8163 - val_auc: 0.8670 - val_loss: 0.4836
Epoch 37/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.8993 - auc: 0.9661 - l
oss: 0.2364 - val_accuracy: 0.8163 - val_auc: 0.8663 - val_loss: 0.4847
Epoch 38/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.8993 - auc: 0.9667 - l
```

```
oss: 0.2345 - val_accuracy: 0.8163 - val_auc: 0.8666 - val_loss: 0.4861
Epoch 39/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9027 - auc: 0.9671 - l
oss: 0.2329 - val_accuracy: 0.8027 - val_auc: 0.8660 - val_loss: 0.4865
Epoch 40/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9044 - auc: 0.9678 - l
oss: 0.2308 - val_accuracy: 0.8027 - val_auc: 0.8662 - val_loss: 0.4880
Epoch 41/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9044 - auc: 0.9682 - l
oss: 0.2294 - val_accuracy: 0.8027 - val_auc: 0.8670 - val_loss: 0.4890
Epoch 42/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9027 - auc: 0.9686 - l
oss: 0.2276 - val_accuracy: 0.8027 - val_auc: 0.8666 - val_loss: 0.4905
Epoch 43/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9044 - auc: 0.9690 - l
oss: 0.2263 - val_accuracy: 0.8027 - val_auc: 0.8670 - val_loss: 0.4916
Epoch 44/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9061 - auc: 0.9695 - l
oss: 0.2246 - val_accuracy: 0.8027 - val_auc: 0.8649 - val_loss: 0.4926
Epoch 45/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9078 - auc: 0.9699 - l
oss: 0.2232 - val_accuracy: 0.8027 - val_auc: 0.8658 - val_loss: 0.4939
Epoch 46/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9078 - auc: 0.9704 - l
oss: 0.2216 - val_accuracy: 0.8027 - val_auc: 0.8661 - val_loss: 0.4956
Epoch 47/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9078 - auc: 0.9705 - l
oss: 0.2204 - val_accuracy: 0.8027 - val_auc: 0.8665 - val_loss: 0.4959
Epoch 48/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9078 - auc: 0.9710 - l
oss: 0.2188 - val_accuracy: 0.8027 - val_auc: 0.8672 - val_loss: 0.4974
Epoch 49/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9130 - auc: 0.9713 - l
oss: 0.2175 - val_accuracy: 0.7891 - val_auc: 0.8667 - val_loss: 0.4983
Epoch 50/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9113 - auc: 0.9717 - l
oss: 0.2161 - val_accuracy: 0.7891 - val_auc: 0.8674 - val_loss: 0.4996
Epoch 51/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9147 - auc: 0.9722 - l
oss: 0.2147 - val_accuracy: 0.7891 - val_auc: 0.8674 - val_loss: 0.5011
Epoch 52/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9147 - auc: 0.9725 - l
oss: 0.2136 - val_accuracy: 0.7891 - val_auc: 0.8680 - val_loss: 0.5021
Epoch 53/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9164 - auc: 0.9729 - l
oss: 0.2122 - val_accuracy: 0.7891 - val_auc: 0.8689 - val_loss: 0.5030
Epoch 54/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9164 - auc: 0.9731 - l
oss: 0.2110 - val_accuracy: 0.7891 - val_auc: 0.8688 - val_loss: 0.5042
Epoch 55/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9181 - auc: 0.9733 - l
oss: 0.2097 - val_accuracy: 0.7891 - val_auc: 0.8694 - val_loss: 0.5048
Epoch 56/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9198 - auc: 0.9736 - l
oss: 0.2085 - val_accuracy: 0.7891 - val_auc: 0.8697 - val_loss: 0.5054
Epoch 57/100
```

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9198 - auc: 0.9738 - l
oss: 0.2074 - val_accuracy: 0.7891 - val_auc: 0.8695 - val_loss: 0.5065
Epoch 58/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9198 - auc: 0.9742 - l
oss: 0.2059 - val_accuracy: 0.7891 - val_auc: 0.8684 - val_loss: 0.5068
Epoch 59/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9215 - auc: 0.9745 - l
oss: 0.2050 - val_accuracy: 0.7891 - val_auc: 0.8694 - val_loss: 0.5076
Epoch 60/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9198 - auc: 0.9748 - l
oss: 0.2036 - val_accuracy: 0.7891 - val_auc: 0.8695 - val_loss: 0.5089
Epoch 61/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9215 - auc: 0.9751 - l
oss: 0.2025 - val_accuracy: 0.7891 - val_auc: 0.8694 - val_loss: 0.5086
Epoch 62/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9232 - auc: 0.9753 - l
oss: 0.2013 - val_accuracy: 0.7891 - val_auc: 0.8677 - val_loss: 0.5104
Epoch 63/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9249 - auc: 0.9755 - l
oss: 0.2001 - val_accuracy: 0.7891 - val_auc: 0.8674 - val_loss: 0.5100
Epoch 64/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9266 - auc: 0.9757 - l
oss: 0.1990 - val_accuracy: 0.7891 - val_auc: 0.8675 - val_loss: 0.5118
Epoch 65/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9283 - auc: 0.9760 - l
oss: 0.1979 - val_accuracy: 0.7891 - val_auc: 0.8675 - val_loss: 0.5110
Epoch 66/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9266 - auc: 0.9767 - l
oss: 0.1966 - val_accuracy: 0.7959 - val_auc: 0.8676 - val_loss: 0.5125
Epoch 67/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9283 - auc: 0.9769 - l
oss: 0.1955 - val_accuracy: 0.7959 - val_auc: 0.8678 - val_loss: 0.5126
Epoch 68/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9266 - auc: 0.9772 - l
oss: 0.1943 - val_accuracy: 0.7959 - val_auc: 0.8680 - val_loss: 0.5132
Epoch 69/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9283 - auc: 0.9775 - l
oss: 0.1932 - val_accuracy: 0.7959 - val_auc: 0.8684 - val_loss: 0.5140
Epoch 70/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9283 - auc: 0.9778 - l
oss: 0.1922 - val_accuracy: 0.7959 - val_auc: 0.8683 - val_loss: 0.5141
Epoch 71/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9283 - auc: 0.9780 - l
oss: 0.1911 - val_accuracy: 0.7959 - val_auc: 0.8684 - val_loss: 0.5160
Epoch 72/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9300 - auc: 0.9783 - l
oss: 0.1899 - val_accuracy: 0.7959 - val_auc: 0.8689 - val_loss: 0.5162
Epoch 73/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9300 - auc: 0.9786 - l
oss: 0.1890 - val_accuracy: 0.7891 - val_auc: 0.8673 - val_loss: 0.5176
Epoch 74/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9300 - auc: 0.9788 - l
oss: 0.1878 - val_accuracy: 0.7891 - val_auc: 0.8679 - val_loss: 0.5181
Epoch 75/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9300 - auc: 0.9791 - l
oss: 0.1869 - val_accuracy: 0.7891 - val_auc: 0.8675 - val_loss: 0.5187
```

```
Epoch 76/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9300 - auc: 0.9793 - l
oss: 0.1857 - val_accuracy: 0.7891 - val_auc: 0.8676 - val_loss: 0.5199
Epoch 77/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9300 - auc: 0.9795 - l
oss: 0.1847 - val_accuracy: 0.7891 - val_auc: 0.8687 - val_loss: 0.5211
Epoch 78/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9300 - auc: 0.9797 - l
oss: 0.1836 - val_accuracy: 0.7891 - val_auc: 0.8690 - val_loss: 0.5219
Epoch 79/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9300 - auc: 0.9799 - l
oss: 0.1826 - val_accuracy: 0.7891 - val_auc: 0.8687 - val_loss: 0.5223
Epoch 80/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9300 - auc: 0.9803 - l
oss: 0.1813 - val_accuracy: 0.7891 - val_auc: 0.8680 - val_loss: 0.5244
Epoch 81/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.9283 - auc: 0.9805 - lo
ss: 0.1806 - val_accuracy: 0.7891 - val_auc: 0.8680 - val_loss: 0.5245
Epoch 82/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9300 - auc: 0.9810 - l
oss: 0.1792 - val_accuracy: 0.7891 - val_auc: 0.8685 - val_loss: 0.5261
Epoch 83/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9283 - auc: 0.9811 - l
oss: 0.1782 - val_accuracy: 0.7891 - val_auc: 0.8673 - val_loss: 0.5270
Epoch 84/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9317 - auc: 0.9813 - l
oss: 0.1770 - val_accuracy: 0.7891 - val_auc: 0.8671 - val_loss: 0.5287
Epoch 85/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9317 - auc: 0.9815 - l
oss: 0.1761 - val_accuracy: 0.7891 - val_auc: 0.8674 - val_loss: 0.5293
Epoch 86/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.9317 - auc: 0.9818 - lo
ss: 0.1747 - val_accuracy: 0.7891 - val_auc: 0.8673 - val_loss: 0.5311
Epoch 87/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9300 - auc: 0.9821 - l
oss: 0.1738 - val_accuracy: 0.7891 - val_auc: 0.8673 - val_loss: 0.5314
Epoch 88/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9334 - auc: 0.9824 - l
oss: 0.1724 - val_accuracy: 0.7823 - val_auc: 0.8664 - val_loss: 0.5338
Epoch 89/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9352 - auc: 0.9827 - l
oss: 0.1713 - val_accuracy: 0.7891 - val_auc: 0.8648 - val_loss: 0.5348
Epoch 90/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9352 - auc: 0.9828 - l
oss: 0.1701 - val_accuracy: 0.7823 - val_auc: 0.8656 - val_loss: 0.5362
Epoch 91/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9352 - auc: 0.9831 - l
oss: 0.1691 - val_accuracy: 0.7823 - val_auc: 0.8652 - val_loss: 0.5370
Epoch 92/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9369 - auc: 0.9834 - l
oss: 0.1676 - val_accuracy: 0.7823 - val_auc: 0.8648 - val_loss: 0.5388
Epoch 93/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9369 - auc: 0.9837 - l
oss: 0.1667 - val_accuracy: 0.7823 - val_auc: 0.8644 - val_loss: 0.5403
Epoch 94/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9403 - auc: 0.9839 - l
```

```
oss: 0.1655 - val_accuracy: 0.7823 - val_auc: 0.8628 - val_loss: 0.5414
Epoch 95/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9403 - auc: 0.9842 - l
oss: 0.1642 - val_accuracy: 0.7823 - val_auc: 0.8633 - val_loss: 0.5424
Epoch 96/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9386 - auc: 0.9844 - l
oss: 0.1633 - val_accuracy: 0.7823 - val_auc: 0.8630 - val_loss: 0.5428
Epoch 97/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9403 - auc: 0.9848 - l
oss: 0.1619 - val_accuracy: 0.7823 - val_auc: 0.8630 - val_loss: 0.5452
Epoch 98/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9403 - auc: 0.9850 - l
oss: 0.1608 - val_accuracy: 0.7823 - val_auc: 0.8628 - val_loss: 0.5460
Epoch 99/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9403 - auc: 0.9853 - l
oss: 0.1597 - val_accuracy: 0.7823 - val_auc: 0.8629 - val_loss: 0.5470
Epoch 100/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.9403 - auc: 0.9854 - lo
ss: 0.1584 - val_accuracy: 0.7823 - val_auc: 0.8632 - val_loss: 0.5495
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step
Test Accuracy (Baseline MLP): 0.875
Test AUC (Baseline MLP): 0.9230033476805356

Classification Report (Baseline MLP):
               precision    recall  f1-score   support

           0       0.85      0.88      0.86        82
           1       0.90      0.87      0.89       102

    accuracy                           0.88       184
   macro avg       0.87      0.88      0.87       184
weighted avg       0.88      0.88      0.88       184

Confusion Matrix (Baseline MLP):
 [[72 10]
 [13 89]]
```

Enhanced MLP

In [120…

```
# (Optional) ensure reproducibility
np.random.seed(42)
tf.random.set_seed(42)

# 1) Define a simple MLP architecture for binary classification
#    - Hidden layers use ReLU to capture non-linearities
#    - BatchNorm + Dropout help stabilize training and reduce overfitting
#    - Output layer is a single neuron with Sigmoid to output P(y=1)
model = Sequential([
    Dense(64, activation="relu", input_shape=(X_train.shape[1],)),
    BatchNormalization(),
    Dropout(0.2),

    Dense(32, activation="relu"),
    BatchNormalization(),
    Dropout(0.2),
```

```
    Dense(16, activation="relu"),
    Dense(1, activation="sigmoid")  # binary classification output (probabil
])

# 2) Compile the model
#    - Binary cross-entropy is the standard loss for 0/1 targets
#    - Track both Accuracy and AUC (discrimination)
model.compile(
    optimizer="adam",
    loss="binary_crossentropy",
    metrics=["accuracy", tf.keras.metrics.AUC(name="auc")]
)

# 3) Callbacks to improve training stability
#    - EarlyStopping: stop when val_loss stops improving; restore best weigh
#    - ReduceLROnPlateau: reduce learning rate when val_loss plateaus
early = EarlyStopping(monitor="val_loss", patience=10, restore_best_weights=
plateau = ReduceLROnPlateau(monitor="val_loss", factor=0.5, patience=5, min_

# 4) Train the model
#    - validation_split=0.2: uses 20% of the TRAINING set as validation
#      (test set remains untouched for final evaluation)
history = model.fit(
    X_train, y_train,
    epochs=100,
    batch_size=32,
    validation_split=0.2,
    callbacks=[early, plateau],
    verbose=1
)

# 5) Final evaluation on the held-out test set
#    - proba: predicted probability of the positive class (y=1)
#    - y_pred: hard labels using a default 0.50 threshold
proba = model.predict(X_test).ravel()
y_pred = (proba >= 0.5).astype(int)

print("Test Accuracy:", accuracy_score(y_test, y_pred))
print("Test AUC:", roc_auc_score(y_test, proba))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))

# (Optional) If you want to optimize the decision threshold instead of using
# you can compute the ROC curve and pick the Youden J point (argmax of TPR -
# Then recompute y_pred and the confusion matrix at that threshold.
```

```
Epoch 1/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 46ms/step - accuracy: 0.5512 - auc: 0.6266 - l
oss: 0.7772 - val_accuracy: 0.6599 - val_auc: 0.7215 - val_loss: 0.6653 - le
arning_rate: 0.0010
Epoch 2/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step - accuracy: 0.7014 - auc: 0.8002 - l
oss: 0.5732 - val_accuracy: 0.7211 - val_auc: 0.8085 - val_loss: 0.6049 - le
arning_rate: 0.0010
Epoch 3/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.8020 - auc: 0.8790 - l
oss: 0.4532 - val_accuracy: 0.7755 - val_auc: 0.8312 - val_loss: 0.5646 - le
arning_rate: 0.0010
Epoch 4/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.8225 - auc: 0.8936 - l
oss: 0.4173 - val_accuracy: 0.7959 - val_auc: 0.8528 - val_loss: 0.5278 - le
arning_rate: 0.0010
Epoch 5/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.8311 - auc: 0.8974 - l
oss: 0.4064 - val_accuracy: 0.8095 - val_auc: 0.8538 - val_loss: 0.5042 - le
arning_rate: 0.0010
Epoch 6/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.8345 - auc: 0.9137 - l
oss: 0.3667 - val_accuracy: 0.8095 - val_auc: 0.8570 - val_loss: 0.4880 - le
arning_rate: 0.0010
Epoch 7/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.8447 - auc: 0.9195 - l
oss: 0.3561 - val_accuracy: 0.8231 - val_auc: 0.8553 - val_loss: 0.4746 - le
arning_rate: 0.0010
Epoch 8/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.8737 - auc: 0.9324 - l
oss: 0.3298 - val_accuracy: 0.8299 - val_auc: 0.8565 - val_loss: 0.4642 - le
arning_rate: 0.0010
Epoch 9/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.8532 - auc: 0.9257 - l
oss: 0.3442 - val_accuracy: 0.8299 - val_auc: 0.8570 - val_loss: 0.4580 - le
arning_rate: 0.0010
Epoch 10/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.8601 - auc: 0.9412 - l
oss: 0.3087 - val_accuracy: 0.8367 - val_auc: 0.8604 - val_loss: 0.4537 - le
arning_rate: 0.0010
Epoch 11/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.8652 - auc: 0.9350 - l
oss: 0.3181 - val_accuracy: 0.8367 - val_auc: 0.8615 - val_loss: 0.4498 - le
arning_rate: 0.0010
Epoch 12/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.8652 - auc: 0.9486 - l
oss: 0.2906 - val_accuracy: 0.8435 - val_auc: 0.8615 - val_loss: 0.4499 - le
arning_rate: 0.0010
Epoch 13/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.8669 - auc: 0.9435 - l
oss: 0.2999 - val_accuracy: 0.8367 - val_auc: 0.8581 - val_loss: 0.4521 - le
arning_rate: 0.0010
Epoch 14/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step - accuracy: 0.8908 - auc: 0.9509 - l
oss: 0.2788 - val_accuracy: 0.8299 - val_auc: 0.8582 - val_loss: 0.4520 - le
arning_rate: 0.0010
```

```
Epoch 15/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.8840 - auc: 0.9496 - l
oss: 0.2866 - val_accuracy: 0.8231 - val_auc: 0.8585 - val_loss: 0.4519 - le
arning_rate: 0.0010
Epoch 16/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.8874 - auc: 0.9570 - l
oss: 0.2683 - val_accuracy: 0.8163 - val_auc: 0.8573 - val_loss: 0.4541 - le
arning_rate: 0.0010
Epoch 17/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.8908 - auc: 0.9560 - l
oss: 0.2662 - val_accuracy: 0.8163 - val_auc: 0.8563 - val_loss: 0.4553 - le
arning_rate: 5.0000e-04
Epoch 18/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.8891 - auc: 0.9572 - l
oss: 0.2628 - val_accuracy: 0.8163 - val_auc: 0.8554 - val_loss: 0.4575 - le
arning_rate: 5.0000e-04
Epoch 19/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.8976 - auc: 0.9473 - l
oss: 0.2857 - val_accuracy: 0.8027 - val_auc: 0.8583 - val_loss: 0.4600 - le
arning_rate: 5.0000e-04
Epoch 20/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.8788 - auc: 0.9567 - l
oss: 0.2637 - val_accuracy: 0.8095 - val_auc: 0.8556 - val_loss: 0.4633 - le
arning_rate: 5.0000e-04
Epoch 21/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.8976 - auc: 0.9623 - l
oss: 0.2463 - val_accuracy: 0.7959 - val_auc: 0.8578 - val_loss: 0.4649 - le
arning_rate: 5.0000e-04
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step
Test Accuracy: 0.8586956521739131
Test AUC: 0.9330463892874223

Classification Report:
              precision    recall  f1-score   support

           0       0.92      0.74      0.82        82
           1       0.82      0.95      0.88       102

    accuracy                           0.86       184
   macro avg       0.87      0.85      0.85       184
weighted avg       0.87      0.86      0.86       184

Confusion Matrix:
 [[61 21]
 [ 5 97]]
```

LSTM

In [142…

```python
# 1) Convert all features to numeric.
#    - If a value cannot be converted, it will be set to NaN.
X_num = X.apply(pd.to_numeric, errors='coerce')

# 2) Handle missing values (NaN).
#    - Here we replace them with 0, but in practice you could also use the m
X_num = X_num.fillna(0)
```

```
# 3) Standardize data types for Keras.
#    - Force all features to float32 (recommended format for neural networks
X_num = X_num.astype('float32')

# 4) Convert the target variable (y) to numeric as well.
#    - Any non-numeric values are coerced to NaN, then replaced with 0.
#    - Finally, cast to int32 (since this is a classification target).
y_num = pd.to_numeric(y, errors='coerce').fillna(0).astype('int32')

X_train = X_train.apply(pd.to_numeric, errors='coerce').fillna(0).astype('fl
X_test  = X_test.apply(pd.to_numeric, errors='coerce').fillna(0).astype('flo
y_train = pd.to_numeric(y_train, errors='coerce').fillna(0).astype('int32')
y_test  = pd.to_numeric(y_test,  errors='coerce').fillna(0).astype('int32')

print("Object dtype columns in X_train:",
      list(X_train.columns[X_train.dtypes == 'object']))
print("Any NaNs? ->", X_train.isna().any().any(), y_train.isna().any())

n_features = X_train.shape[1]
X_train_seq = np.asarray(X_train, dtype=np.float32).reshape(-1, n_features,
X_test_seq  = np.asarray(X_test,  dtype=np.float32).reshape(-1, n_features,
```

```
Object dtype columns in X_train: []
Any NaNs? -> False False
```

In [152…

```
model_lstm = Sequential([
    LSTM(32, input_shape=(X_train_seq.shape[1], X_train_seq.shape[2])),
    Dense(1, activation='sigmoid')
])
model_lstm.compile(optimizer='adam', loss='binary_crossentropy', metrics=['a

# 6) Eğit
hist_lstm = model_lstm.fit(
    X_train_seq, y_train,
    epochs=20, batch_size=32,
    validation_split=0.2, verbose=1
)

# 7) Tahmin ve metrikler
proba_lstm = model_lstm.predict(X_test_seq).ravel()
y_pred_lstm = (proba_lstm >= 0.5).astype(int)

# Accuracy ve AUC (bunlar zaten sende var)
print("Accuracy:", accuracy_score(y_test, y_pred_lstm))
print("AUC:", roc_auc_score(y_test, proba_lstm))

# Recall ve F1'i ek import olmadan hesapla (class=1)
cm = confusion_matrix(y_test, y_pred_lstm, labels=[0, 1])
tn, fp, fn, tp = cm.ravel()
precision = tp / (tp + fp) if (tp + fp) > 0 else 0.0
recall    = tp / (tp + fn) if (tp + fn) > 0 else 0.0
f1        = (2 * precision * recall / (precision + recall)) if (precision +

print("Recall (class=1):", round(recall, 4))
print("F1 (class=1):", round(f1, 4))
```

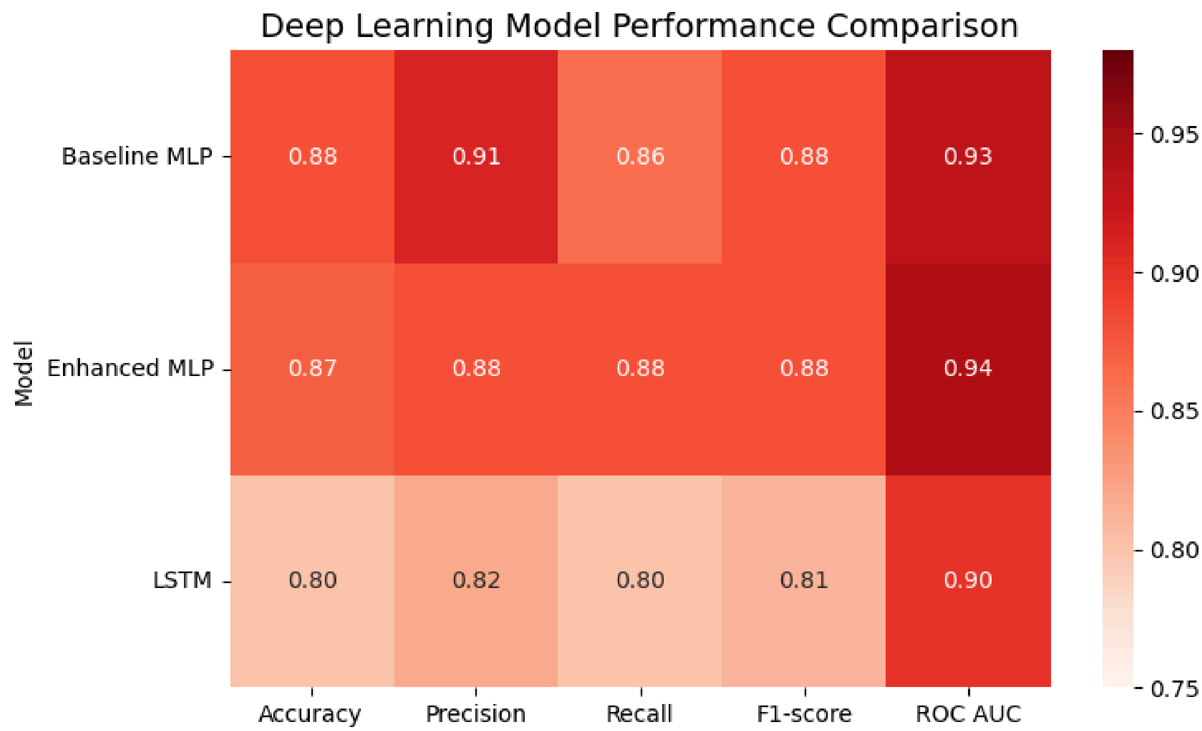
```
# Detaylı rapor ve matris (bunlar da sende var)
print("\nClassification Report:\n", classification_report(y_test, y_pred_lst
print("Confusion Matrix:\n", cm)
```

```
Epoch 1/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.5580 - loss: 0.6878 -
val_accuracy: 0.5714 - val_loss: 0.6789
Epoch 2/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.5580 - loss: 0.6695 - v
al_accuracy: 0.5782 - val_loss: 0.6606
Epoch 3/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.5956 - loss: 0.6558 - v
al_accuracy: 0.6122 - val_loss: 0.6503
Epoch 4/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.6246 - loss: 0.6434 - v
al_accuracy: 0.6599 - val_loss: 0.6427
Epoch 5/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.6314 - loss: 0.6317 - v
al_accuracy: 0.6395 - val_loss: 0.6400
Epoch 6/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.6365 - loss: 0.6253 - v
al_accuracy: 0.6395 - val_loss: 0.6356
Epoch 7/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.6451 - loss: 0.6176 - v
al_accuracy: 0.6667 - val_loss: 0.6280
Epoch 8/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.6519 - loss: 0.6068 - v
al_accuracy: 0.6667 - val_loss: 0.6158
Epoch 9/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.6621 - loss: 0.5901 -
val_accuracy: 0.6871 - val_loss: 0.5978
Epoch 10/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.7048 - loss: 0.5663 - v
al_accuracy: 0.7143 - val_loss: 0.5764
Epoch 11/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.7611 - loss: 0.5396 - v
al_accuracy: 0.7755 - val_loss: 0.5569
Epoch 12/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.8089 - loss: 0.5152 - v
al_accuracy: 0.7823 - val_loss: 0.5415
Epoch 13/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.8259 - loss: 0.4964 - v
al_accuracy: 0.7891 - val_loss: 0.5313
Epoch 14/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.8276 - loss: 0.4825 - v
al_accuracy: 0.7959 - val_loss: 0.5247
Epoch 15/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.8294 - loss: 0.4726 - v
al_accuracy: 0.7959 - val_loss: 0.5209
Epoch 16/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.8294 - loss: 0.4652 - v
al_accuracy: 0.7959 - val_loss: 0.5192
Epoch 17/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.8276 - loss: 0.4597 -
val_accuracy: 0.7959 - val_loss: 0.5187
Epoch 18/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.8259 - loss: 0.4555 - v
al_accuracy: 0.7959 - val_loss: 0.5187
Epoch 19/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.8276 - loss: 0.4517 - v
```

```
al_accuracy: 0.7959 - val_loss: 0.5187
Epoch 20/20
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.8294 - loss: 0.4480 - v
al_accuracy: 0.7959 - val_loss: 0.5187
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step
Accuracy: 0.7934782608695652
AUC: 0.8950263032042085
Recall (class=1): 0.8039
F1 (class=1): 0.8119

Classification Report:
              precision    recall  f1-score   support

           0       0.76      0.78      0.77        82
           1       0.82      0.80      0.81       102

    accuracy                           0.79       184
   macro avg       0.79      0.79      0.79       184
weighted avg       0.79      0.79      0.79       184

Confusion Matrix:
 [[64 18]
 [20 82]]
```

In [156…

```
# Deep Learning model performance results
data_dl = {
    "Model": ["Baseline MLP", "Enhanced MLP", "LSTM"],
    "Accuracy": [0.88, 0.87, 0.80],
    "Precision": [0.91, 0.88, 0.82],
    "Recall": [0.86, 0.88, 0.80],
    "F1-score": [0.88, 0.88, 0.81],
    "ROC AUC": [0.93, 0.94, 0.90]
}

# Convert to DataFrame
df_dl = pd.DataFrame(data_dl).set_index("Model")

# Plot heatmap (red tones)
plt.figure(figsize=(8, 5))
sns.heatmap(df_dl, annot=True, fmt=".2f", cmap="Reds", vmin=0.75, vmax=0.98,
plt.title("Deep Learning Model Performance Comparison", fontsize=14)
plt.yticks(rotation=0)
plt.show()
```



In [158…

```
# Deep Learning model performance results
data_dl = {
    "Model": [
        "Baseline MLP",
        "Enhanced MLP",
        "LSTM"
    ],
    "Accuracy": [0.88, 0.87, 0.80],
    "Precision": [0.91, 0.88, 0.82],
    "Recall": [0.86, 0.88, 0.80],
    "F1-score": [0.88, 0.88, 0.81],
    "ROC AUC": [0.93, 0.94, 0.90]
}

df_dl = pd.DataFrame(data_dl)
print(df_dl)
```

```
          Model  Accuracy  Precision  Recall  F1-score  ROC AUC
0  Baseline MLP      0.88       0.91    0.86      0.88     0.93
1  Enhanced MLP      0.87       0.88    0.88      0.88     0.94
2          LSTM      0.80       0.82    0.80      0.81     0.90
```

Thank you