# Group16_Project_Final_Report

May 3, 2025

[ ]:
```
#For Final Project with an objective to find a good model that is able to␣
↪predict which water pumps are functional and which are not.
```

[31]:
```
#All relevant packages are imported

import pandas as pd
import numpy as np
import seaborn as sns
import missingno as msno
from scipy import stats
import matplotlib.pyplot as plt
from sklearn.metrics import log_loss
from imblearn.over_sampling import SMOTE
from sklearn.feature_selection import RFE
from tensorflow.keras.regularizers import l2
from tensorflow.keras.models import Sequential
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from tensorflow.keras.layers import Dense, Dropout, Input
from sklearn.model_selection import cross_val_score, GridSearchCV
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.naive_bayes import GaussianNB, MultinomialNB, BernoulliNB
from sklearn.metrics import confusion_matrix, auc, roc_auc_score, roc_curve,␣
↪classification_report,log_loss,accuracy_score, ConfusionMatrixDisplay,␣
↪RocCurveDisplay, precision_score, recall_score, f1_score
from sklearn.feature_selection import SelectFromModel
from sklearn.impute import KNNImputer
from sklearn.ensemble import BaggingClassifier
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, callbacks, regularizers
from sklearn.metrics import log_loss
import os

# Set seed for reproducibility
```

```
SEED = 42
import os, random, numpy as np
random.seed(SEED)                    # Python's built-in random module
np.random.seed(SEED)                 # NumPy's random module
os.environ['PYTHONHASHSEED'] = str(SEED)  # Hash seed (affects dict ordering␣
 ↪etc. in rare cases)

df = pd.read_csv(r"C:\Users\HP\Desktop\2025SPRING\DSCI5240\PROJECT STATUS␣
 ↪REPORT\DSCI 5240 Project Data.csv")
print(df.head())

import warnings
warnings.filterwarnings("ignore", category=FutureWarning)
```

```
  Water Pump ID Water Source Type Water Quality  Distance to Nearest Town  \
0         WP001              Well         Clean                      44.0
1         WP002              Lake         Clean                      13.0
2         WP003              Lake         Clean                      27.0
3         WP004              Well         Clean                      14.0
4         WP005              Lake         Clean                      41.0

   Population Served  Installation Year      Funder Payment Type  \
0            13000.0             2006.0  World Bank         Free
1            13000.0             1990.0   Red Cross         Free
2            12000.0             1997.0       Oxfam  Pay per use
3             9000.0             1992.0       Oxfam  Pay per use
4            16000.0             2006.0         NaN  Pay per use

   Water Pump Age        Pump Type                           GPS Coordinates  \
0            18.0  Motorized Pump  (-20.599463060030295, 26.696000047794744)
1            34.0       Hand Pump   (-20.69129769992364, 23.313405231404484)
2            27.0       Hand Pump  (-19.830951420391948, 26.650358442338003)
3            32.0             NaN   (-22.335866062765565, 22.83485684389231)
4            18.0       Hand Pump  (-21.099305692773278, 24.799143614430015)

  Functioning Status
0        Functioning
1    Not Functioning
2    Not Functioning
3        Functioning
4        Functioning
```

[32]:
```
#Data visualization and exploratory analysis codes have been removed from this␣
 ↪python file so as to make it easier for model codes to run.
#Exploratory Data Analysis has already been submited and shared in Python code␣
 ↪file in Project Status Report.
```

```python
#This file contains only the relevant data cleaning codes precisely required␣
 ↪for running all algorithms sufficiently well.

# DATA CLEANING STARTS
```

[33]:
```python
# MISSING DATA IMPUTATION: Numerical Features

# Taking imputation action based on distribution
# Features with normal distribution - Fill missing values with mean and replace␣
 ↪negative values with mean for specified columns

# Fill missing values with mean and replace negative values with mean for␣
 ↪'Distance to Nearest Town'
mean_distance = df['Distance to Nearest Town'].mean()
df['Distance to Nearest Town'] = df['Distance to Nearest Town'].apply(lambda x:␣
 ↪mean_distance if x < 0 or pd.isna(x) else x)

# Fill missing values with mean for 'Population Served'
df['Population Served'] = df['Population Served'].fillna(df['Population␣
 ↪Served'].mean())

# Taking imputation action based on distribution
# Features without normal distribution
# Fill missing values with mode for specified columns
mode_value = df['Installation Year'].mode()[0]  # Get the mode (most frequent␣
 ↪value)
df['Installation Year'] = df['Installation Year'].fillna(mode_value)
mode_value = df['Water Pump Age'].mode()[0]  # Get the mode (most frequent␣
 ↪value)
df['Water Pump Age'] = df['Water Pump Age'].fillna(mode_value)

# Checking missing values again after application of imputation technique
missing_values_df = pd.DataFrame({'Feature': df.columns, 'Missing Values': df.
 ↪isnull().sum().values})
print(missing_values_df)
```

```
                    Feature  Missing Values
0             Water Pump ID             250
1         Water Source Type             250
2             Water Quality             250
3  Distance to Nearest Town               0
4         Population Served               0
5         Installation Year               0
6                    Funder             250
7              Payment Type             250
8            Water Pump Age               0
9                 Pump Type             250
```

```
10           GPS Coordinates             250
11        Functioning Status             250
```

[34]:
```python
# Missing Values Check

# Checking the count of number of rows with missing values in any column, after␣
 ↪applying imputation techniques on numerical/float data type features
missing_rows = df.isnull().any(axis=1).sum()
print(f"Number of rows with at least one missing value: {missing_rows}")
```

```
Number of rows with at least one missing value: 1667
```

[35]:
```python
# MISSING DATA IMPUTATION: Categorical Features

# Proportional imputation function
def proportional_imputation(df, categorical_cols):
    """
    Imputes missing values in categorical columns of a DataFrame,
    preserving the original proportions of each category within the column.

    Args:
        df (pd.DataFrame): The DataFrame to impute.
        categorical_cols (list): A list of column names that are categorical.

    Returns:
        pd.DataFrame: The DataFrame with missing values imputed.
    """
    for col in categorical_cols:
        # Calculate the existing value counts and their proportions
        value_counts = df[col].value_counts(normalize=True)

        # Identify the missing values in the column
        missing_mask = df[col].isnull()
        num_missing = missing_mask.sum()

        # If there are no missing values, skip to the next column
        if num_missing == 0:
            continue

        # Randomly choose values to fill the missing spots based on the␣
 ↪proportions
        imputed_values = np.random.choice(value_counts.index, size=num_missing,␣
 ↪p=value_counts.values)

        # Fill the missing values with the randomly chosen values
        df.loc[missing_mask, col] = imputed_values

    return df  # Return the modified DataFrame
```

```python
# List of categorical columns to apply proportional imputation
categorical_features = ['Water Source Type', 'Water Quality',
                        'Funder', 'Payment Type', 'Pump Type',
                        'Functioning Status']

# Apply proportional imputation directly to the original DataFrame
df = proportional_imputation(df, categorical_features)

# Check for missing values after imputation
print("Missing values after imputation:")
print(df.isnull().sum())  # Prints the number of missing values for each column
```

```
Missing values after imputation:
Water Pump ID              250
Water Source Type            0
Water Quality                0
Distance to Nearest Town     0
Population Served            0
Installation Year            0
Funder                       0
Payment Type                 0
Water Pump Age               0
Pump Type                    0
GPS Coordinates            250
Functioning Status           0
dtype: int64
```

[36]:
```python
# Missing Values Check again

missing_values_df = pd.DataFrame({'Feature': df.columns, 'Missing Values': df.
 ↪isnull().sum().values})
print(missing_values_df)
```

```
                     Feature  Missing Values
0              Water Pump ID             250
1          Water Source Type               0
2              Water Quality               0
3   Distance to Nearest Town               0
4          Population Served               0
5          Installation Year               0
6                     Funder               0
7               Payment Type               0
8             Water Pump Age               0
9                  Pump Type               0
10           GPS Coordinates             250
11        Functioning Status               0
```

[37]:
```python
#Handling Outliers

# Calculate IQR for the 'Population Served' column
Q1 = df['Population Served'].quantile(0.25)
Q3 = df['Population Served'].quantile(0.75)
IQR = Q3 - Q1

# Calculate lower and upper bounds for outliers
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Identify outliers (values below lower_bound or above upper_bound)
outliers = df[(df['Population Served'] < lower_bound) | (df['Population␣
↪Served'] > upper_bound)]

# Option 1: Remove outliers
df_no_outliers = df[~df['Population Served'].isin(outliers['Population␣
↪Served'])]

# Option 2: Cap outliers (replace outliers with lower/upper bounds)
df['Population Served'] = df['Population Served'].clip(lower=lower_bound,␣
↪upper=upper_bound)

# Print the result
print("Outliers removed:")
print(outliers)
print("\nData after handling outliers:")
print(df.head())
```

```
Outliers removed:
     Water Pump ID Water Source Type Water Quality  Distance to Nearest Town  \
351          WP352              Lake         Clean                      20.0
1623        WP1624              Well         Clean                      37.0
1761        WP1762              Lake         Clean                      50.0
1782        WP1783              Well         Clean                      48.0
2114        WP2115              Lake         Clean                       6.0
2391        WP2392              Well         Clean                      34.0
2503        WP2504              Well         Clean                      35.0
2706        WP2707              Lake         Clean                      44.0
2932        WP2933              Lake         Clean                      45.0
3072        WP3073              Well  Contaminated                      13.0
3207        WP3208             River  Contaminated                      46.0
3518        WP3519              Lake  Contaminated                      12.0
3534        WP3535              Lake         Clean                      36.0
3684        WP3685              Lake  Contaminated                      15.0
4277        WP4278              Well  Contaminated                       7.0
4609        WP4610              Lake         Clean                      40.0
```

| | | | | |
|---|---|---|---|---|
| 4691 | WP4692 | Lake | Clean | 35.0 |
| 4766 | WP4767 | Lake | Clean | 28.0 |
| 4913 | WP4914 | Well | Contaminated | 37.0 |

| | Population Served | Installation Year | Funder | Payment Type | \ |
|---|---|---|---|---|---|
| 351 | 22000.0 | 2020.0 | Red Cross | Free | |
| 1623 | 4000.0 | 2006.0 | Oxfam | Pay per use | |
| 1761 | 22000.0 | 2014.0 | Red Cross | Pay per use | |
| 1782 | 22000.0 | 1998.0 | UNICEF | Pay per use | |
| 2114 | 22000.0 | 1991.0 | USAID | Pay per use | |
| 2391 | 4000.0 | 2012.0 | Red Cross | Pay per use | |
| 2503 | 3000.0 | 2001.0 | Oxfam | Pay per use | |
| 2706 | 22000.0 | 2004.0 | UNICEF | Pay per use | |
| 2932 | 4000.0 | 2006.0 | Red Cross | Pay per use | |
| 3072 | 4000.0 | 2006.0 | UNICEF | Free | |
| 3207 | 2000.0 | 1998.0 | Red Cross | Pay per use | |
| 3518 | 4000.0 | 2011.0 | Red Cross | Pay per use | |
| 3534 | 4000.0 | 2006.0 | Oxfam | Pay per use | |
| 3684 | 4000.0 | 2014.0 | Red Cross | Free | |
| 4277 | 22000.0 | 2012.0 | USAID | Pay per use | |
| 4609 | 4000.0 | 2016.0 | USAID | Pay per use | |
| 4691 | 22000.0 | 1993.0 | Red Cross | Pay per use | |
| 4766 | 22000.0 | 1990.0 | Oxfam | Free | |
| 4913 | 4000.0 | 2014.0 | Red Cross | Pay per use | |

| | Water Pump Age | Pump Type | \ |
|---|---|---|---|
| 351 | 13.0 | Hand Pump | |
| 1623 | 18.0 | Motorized Pump | |
| 1761 | 10.0 | Motorized Pump | |
| 1782 | 26.0 | Motorized Pump | |
| 2114 | 33.0 | Motorized Pump | |
| 2391 | 12.0 | Hand Pump | |
| 2503 | 23.0 | Hand Pump | |
| 2706 | 20.0 | Hand Pump | |
| 2932 | 18.0 | Motorized Pump | |
| 3072 | 18.0 | Hand Pump | |
| 3207 | 26.0 | Hand Pump | |
| 3518 | 28.0 | Motorized Pump | |
| 3534 | 18.0 | Hand Pump | |
| 3684 | 13.0 | Hand Pump | |
| 4277 | 12.0 | Solar Pump | |
| 4609 | 13.0 | Motorized Pump | |
| 4691 | 31.0 | Hand Pump | |
| 4766 | 34.0 | Motorized Pump | |
| 4913 | 10.0 | Motorized Pump | |

| | GPS Coordinates | Functioning Status |
|---|---|---|
| 351 | (-18.93340276075529, 25.208132833432785) | Not Functioning |

```
1623     (-22.5011154948989, 22.44232179713561)       Functioning
1761    (-19.91705627831145, 22.88334295790117)   Not Functioning
1782   (-21.395185744722475, 21.83400215915583)       Functioning
2114   (-18.445136023192276, 19.83568408473501)   Not Functioning
2391                                        NaN   Not Functioning
2503  (-21.846849904094423, 27.875202606048454)       Functioning
2706   (-20.115143959613338, 27.96842372307367)   Not Functioning
2932  (-23.880332866455312, 22.718900901753344)       Functioning
3072  (-21.947956908132944, 23.705769697466252)   Not Functioning
3207   (-23.82235931998412, 22.517421974546835)       Functioning
3518    (-22.3381539369204, 22.574595841673172)   Not Functioning
3534   (-22.93903359350564, 22.475883440760576)   Not Functioning
3684  (-22.014944464080397, 23.970354016215058)   Not Functioning
4277  (-19.175938524367503, 28.088288834906066)       Functioning
4609  (-18.829727335050492, 24.234630718789543)   Not Functioning
4691  (-20.885178466819852, 26.412262104843453)   Not Functioning
4766  (-21.689982610996893, 26.041732729255877)   Not Functioning
4913    (-21.307601730750495, 25.1211502927296)   Not Functioning

Data after handling outliers:
  Water Pump ID Water Source Type Water Quality  Distance to Nearest Town  \
0         WP001              Well         Clean                      44.0
1         WP002              Lake         Clean                      13.0
2         WP003              Lake         Clean                      27.0
3         WP004              Well         Clean                      14.0
4         WP005              Lake         Clean                      41.0

   Population Served  Installation Year      Funder Payment Type  \
0            13000.0             2006.0  World Bank         Free
1            13000.0             1990.0   Red Cross         Free
2            12000.0             1997.0       Oxfam  Pay per use
3             9000.0             1992.0       Oxfam  Pay per use
4            16000.0             2006.0       USAID  Pay per use

   Water Pump Age       Pump Type                             GPS Coordinates  \
0            18.0  Motorized Pump  (-20.599463060030295, 26.696000047794744)
1            34.0       Hand Pump   (-20.69129769992364, 23.313405231404484)
2            27.0       Hand Pump  (-19.830951420391948, 26.650358442338003)
3            32.0       Hand Pump   (-22.335866062765565, 22.83485684389231)
4            18.0       Hand Pump  (-21.099305692773278, 24.799143614430015)

  Functioning Status
0        Functioning
1    Not Functioning
2    Not Functioning
3        Functioning
4        Functioning
```

[38]:
```python
# FIXING INCONSISTENCIES

#Convert Installation Year to Integer
df['Installation Year'] = df['Installation Year'].astype(int)

#Fix GPS Coordinates (Split into Latitude & Longitude)
df[['Latitude', 'Longitude']] = df['GPS Coordinates'].str.extract(r'\((.*), (.
↪*)\)').astype(float)

#Checking the dataset top rows again after correcting inconsistencies:
print(df.head())
```

```
  Water Pump ID Water Source Type Water Quality  Distance to Nearest Town  \
0         WP001              Well         Clean                      44.0
1         WP002              Lake         Clean                      13.0
2         WP003              Lake         Clean                      27.0
3         WP004              Well         Clean                      14.0
4         WP005              Lake         Clean                      41.0

   Population Served  Installation Year      Funder Payment Type  \
0            13000.0               2006  World Bank         Free
1            13000.0               1990   Red Cross         Free
2            12000.0               1997       Oxfam  Pay per use
3             9000.0               1992       Oxfam  Pay per use
4            16000.0               2006       USAID  Pay per use

   Water Pump Age        Pump Type                             GPS Coordinates  \
0            18.0  Motorized Pump  (-20.599463060030295, 26.696000047794744)
1            34.0       Hand Pump   (-20.69129769992364, 23.313405231404484)
2            27.0       Hand Pump  (-19.830951420391948, 26.650358442338003)
3            32.0       Hand Pump   (-22.335866062765565, 22.83485684389231)
4            18.0       Hand Pump  (-21.099305692773278, 24.799143614430015)

  Functioning Status   Latitude  Longitude
0        Functioning -20.599463  26.696000
1    Not Functioning -20.691298  23.313405
2    Not Functioning -19.830951  26.650358
3        Functioning -22.335866  22.834857
4        Functioning -21.099306  24.799144
```

[39]:
```python
# Drop unnecessary columns
df = df.drop(['Water Pump ID', 'GPS Coordinates'], axis=1)
```

[40]:
```python
# MISSING DATA IMPUTATION:Remaining Features

#Imputation using KNNImputer of the remaining features
```

```
# Select relevant numeric columns (those that might relate to location)
numeric_cols = ['Latitude', 'Longitude', 'Distance to Nearest Town',
                'Population Served', 'Installation Year', 'Water Pump Age']

# Create a subset of the dataframe
df_numeric = df[numeric_cols]

# Initialize KNNImputer (k=5 is common)
imputer = KNNImputer(n_neighbors=5)

# Fit and transform the data
df_imputed = imputer.fit_transform(df_numeric)

# Create back a DataFrame and assign imputed Lat/Long to original df
df_imputed = pd.DataFrame(df_imputed, columns=numeric_cols)

# Replace original Latitude and Longitude with imputed ones
df['Latitude'] = df_imputed['Latitude']
df['Longitude'] = df_imputed['Longitude']
```

[41]:
```
#ONE HOT ENCODING

# One-hot encode categorical variables (excluding target for now)
df_encoded = pd.get_dummies(df, columns=['Water Source Type', 'Water Quality',
                                         'Funder', 'Payment Type', 'Pump␣
 ↪Type'], drop_first=True)

# Encode target column (Functioning: 1, Not Functioning: 0)
df_encoded['Functioning Status'] = df['Functioning Status'].map({'Functioning':␣
 ↪1, 'Not Functioning': 0})
```

[42]:
```
#Due to performance of Models not improving, we decided to drop irrelevant␣
 ↪features, which were earlier not dropped

# Since water pump age and Installation Year are strongly negatively␣
 ↪correlated,it is reasonable to keep only one to reduce collinearity. One␣
 ↪feature is enough to tell about the other.

# List of columns to drop
columns_to_drop_nn = [
    'Latitude', 'Longitude',
    'Funder_Red Cross', 'Funder_UNICEF', 'Funder_USAID', 'Funder_World Bank',
    'Payment Type_Pay per use',
    'Installation Year'
]

# Drop the specified columns from df_encoded
```

```
df_encoded = df_encoded.drop(columns=columns_to_drop_nn)

# Check to confirm they're gone
print("\nUpdated Column Names:\n", df_encoded.columns.tolist())
```

```
Updated Column Names:
 ['Distance to Nearest Town', 'Population Served', 'Water Pump Age',
'Functioning Status', 'Water Source Type_Lake', 'Water Source Type_River',
'Water Source Type_Well', 'Water Quality_Contaminated', 'Pump Type_Motorized
Pump', 'Pump Type_Solar Pump']
```

[43]:

```
# PREPARAING FOR MODELLING

# Define feature columns and target
X = df_encoded.drop(columns=['Functioning Status'])
y = df_encoded['Functioning Status']

# Split first
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=42, stratify=y)

# Apply SMOTE to training set
smote = SMOTE(random_state=42)
X_res, y_res = smote.fit_resample(X_train, y_train)

# Scale numeric columns
scaler = StandardScaler()
numeric_cols = ['Distance to Nearest Town', 'Population Served', 'Water Pump␣
 ↪Age']

X_res[numeric_cols] = scaler.fit_transform(X_res[numeric_cols])
X_test[numeric_cols] = scaler.transform(X_test[numeric_cols])
```

[44]:

```
# MODELING STARTS;
# For every Algorithm, first a baseline algorithm was ran and later versions␣
 ↪with improved parameters
# Logistic Regression (6) , Decsion Tree (5), Ensemble Method (3), Naive Bayes␣
 ↪(3), Neural Network (3)
```

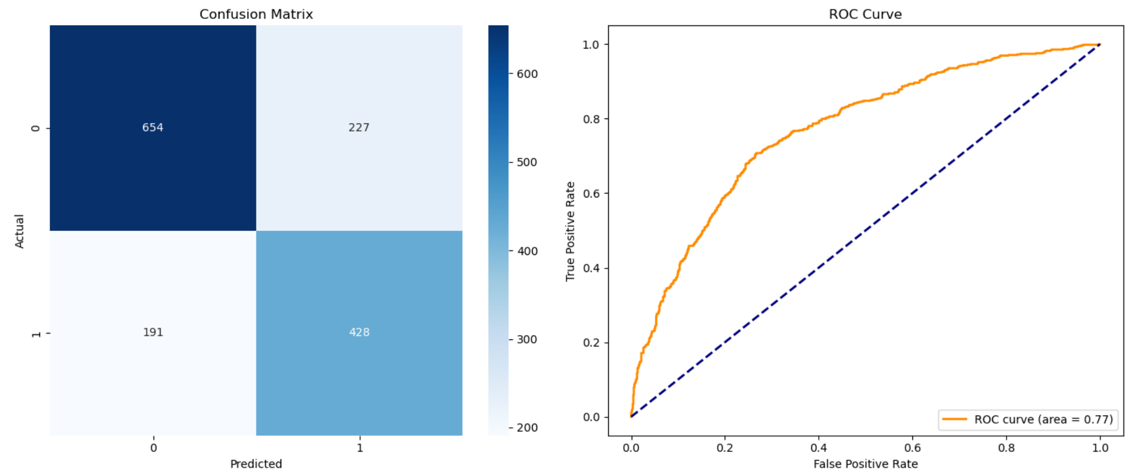
[45]:

```
# Logistic Regression 1: Basic Logistic Regression

# Set seed for reproducibility
SEED = 42
random.seed(SEED)
np.random.seed(SEED)
os.environ['PYTHONHASHSEED'] = str(SEED)
```

```python
# Basic Logistic Regression with random_state
model1 = LogisticRegression(max_iter=1000, random_state=SEED)
model1.fit(X_res, y_res)

train_acc1 = model1.score(X_res, y_res)
test_acc1 = model1.score(X_test, y_test)
y_test_pred1 = model1.predict(X_test)
y_pred_proba1 = model1.predict_proba(X_test)

print("Basic Logistic Regression")
print(f"Train Accuracy: {train_acc1:.4f}")
print(f"Test Accuracy: {test_acc1:.4f}")

# Create a figure with 1 row and 2 columns
fig, axes = plt.subplots(1, 2, figsize=(14, 6))

# Plot Confusion Matrix on the first subplot
cm = confusion_matrix(y_test, y_test_pred1)
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
            xticklabels=["0", "1"], yticklabels=["0", "1"],
            ax=axes[0])
axes[0].set_title("Confusion Matrix")
axes[0].set_xlabel("Predicted")
axes[0].set_ylabel("Actual")

# Plot ROC Curve on the second subplot
roc_auc = roc_auc_score(y_test, y_pred_proba1[:, 1])
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba1[:, 1])
axes[1].plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area =␣
↪{roc_auc:.2f})')
axes[1].plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
axes[1].set_xlabel('False Positive Rate')
axes[1].set_ylabel('True Positive Rate')
axes[1].set_title('ROC Curve')
axes[1].legend(loc='lower right')

# Adjust layout
plt.tight_layout()
plt.show()

print("\nClassification Report (Test Data):\n")
print(classification_report(y_test, y_test_pred1))

logloss1 = log_loss(y_test, y_pred_proba1)
print(f"Log Loss: {logloss1:.4f}")
```

```
Basic Logistic Regression
```

Train Accuracy: 0.7453
Test Accuracy: 0.7213



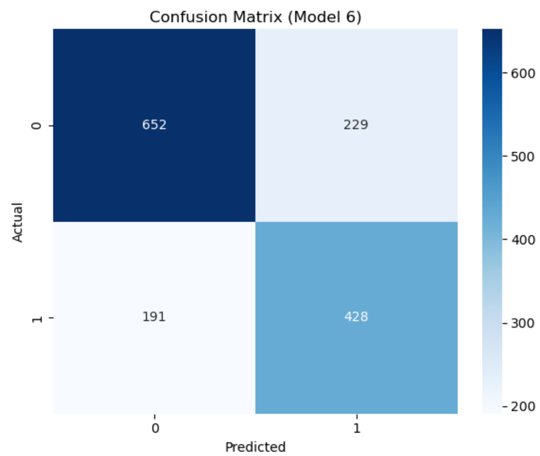

Classification Report (Test Data):

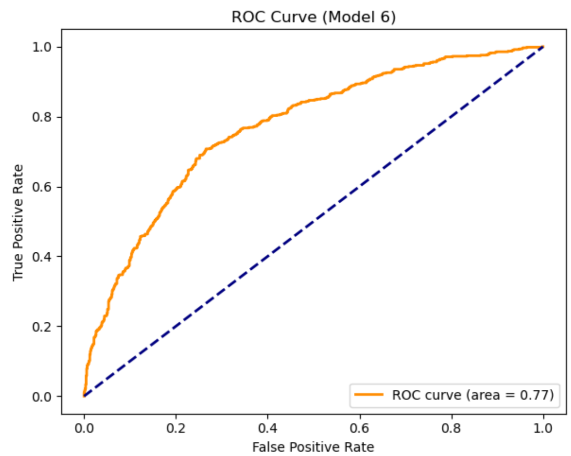
| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.77 | 0.74 | 0.76 | 881 |
| 1 | 0.65 | 0.69 | 0.67 | 619 |
| accuracy | | | 0.72 | 1500 |
| macro avg | 0.71 | 0.72 | 0.71 | 1500 |
| weighted avg | 0.72 | 0.72 | 0.72 | 1500 |

Log Loss: 0.5688

[46]:

Logistic Regression with Hyperparameter Tuning (Model 6)
Best Params: {'C': 1, 'penalty': 'l2', 'solver': 'liblinear'}
Train Accuracy: 0.7460
Test Accuracy: 0.7200
Log Loss: 0.5687




Classification Report (Test Data):

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.77 | 0.74 | 0.76 | 881 |
| 1 | 0.65 | 0.69 | 0.67 | 619 |
| accuracy | | | 0.72 | 1500 |
| macro avg | 0.71 | 0.72 | 0.71 | 1500 |
| weighted avg | 0.72 | 0.72 | 0.72 | 1500 |

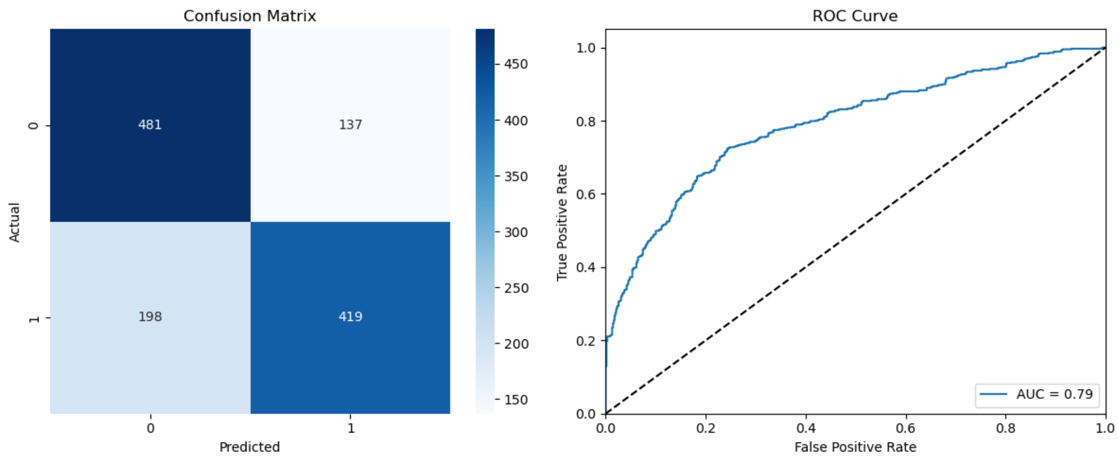
[47]:

```
Logistic Regression with Bagging
Train Accuracy: 0.7496
Test Accuracy: 0.7287
```

Classification Report (Test Data):

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.71 | 0.78 | 0.74 | 618 |
| 1 | 0.75 | 0.68 | 0.71 | 617 |
| accuracy | | | 0.73 | 1235 |
| macro avg | 0.73 | 0.73 | 0.73 | 1235 |
| weighted avg | 0.73 | 0.73 | 0.73 | 1235 |

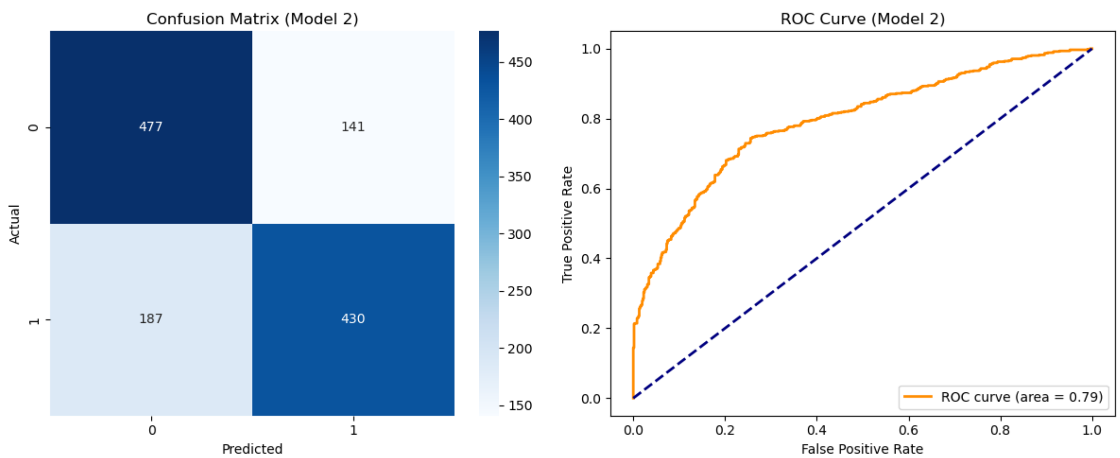
AUC: 0.79




[48]:
```
#Logistic Regression 2: with Regularization (with random state)

model2 = LogisticRegression(penalty='l2', C=1.0, solver='lbfgs', max_iter=1000,␣
↪random_state=42)
model2.fit(X_res, y_res)

train_acc2 = model2.score(X_res, y_res)
test_acc2 = model2.score(X_test, y_test)

cv_acc2 = cross_val_score(model2, X_res, y_res, cv=5, scoring='accuracy').mean()

# Predictions
y_test_pred2 = model2.predict(X_test)
y_pred_proba2 = model2.predict_proba(X_test)

# Log Loss
logloss2 = log_loss(y_test, y_pred_proba2)

print("\nLogistic Regression with Regularization (Model 2)")
print(f"Train Accuracy: {train_acc2:.4f}")
print(f"Test Accuracy: {test_acc2:.4f}")
print(f"Cross-Validated Accuracy: {cv_acc2:.4f}")
print(f"Log Loss: {logloss2:.4f}")

# Evaluation Metrics
cm = confusion_matrix(y_test, y_test_pred2)
roc_auc = roc_auc_score(y_test, y_pred_proba2[:, 1])
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba2[:, 1])

fig, axes = plt.subplots(1, 2, figsize=(12, 5))
```

```
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=["0", "1"],␣
↪yticklabels=["0", "1"], ax=axes[0])
axes[0].set_title("Confusion Matrix (Model 2)")
axes[0].set_xlabel("Predicted")
axes[0].set_ylabel("Actual")

axes[1].plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area =␣
↪{roc_auc:.2f})')
axes[1].plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
axes[1].set_xlabel('False Positive Rate')
axes[1].set_ylabel('True Positive Rate')
axes[1].set_title('ROC Curve (Model 2)')
axes[1].legend(loc='lower right')

plt.tight_layout()
plt.show()

print("\nClassification Report (Test Data):\n")
print(classification_report(y_test, y_test_pred2))
```

```
Logistic Regression with Regularization (Model 2)
Train Accuracy: 0.7453
Test Accuracy: 0.7344
Cross-Validated Accuracy: 0.7431
Log Loss: 0.5393
```




```
Classification Report (Test Data):

              precision    recall  f1-score   support
```

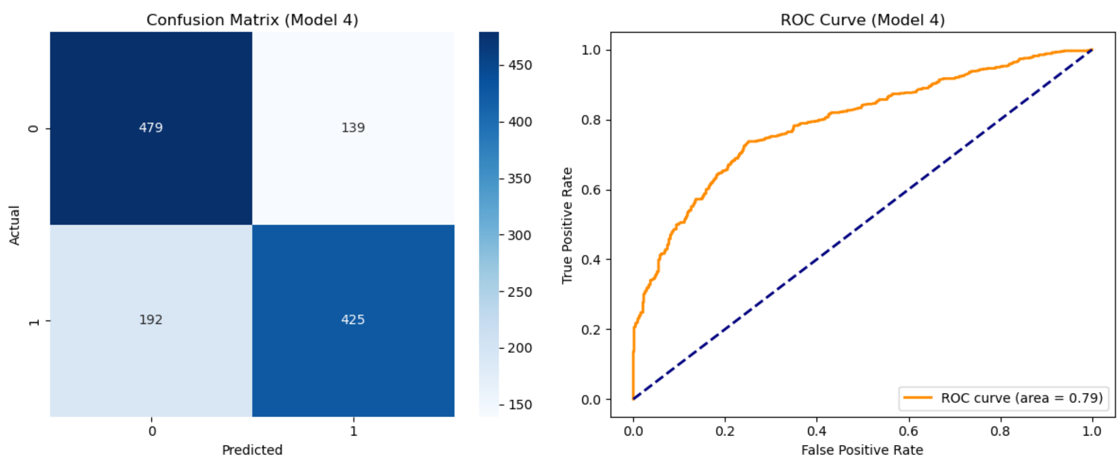
```
              0       0.72      0.77      0.74       618
              1       0.75      0.70      0.72       617

       accuracy                           0.73      1235
      macro avg       0.74      0.73      0.73      1235
   weighted avg       0.74      0.73      0.73      1235
```

[49]:

```
#Logistic Regression 3: with RFE and SMOTE (random state added to SMOTE and␣
 ↪model)

base_model = LogisticRegression(max_iter=1000, random_state=42)
selector = RFE(base_model, n_features_to_select=8)
selector.fit(X_train, y_train)

X_train_rfe = selector.transform(X_train)
X_test_rfe = selector.transform(X_test)

X_rfe_res, y_rfe_res = SMOTE(random_state=42).fit_resample(X_train_rfe, y_train)

model4 = LogisticRegression(max_iter=1000, random_state=42)
model4.fit(X_rfe_res, y_rfe_res)

train_acc4 = model4.score(X_rfe_res, y_rfe_res)
test_acc4 = model4.score(X_test_rfe, y_test)
cv_acc4 = cross_val_score(model4, X_rfe_res, y_rfe_res, cv=5,␣
 ↪scoring='accuracy').mean()

y_test_pred4 = model4.predict(X_test_rfe)
y_pred_proba4 = model4.predict_proba(X_test_rfe)
logloss4 = log_loss(y_test, y_pred_proba4)

print("\nLogistic Regression with RFE (Model 4)")
print(f"Train Accuracy: {train_acc4:.4f}")
print(f"Test Accuracy: {test_acc4:.4f}")
print(f"Cross-Validated Accuracy: {cv_acc4:.4f}")
print(f"Log Loss: {logloss4:.4f}")

cm = confusion_matrix(y_test, y_test_pred4)
roc_auc = roc_auc_score(y_test, y_pred_proba4[:, 1])
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba4[:, 1])

fig, axes = plt.subplots(1, 2, figsize=(12, 5))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=["0", "1"],␣
 ↪yticklabels=["0", "1"], ax=axes[0])
axes[0].set_title("Confusion Matrix (Model 4)")
axes[0].set_xlabel("Predicted")
```

```
axes[0].set_ylabel("Actual")

axes[1].plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area =␣
 ↪{roc_auc:.2f})')
axes[1].plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
axes[1].set_xlabel('False Positive Rate')
axes[1].set_ylabel('True Positive Rate')
axes[1].set_title('ROC Curve (Model 4)')
axes[1].legend(loc='lower right')

plt.tight_layout()
plt.show()

print("\nClassification Report (Test Data):\n")
print(classification_report(y_test, y_test_pred4))
```

```
Logistic Regression with RFE (Model 4)
Train Accuracy: 0.7490
Test Accuracy: 0.7320
Cross-Validated Accuracy: 0.7507
Log Loss: 0.5423
```




```
Classification Report (Test Data):

              precision    recall  f1-score   support

           0       0.71      0.78      0.74       618
           1       0.75      0.69      0.72       617

    accuracy                           0.73      1235
```

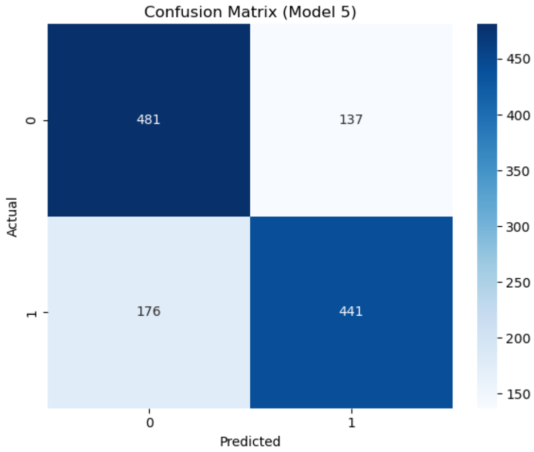
```
   macro avg       0.73      0.73      0.73      1235
weighted avg       0.73      0.73      0.73      1235
```

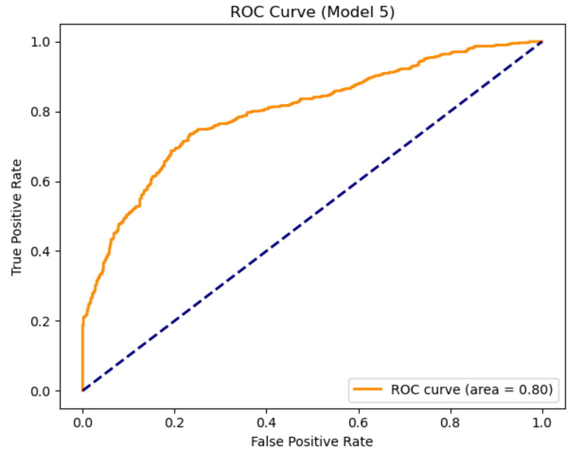
[50]:

```
#Logistic Regression 4: with Polynomial Features (random state added to model)

poly = PolynomialFeatures(degree=2, include_bias=False)
X_poly_train = poly.fit_transform(X_res)
X_poly_test = poly.transform(X_test)

model5 = LogisticRegression(max_iter=1000, random_state=42)
model5.fit(X_poly_train, y_res)

train_acc5 = model5.score(X_poly_train, y_res)
test_acc5 = model5.score(X_poly_test, y_test)

y_test_pred5 = model5.predict(X_poly_test)
y_pred_proba5 = model5.predict_proba(X_poly_test)
logloss5 = log_loss(y_test, y_pred_proba5)

print("\nLogistic Regression with Polynomial Features (Model 5)")
print(f"Train Accuracy: {train_acc5:.4f}")
print(f"Test Accuracy: {test_acc5:.4f}")
print(f"Log Loss: {logloss5:.4f}")

cm = confusion_matrix(y_test, y_test_pred5)
roc_auc = roc_auc_score(y_test, y_pred_proba5[:, 1])
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba5[:, 1])

fig, axes = plt.subplots(1, 2, figsize=(12, 5))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=["0", "1"],␣
 ↪yticklabels=["0", "1"], ax=axes[0])
axes[0].set_title("Confusion Matrix (Model 5)")
axes[0].set_xlabel("Predicted")
axes[0].set_ylabel("Actual")

axes[1].plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area =␣
 ↪{roc_auc:.2f})')
axes[1].plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
axes[1].set_xlabel('False Positive Rate')
axes[1].set_ylabel('True Positive Rate')
axes[1].set_title('ROC Curve (Model 5)')
axes[1].legend(loc='lower right')

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

```
print("\nClassification Report (Test Data):\n")
print(classification_report(y_test, y_test_pred5))
```

```
Logistic Regression with Polynomial Features (Model 5)
Train Accuracy: 0.7489
Test Accuracy: 0.7466
Log Loss: 0.5305
```




```
Classification Report (Test Data):

              precision    recall  f1-score   support

           0       0.73      0.78      0.75       618
           1       0.76      0.71      0.74       617

    accuracy                           0.75      1235
   macro avg       0.75      0.75      0.75      1235
weighted avg       0.75      0.75      0.75      1235
```

[ ]:
```
# Logistic Regression 5: with Hyperparameter Tuning (GridSearchCV)
param_grid = {
    'C': [0.01, 0.1, 1, 10],
    'penalty': ['l1', 'l2'],
    'solver': ['liblinear']
}
grid_model = GridSearchCV(LogisticRegression(max_iter=1000, random_state=42),␣
 ↪param_grid, cv=5)
grid_model.fit(X_res, y_res)
```

```
train_acc6 = grid_model.score(X_res, y_res)
test_acc6 = grid_model.score(X_test, y_test)

y_test_pred6 = grid_model.predict(X_test)
y_pred_proba6 = grid_model.predict_proba(X_test)
logloss6 = log_loss(y_test, y_pred_proba6)

print("\nLogistic Regression with Hyperparameter Tuning (Model 6)")
print(f"Best Params: {grid_model.best_params_}")
print(f"Train Accuracy: {train_acc6:.4f}")
print(f"Test Accuracy: {test_acc6:.4f}")
print(f"Log Loss: {logloss6:.4f}")

cm = confusion_matrix(y_test, y_test_pred6)
roc_auc = roc_auc_score(y_test, y_pred_proba6[:, 1])
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba6[:, 1])

fig, axes = plt.subplots(1, 2, figsize=(12, 5))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=["0", "1"],␣
↪yticklabels=["0", "1"], ax=axes[0])
axes[0].set_title("Confusion Matrix (Model 6)")
axes[0].set_xlabel("Predicted")
axes[0].set_ylabel("Actual")

axes[1].plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area =␣
↪{roc_auc:.2f})')
axes[1].plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
axes[1].set_xlabel('False Positive Rate')
axes[1].set_ylabel('True Positive Rate')
axes[1].set_title('ROC Curve (Model 6)')
axes[1].legend(loc='lower right')

plt.tight_layout()
plt.show()

print("\nClassification Report (Test Data):\n")
print(classification_report(y_test, y_test_pred6))
```

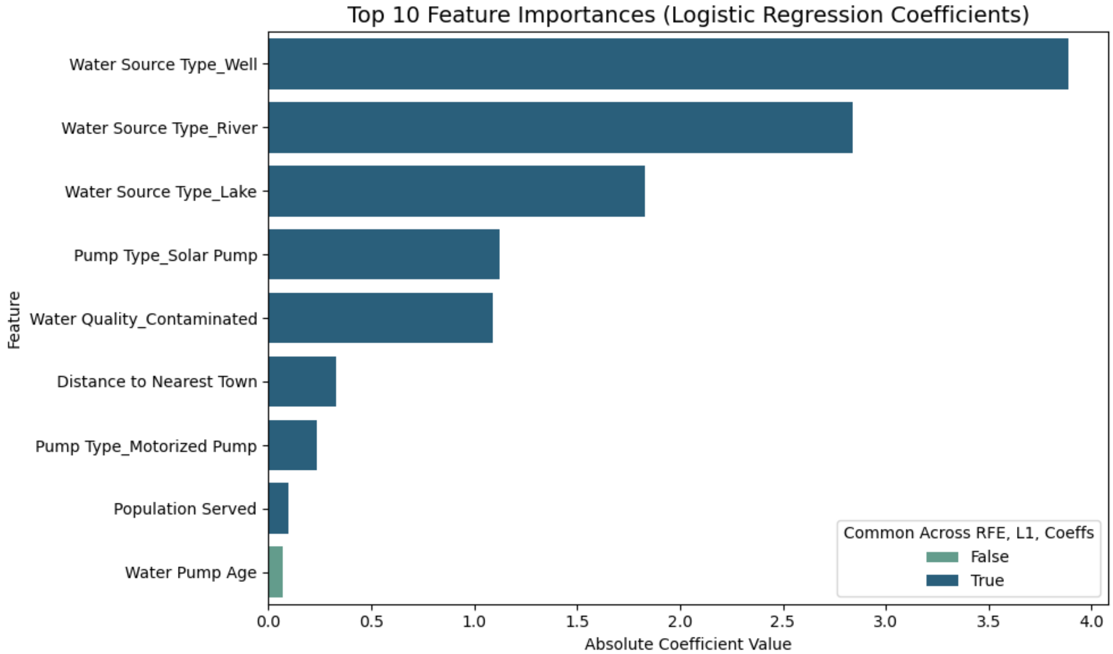
[51]:
```
#Logistic Regression 6: with Common Important Features
#Feature Importance Intersection
# Set random seed for reproducibility
np.random.seed(42)

# Feature importance
base_model = LogisticRegression(max_iter=1000, solver='liblinear',␣
↪random_state=42)
rfe = RFE(base_model, n_features_to_select=8)
```

```python
rfe.fit(X_train, y_train)
rfe_features = X_train.columns[rfe.support_]

l1_model = LogisticRegression(penalty='l1', solver='liblinear', C=1,␣
 ↪max_iter=1000, random_state=42)
l1_model.fit(X_train, y_train)
l1_features = X_train.columns[(l1_model.coef_[0] != 0)]

basic_model = LogisticRegression(max_iter=1000, random_state=42)
basic_model.fit(X_train, y_train)
coeff_basic = pd.Series(basic_model.coef_[0], index=X_train.columns)
top_basic = coeff_basic.abs().sort_values(ascending=False).head(10).index

common_features = set(rfe_features) & set(l1_features) & set(top_basic)
print("Common important features across RFE, L1, and Coefficients:\n",␣
 ↪common_features)

# Plot
palette = sns.color_palette("crest", n_colors=6)
sorted_coeffs = coeff_basic.abs().sort_values(ascending=False).head(10)

feature_importance_df = pd.DataFrame({
    'Feature': sorted_coeffs.index,
    'Importance': sorted_coeffs.values,
    'Common': sorted_coeffs.index.isin(common_features)
})
custom_colors = {True: palette[4], False: palette[1]}
plt.figure(figsize=(10, 6))
sns.barplot(
    data=feature_importance_df,
    x='Importance',
    y='Feature',
    hue='Common',
    dodge=False,
    palette=custom_colors
)
plt.title('Top 10 Feature Importances (Logistic Regression Coefficients)',␣
 ↪fontsize=14)
plt.xlabel('Absolute Coefficient Value')
plt.ylabel('Feature')
plt.legend(title='Common Across RFE, L1, Coeffs')
plt.tight_layout()
plt.show()

# Select only the common important features
X_train_selected = X_train[list(common_features)]
X_test_selected = X_test[list(common_features)]
```

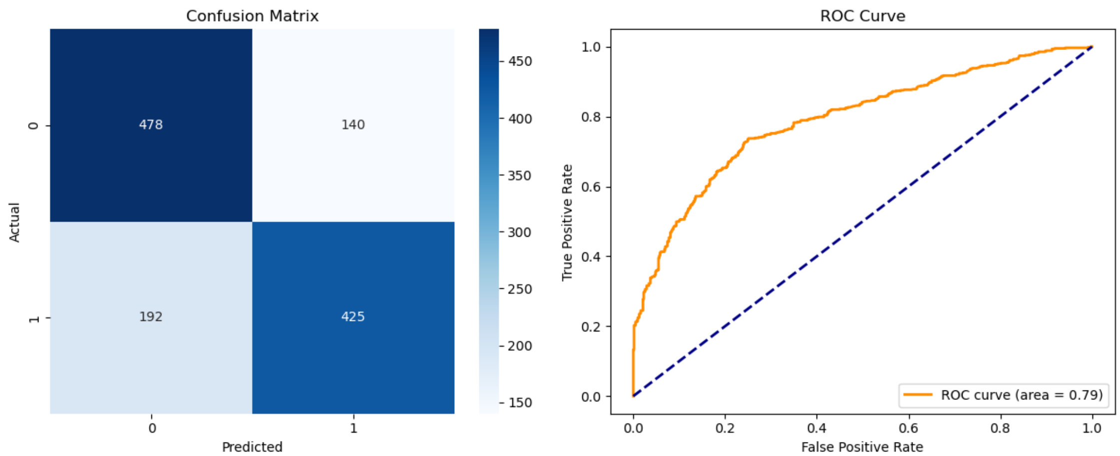
```python
# Build the model
model_selected = LogisticRegression(max_iter=1000, solver='liblinear', random_state=42)
model_selected.fit(X_train_selected, y_train)

# Predict on train and test data
y_train_pred = model_selected.predict(X_train_selected)
y_test_pred = model_selected.predict(X_test_selected)
y_pred_proba_selected = model_selected.predict_proba(X_test_selected)

# Evaluate
train_accuracy = accuracy_score(y_train, y_train_pred)
test_accuracy = accuracy_score(y_test, y_test_pred)
logloss_selected = log_loss(y_test, y_pred_proba_selected)

print("\nLogistic Regression with Selected Features")
print(f"Train Accuracy (Selected Features): {train_accuracy:.4f}")
print(f"Test Accuracy (Selected Features): {test_accuracy:.4f}")
print(f"Log Loss: {logloss_selected:.4f}")

# Metrics
cm = confusion_matrix(y_test, y_test_pred)
roc_auc = roc_auc_score(y_test, y_pred_proba_selected[:, 1])
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba_selected[:, 1])

# Set up side-by-side plots
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# Confusion Matrix
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
            xticklabels=["0", "1"],
            yticklabels=["0", "1"],
            ax=axes[0])
axes[0].set_title("Confusion Matrix")
axes[0].set_xlabel("Predicted")
axes[0].set_ylabel("Actual")

# ROC Curve
axes[1].plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {roc_auc:.2f})')
axes[1].plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
axes[1].set_xlabel('False Positive Rate')
axes[1].set_ylabel('True Positive Rate')
axes[1].set_title('ROC Curve')
axes[1].legend(loc='lower right')
```

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

print("\nClassification Report (Test Data):\n")
print(classification_report(y_test, y_test_pred))
```

```
Common important features across RFE, L1, and Coefficients:
 {'Population Served', 'Water Source Type_River', 'Distance to Nearest Town',
'Pump Type_Motorized Pump', 'Pump Type_Solar Pump', 'Water Source Type_Well',
'Water Source Type_Lake', 'Water Quality_Contaminated'}
```




```
Logistic Regression with Selected Features
Train Accuracy (Selected Features): 0.7492
Test Accuracy (Selected Features): 0.7312
Log Loss: 0.5426
```




```
Classification Report (Test Data):

              precision    recall  f1-score   support

           0       0.71      0.77      0.74       618
           1       0.75      0.69      0.72       617

    accuracy                           0.73      1235
   macro avg       0.73      0.73      0.73      1235
weighted avg       0.73      0.73      0.73      1235
```

[52]:

```
#Decision Tree Modelling
```

[53]:

```
# Define a function to evaluate models

def evaluate_model(model, X_train, y_train, X_test, y_test, model_name="Model"):
    print(f"\n=== {model_name} ===")

    # Predictions
    y_pred_train = model.predict(X_train)
    y_pred_test = model.predict(X_test)

    # Probabilities for ROC
    if hasattr(model, "predict_proba"):
        y_probs_test = model.predict_proba(X_test)[:, 1]
    else:
        y_probs_test = None

    # Accuracy
```

```
    print(f"Train Accuracy: {accuracy_score(y_train, y_pred_train):.4f}")
    print(f"Test Accuracy: {accuracy_score(y_test, y_pred_test):.4f}")

    # Precision, Recall, F1-Score
    print("\nClassification Report (Test Data):")
    print(classification_report(y_test, y_pred_test))

    # ROC-AUC
    if y_probs_test is not None:
        roc_auc = roc_auc_score(y_test, y_probs_test)
        print(f"ROC-AUC Score (Test Data): {roc_auc:.4f}")

    # 5-fold Cross Validation Accuracy
    cv_scores = cross_val_score(model, X_train, y_train, cv=5,␣
↪scoring='accuracy')
    print(f"Cross-Validation Accuracy (mean ± std): {cv_scores.mean():.4f} ±␣
↪{cv_scores.std():.4f}")

    # Plot Confusion Matrix and ROC Curve side-by-side
    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    # Confusion Matrix
    cm = confusion_matrix(y_test, y_pred_test)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm)
    disp.plot(ax=axes[0], values_format='d')
    axes[0].set_title(f"{model_name} - Confusion Matrix")

    # ROC Curve
    if y_probs_test is not None:
        fpr, tpr, _ = roc_curve(y_test, y_probs_test)
        RocCurveDisplay(fpr=fpr, tpr=tpr).plot(ax=axes[1])
        axes[1].set_title(f"{model_name} - ROC Curve")

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

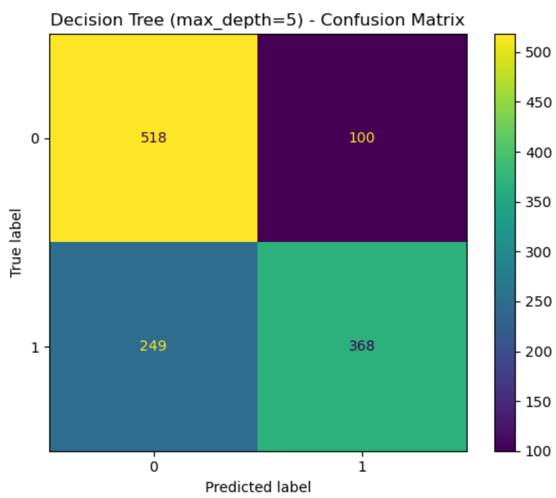
[54]:
```
# Model 1 Basic Decision Tree
print("\n=== DECISION TREE v1 - BASIC ===")
dt1 = DecisionTreeClassifier(random_state=42)
dt1.fit(X_train, y_train)
evaluate_model(dt1, X_train, y_train, X_test, y_test, model_name="Decision Tree␣
↪v1 (Basic)")

# Model 2 Pruned Decision Tree (Max_Depth=5):
print("\n=== DECISION TREE Max_Depth=5 ===")
dt = DecisionTreeClassifier(max_depth=5, random_state=42)
dt.fit(X_train, y_train)
```

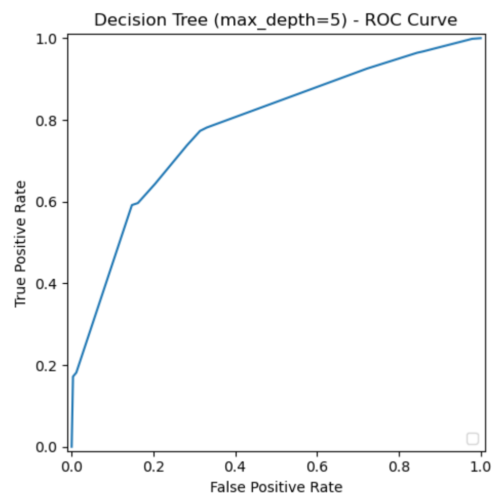
```
evaluate_model(dt, X_train, y_train, X_test, y_test, model_name="Decision Tree␣
↪(max_depth=5)")

# Model 3 Optimized Tree Depth (Max_Depth=6)
print("\n=== DECISION TREE v2 - Tuned max_depth=6 ===")
dt2 = DecisionTreeClassifier(max_depth=6, random_state=42)
dt2.fit(X_train, y_train)
evaluate_model(dt2, X_train, y_train, X_test, y_test, model_name="Decision Tree␣
↪v2 (max_depth=6)")

# Model 4 Criterion and Minimum Samples Split Optimization (Entropy, depth=8,␣
↪min_samples_split=10)
print("\n=== DECISION TREE v3 - Entropy, Depth=8, Min Split=10 ===")
dt3 = DecisionTreeClassifier(criterion='entropy', max_depth=8,␣
↪min_samples_split=10, random_state=42)
dt3.fit(X_train, y_train)
evaluate_model(dt3, X_train, y_train, X_test, y_test, model_name="Decision Tree␣
↪v3 (Entropy, depth=8, split=10)")

# Model 5 Tuned Decision Tree (SMOTE + Feature Selection)
print("\n=== DECISION TREE - Tuned with SMOTE + Feature Selection ===")
# SMOTE resampling
smote = SMOTE(random_state=42)
X_train_res, y_train_res = smote.fit_resample(X_train, y_train)

# Feature selection
selector = SelectFromModel(RandomForestClassifier(n_estimators=100,␣
↪random_state=42))
selector.fit(X_train_res, y_train_res)
X_train_sel = selector.transform(X_train_res)
X_test_sel = selector.transform(X_test)

# Hyperparameter tuning
dt_params = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [6, 8, 10, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 3, 5],
    'class_weight': [None, 'balanced']
}
dt_grid = GridSearchCV(DecisionTreeClassifier(random_state=42), dt_params,␣
↪cv=5, scoring='accuracy')
dt_grid.fit(X_train_sel, y_train_res)

dt_best = dt_grid.best_estimator_
print("\nBest Parameters from Grid Search:", dt_grid.best_params_)
```

```
evaluate_model(dt_best, X_train_sel, y_train_res, X_test_sel, y_test,␣
↪model_name="Tuned Decision Tree (GridSearch)")
```

```
=== DECISION TREE Max_Depth=5 ===

=== Decision Tree (max_depth=5) ===
Train Accuracy: 0.7471
Test Accuracy: 0.7174

Classification Report (Test Data):
              precision    recall  f1-score   support

           0       0.68      0.84      0.75       618
           1       0.79      0.60      0.68       617

    accuracy                           0.72      1235
   macro avg       0.73      0.72      0.71      1235
weighted avg       0.73      0.72      0.71      1235

ROC-AUC Score (Test Data): 0.7828
Cross-Validation Accuracy (mean ± std): 0.7339 ± 0.0198
```

```
C:\Users\HP\anaconda3\Lib\site-packages\sklearn\metrics\_plot\roc_curve.py:189:
UserWarning: No artists with labels found to put in legend.  Note that artists
whose label start with an underscore are ignored when legend() is called with no
argument.
  self.ax_.legend(loc="lower right")
```

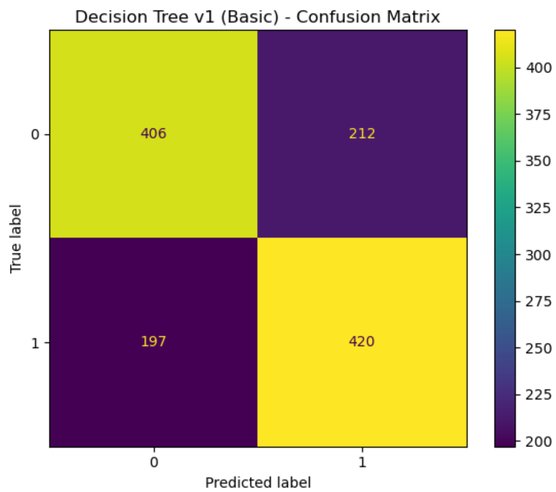


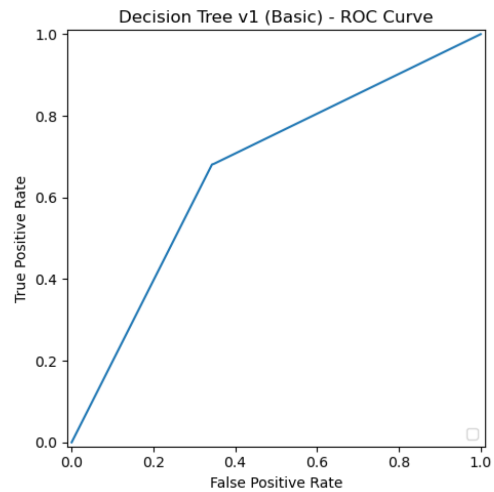




```
=== DECISION TREE v1 - BASIC ===
```

```
=== Decision Tree v1 (Basic) ===
Train Accuracy: 0.9951
Test Accuracy: 0.6688

Classification Report (Test Data):
              precision    recall  f1-score   support

           0       0.67      0.66      0.67       618
           1       0.66      0.68      0.67       617

    accuracy                           0.67      1235
   macro avg       0.67      0.67      0.67      1235
weighted avg       0.67      0.67      0.67      1235

ROC-AUC Score (Test Data): 0.6686
Cross-Validation Accuracy (mean ± std): 0.6488 ± 0.0188
```

```
C:\Users\HP\anaconda3\Lib\site-packages\sklearn\metrics\_plot\roc_curve.py:189:
UserWarning: No artists with labels found to put in legend.  Note that artists
whose label start with an underscore are ignored when legend() is called with no
argument.
  self.ax_.legend(loc="lower right")
```

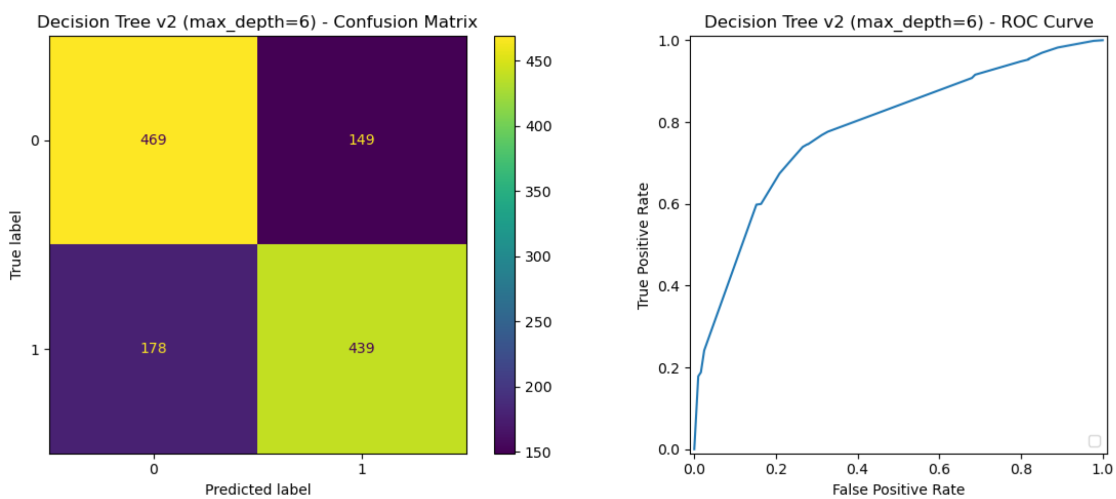



```
=== DECISION TREE v2 - Tuned max_depth=6 ===

=== Decision Tree v2 (max_depth=6) ===
Train Accuracy: 0.7614
Test Accuracy: 0.7352
```

```
Classification Report (Test Data):
              precision    recall  f1-score   support

           0       0.72      0.76      0.74       618
           1       0.75      0.71      0.73       617

    accuracy                           0.74      1235
   macro avg       0.74      0.74      0.74      1235
weighted avg       0.74      0.74      0.74      1235

ROC-AUC Score (Test Data): 0.7848
Cross-Validation Accuracy (mean ± std): 0.7343 ± 0.0238
```

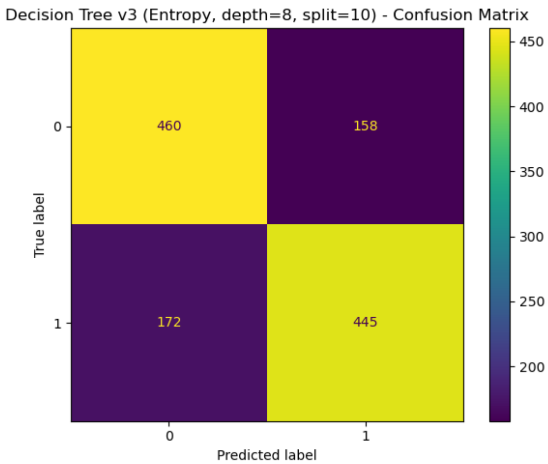
```
C:\Users\HP\anaconda3\Lib\site-packages\sklearn\metrics\_plot\roc_curve.py:189:
UserWarning: No artists with labels found to put in legend.  Note that artists
whose label start with an underscore are ignored when legend() is called with no
argument.
  self.ax_.legend(loc="lower right")
```

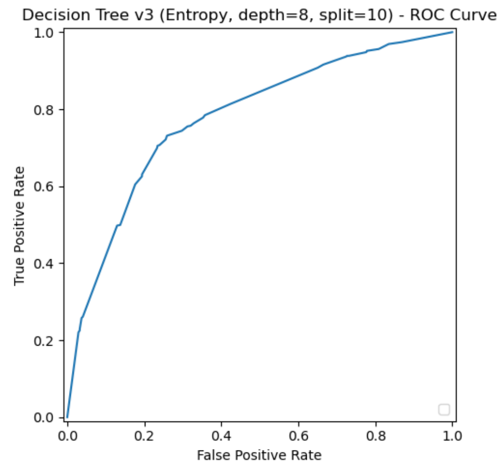



```
=== DECISION TREE v3 - Entropy, Depth=8, Min Split=10 ===

=== Decision Tree v3 (Entropy, depth=8, split=10) ===
Train Accuracy: 0.7714
Test Accuracy: 0.7328

Classification Report (Test Data):
              precision    recall  f1-score   support

           0       0.73      0.74      0.74       618
           1       0.74      0.72      0.73       617
```

```
    accuracy                           0.73      1235
   macro avg       0.73      0.73      0.73      1235
weighted avg       0.73      0.73      0.73      1235

ROC-AUC Score (Test Data): 0.7784
Cross-Validation Accuracy (mean ± std): 0.7367 ± 0.0191
```

```
C:\Users\HP\anaconda3\Lib\site-packages\sklearn\metrics\_plot\roc_curve.py:189:
UserWarning: No artists with labels found to put in legend.  Note that artists
whose label start with an underscore are ignored when legend() is called with no
argument.
  self.ax_.legend(loc="lower right")
```

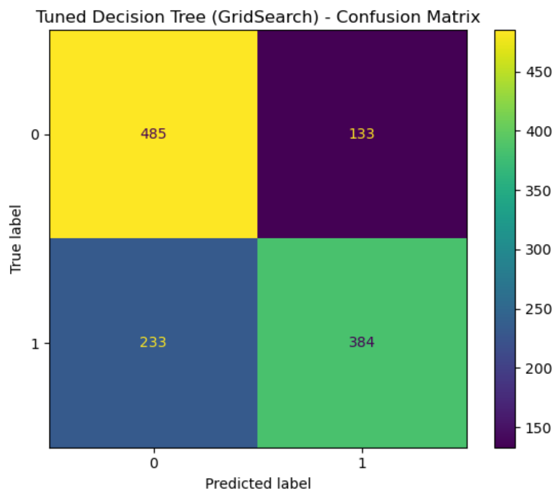


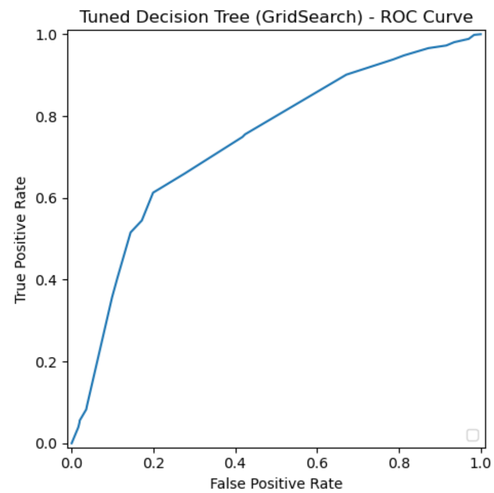




```
=== DECISION TREE - Tuned with SMOTE + Feature Selection ===

Best Parameters from Grid Search: {'class_weight': None, 'criterion': 'entropy',
'max_depth': 6, 'min_samples_leaf': 3, 'min_samples_split': 10}

=== Tuned Decision Tree (GridSearch) ===
Train Accuracy: 0.7347
Test Accuracy: 0.7036

Classification Report (Test Data):
              precision    recall  f1-score   support

           0       0.68      0.78      0.73       618
           1       0.74      0.62      0.68       617

    accuracy                           0.70      1235
   macro avg       0.71      0.70      0.70      1235
```

```
weighted avg       0.71      0.70      0.70      1235

ROC-AUC Score (Test Data): 0.7370
Cross-Validation Accuracy (mean ± std): 0.7267 ± 0.0186
```

```
C:\Users\HP\anaconda3\Lib\site-packages\sklearn\metrics\_plot\roc_curve.py:189:
UserWarning: No artists with labels found to put in legend.  Note that artists
whose label start with an underscore are ignored when legend() is called with no
argument.
  self.ax_.legend(loc="lower right")
```




[55]:
```
#Ensemble Method
```

[56]:
```
print("\n=== RANDOM FOREST MODELS ===")

# Helper function to evaluate models
def evaluate_model(model, X_train, X_test, y_train, y_test):
    train_acc = model.score(X_train, y_train)
    test_acc = model.score(X_test, y_test)
    overfit_gap = train_acc - test_acc
    y_pred = model.predict(X_test)
    y_proba = model.predict_proba(X_test)[:,1]

    # Cross-validation
    cv_scores = cross_val_score(model, X_train, y_train, cv=5)

    # Metrics
    roc_auc = roc_auc_score(y_test, y_proba)
    f1 = f1_score(y_test, y_pred)
    cm = confusion_matrix(y_test, y_pred)
    cr = classification_report(y_test, y_pred)
```

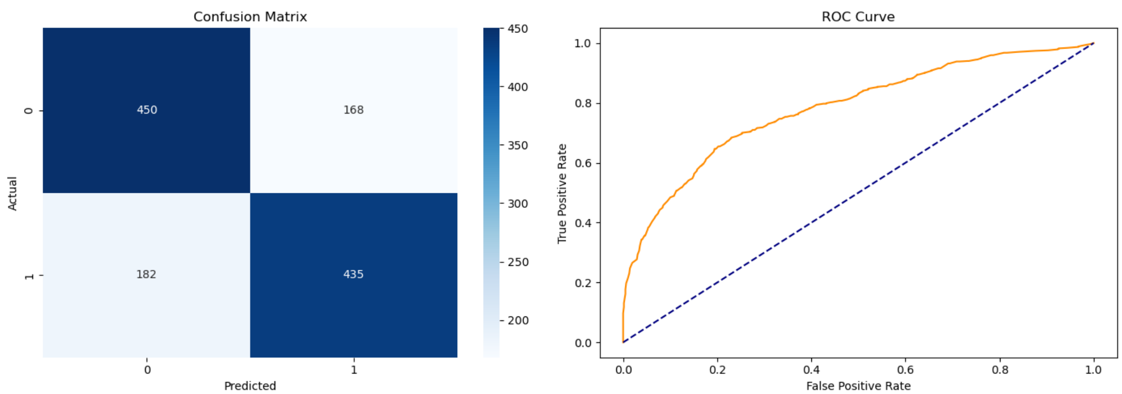
```
    # Print Metrics
    print(f"Train Accuracy: {train_acc:.4f}")
    print(f"Test Accuracy: {test_acc:.4f}")
    print(f"Overfitting Gap (Train - Test): {overfit_gap:.4f}")
    print(f"Cross-Validation Score: {cv_scores.mean():.4f} ± {cv_scores.std():.
 ↪4f}")
    print(f"ROC-AUC Score: {roc_auc:.4f}")
    print(f"F1-Score: {f1:.4f}")
    print("\nClassification Report:\n", cr)

    # Side-by-side Plots
    fig, ax = plt.subplots(1, 2, figsize=(14, 5))

    # Confusion Matrix
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", ax=ax[0])
    ax[0].set_title('Confusion Matrix')
    ax[0].set_xlabel('Predicted')
    ax[0].set_ylabel('Actual')

    # ROC Curve
    fpr, tpr, thresholds = roc_curve(y_test, y_proba)
    ax[1].plot(fpr, tpr, color='darkorange')
    ax[1].plot([0, 1], [0, 1], color='navy', linestyle='--')
    ax[1].set_title('ROC Curve')
    ax[1].set_xlabel('False Positive Rate')
    ax[1].set_ylabel('True Positive Rate')

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

=== RANDOM FOREST MODELS ===

[57]:
```
# ENSEMBLE METHOD

# Random Forest 1: Basic Random Forest
rf1 = RandomForestClassifier(random_state=42)
rf1.fit(X_train, y_train)
print("\nRandom Forest v1 - Basic:")
evaluate_model(rf1, X_train, X_test, y_train, y_test)

# Random Forest 2: With Constrained Random Forest (Tuned with n_estimators=100,␣
 ↪max_depth=10
rf2 = RandomForestClassifier(n_estimators=100, max_depth=10, random_state=42)
rf2.fit(X_train, y_train)
print("\nRandom Forest v2 - n_estimators=100, max_depth=10:")
```

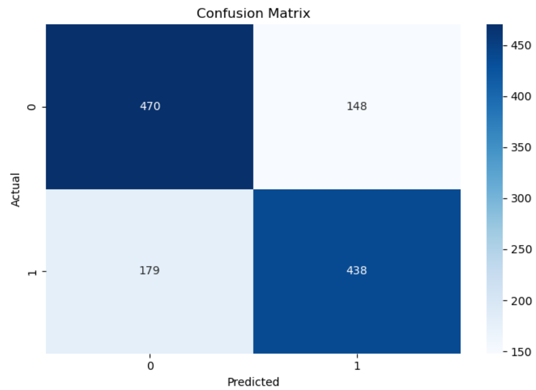
```
evaluate_model(rf2, X_train, X_test, y_train, y_test)
```

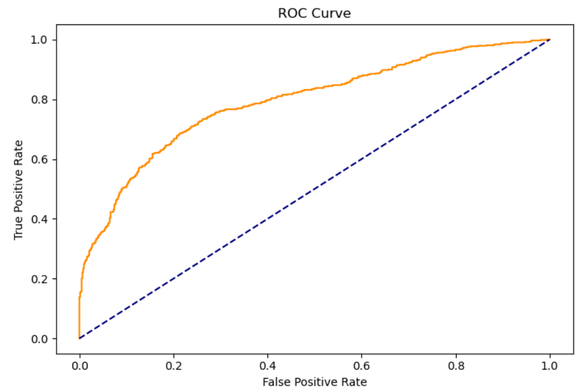
```
Random Forest v1 - Basic:
Train Accuracy: 0.9951
Test Accuracy: 0.7166
Overfitting Gap (Train - Test): 0.2785
Cross-Validation Score: 0.7221 ± 0.0139
ROC-AUC Score: 0.7830
F1-Score: 0.7131

Classification Report:
              precision    recall  f1-score   support

           0       0.71      0.73      0.72       618
           1       0.72      0.71      0.71       617

    accuracy                           0.72      1235
   macro avg       0.72      0.72      0.72      1235
weighted avg       0.72      0.72      0.72      1235
```




```
Random Forest v2 - n_estimators=100, max_depth=10:
Train Accuracy: 0.8229
Test Accuracy: 0.7352
Overfitting Gap (Train - Test): 0.0876
Cross-Validation Score: 0.7454 ± 0.0183
ROC-AUC Score: 0.7934
F1-Score: 0.7282

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

| | | | | |
|---|---|---|---|---|
| 0 | 0.72 | 0.76 | 0.74 | 618 |
| 1 | 0.75 | 0.71 | 0.73 | 617 |
| | | | | |
| accuracy | | | 0.74 | 1235 |
| macro avg | 0.74 | 0.74 | 0.74 | 1235 |
| weighted avg | 0.74 | 0.74 | 0.74 | 1235 |




[ ]:
```
# Ensemble Method 3

#Bagging with Logistic Regression(Random Seed)

# Set random seed to ensure reproducibility
random_seed = 42

# Ensure consistent train-test split with random_state
X_train, X_test, y_train, y_test = train_test_split(X_res, y_res, test_size=0.
↪3, random_state=random_seed)

# Initialize the base model
base_model = LogisticRegression(random_state=random_seed)

# Create the bagging ensemble with the random_state parameter
bagging_model = BaggingClassifier(estimator=base_model, n_estimators=10,␣
↪random_state=random_seed)

# Fit on the resampled training data
bagging_model.fit(X_train, y_train)

# Predictions
y_train_pred = bagging_model.predict(X_train)
y_test_pred = bagging_model.predict(X_test)
```

```python
# Calculate probabilities for ROC curve
y_test_proba = bagging_model.predict_proba(X_test)[:, 1]

# Performance metrics
train_accuracy = accuracy_score(y_train, y_train_pred)
test_accuracy = accuracy_score(y_test, y_test_pred)

print(f"Logistic Regression with Bagging")
print(f"Train Accuracy: {train_accuracy:.4f}")
print(f"Test Accuracy: {test_accuracy:.4f}")

# Confusion matrix
cm = confusion_matrix(y_test, y_test_pred)

# Classification report
print("\nClassification Report (Test Data):")
print(classification_report(y_test, y_test_pred))

# ROC curve calculation
fpr, tpr, _ = roc_curve(y_test, y_test_proba)
roc_auc = auc(fpr, tpr)
print(f"AUC: {roc_auc:.2f}")

# Plot Confusion Matrix and ROC Curve side-by-side
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# Confusion Matrix
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", ax=axes[0])
axes[0].set_title('Confusion Matrix')
axes[0].set_xlabel('Predicted')
axes[0].set_ylabel('Actual')

# ROC Curve
axes[1].plot(fpr, tpr, label=f'AUC = {roc_auc:.2f}')
axes[1].plot([0, 1], [0, 1], 'k--')  # Random guess line
axes[1].set_xlim([0.0, 1.0])
axes[1].set_ylim([0.0, 1.05])
axes[1].set_xlabel('False Positive Rate')
axes[1].set_ylabel('True Positive Rate')
axes[1].set_title('ROC Curve')
axes[1].legend(loc="lower right")

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

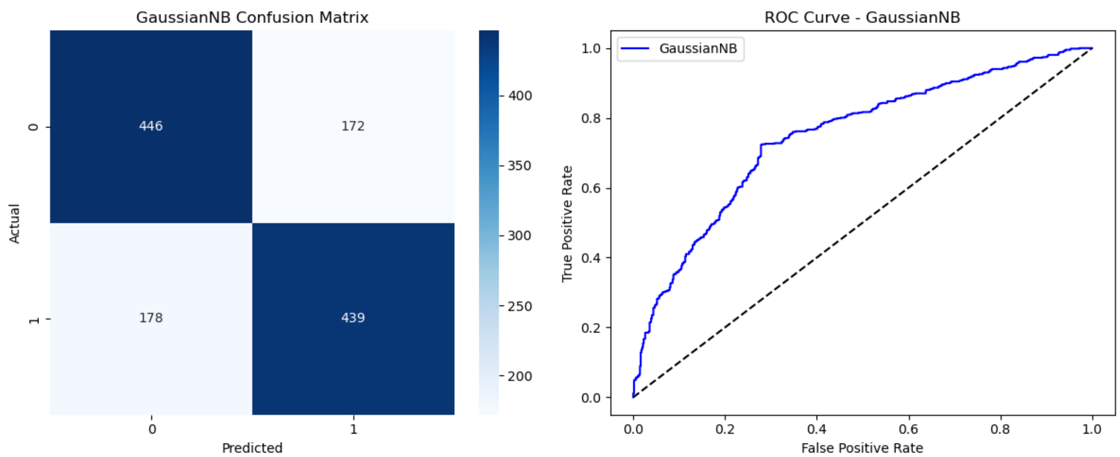
[58]:

```python
#Naive Bayes
```

[59]:
```
# Model 1: Gaussian Naive Bayes

# Gaussian Naive Bayes assumes features are continuous and normally distributed␣
↪(Gaussian).

# Initialize the GaussianNB model and Train the model using the training data
gnb = GaussianNB()
gnb.fit(X_res, y_res)  # Corrected to use X_res and y_res

# Predict outcomes on training and testing datasets
y_train_pred = gnb.predict(X_res)
y_test_pred = gnb.predict(X_test)

# Evaluate performance by checking training and testing accuracies
train_acc_gnb = accuracy_score(y_res, y_train_pred)
test_acc_gnb = accuracy_score(y_test, y_test_pred)

print(f"Training Accuracy (GaussianNB): {train_acc_gnb:.4f}")
print(f"Testing Accuracy (GaussianNB): {test_acc_gnb:.4f}")

# Overfitting check
if abs(train_acc_gnb - test_acc_gnb) < 0.05:
    print("No major overfitting detected for GaussianNB.")
else:
    print("Possible overfitting detected for GaussianNB.")

# To visualize performance:
# Confusion Matrix and ROC Curve Side-by-Side
fig, axes = plt.subplots(ncols=2, figsize=(12, 5))

# Confusion Matrix
cm = confusion_matrix(y_test, y_test_pred)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=np.unique(y_test_pred),
            yticklabels=np.unique(y_test), ax=axes[0])
axes[0].set_title("GaussianNB Confusion Matrix")
axes[0].set_xlabel("Predicted")
axes[0].set_ylabel("Actual")

# ROC Curve
y_test_binary = y_test.astype(int)

y_probs = gnb.predict_proba(X_test)[:, 1]
fpr, tpr, _ = roc_curve(y_test_binary, y_probs)

axes[1].plot(fpr, tpr, color='blue', label='GaussianNB')
axes[1].plot([0, 1], [0, 1], 'k--')
```

```
axes[1].set_title("ROC Curve - GaussianNB")
axes[1].set_xlabel("False Positive Rate")
axes[1].set_ylabel("True Positive Rate")
axes[1].legend()

plt.tight_layout()
plt.show()

print("AUC Score (GaussianNB):", roc_auc_score(y_test_binary, y_probs))
```

```
Training Accuracy (GaussianNB): 0.7253
Testing Accuracy (GaussianNB): 0.7166
No major overfitting detected for GaussianNB.
```



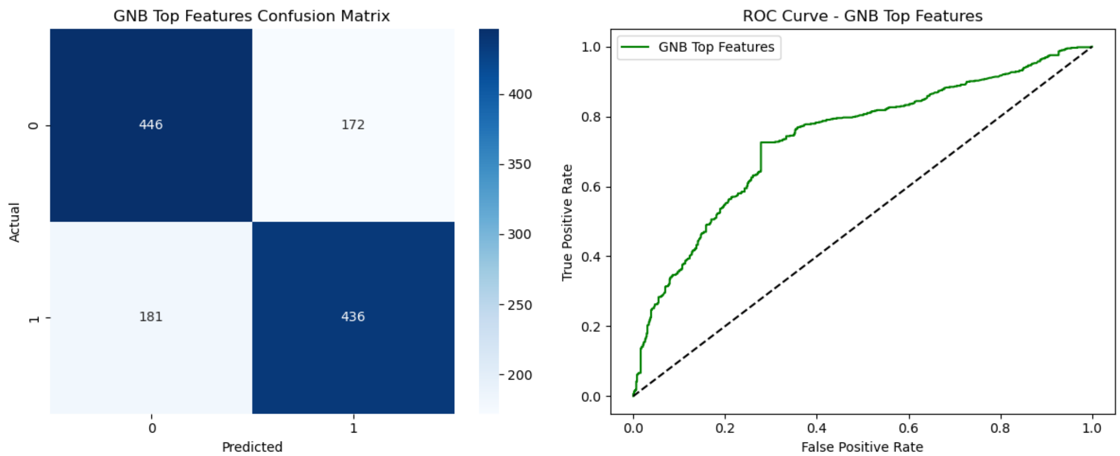

```
AUC Score (GaussianNB): 0.7476055976040241
```

[60]:
```
# Method 2. GaussianNB Improvement: Manual Top Feature Selection
# ----------------------------------------------------------------------

# Step 1: Map 'Functioning' and 'Not Functioning' to binary (0/1)
y_binary = y.map({'Not Functioning': 0, 'Functioning': 1})

# Temporarily add the binary target to feature dataframe
df_temp = X.copy()
df_temp['Functioning Status'] = y_binary.values

# Now compute correlation
correlations = df_temp.corr()['Functioning Status'].abs().
 ↪sort_values(ascending=False)

# Select top 20 features most correlated with 'Functioning Status'
top_features = correlations.index[1:21]  # Exclude the target itself
```

```
# Ensure 'Functioning Status' is excluded from top_features
top_features = [feature for feature in top_features if feature != 'Functioning␣
↪Status']

print("Top 20 features selected based on highest correlation:\n", top_features)

# Step 2: Subset X_res and X_test using only the selected features
X_res_top = X_res[top_features]
X_test_top = X_test[top_features]

# Step 3: Train GaussianNB on top selected features
gnb_top = GaussianNB()
gnb_top.fit(X_res_top, y_res)

# Step 4: Predict outcomes on training and testing datasets
y_train_pred_top = gnb_top.predict(X_res_top)
y_test_pred_top = gnb_top.predict(X_test_top)

# Step 5: Evaluate performance by checking training and testing accuracies
train_acc_gnb_top = accuracy_score(y_res, y_train_pred_top)
test_acc_gnb_top = accuracy_score(y_test, y_test_pred_top)

print(f"Training Accuracy (GaussianNB with Top Features): {train_acc_gnb_top:.
↪4f}")
print(f"Testing Accuracy (GaussianNB with Top Features): {test_acc_gnb_top:.
↪4f}")

# Confusion Matrix and ROC Curve Side-by-Side
fig, axes = plt.subplots(ncols=2, figsize=(12, 5))

# Confusion Matrix
cm_top = confusion_matrix(y_test, y_test_pred_top)
sns.heatmap(cm_top, annot=True, fmt='d', cmap='Blues',
            xticklabels=np.unique(y_test_pred_top),
            yticklabels=np.unique(y_test), ax=axes[0])
axes[0].set_title("GNB Top Features Confusion Matrix")
axes[0].set_xlabel("Predicted")
axes[0].set_ylabel("Actual")

# ROC Curve
y_test_binary = y_test.astype(int)

y_probs_top = gnb_top.predict_proba(X_test_top)[:, 1]
fpr_top, tpr_top, _ = roc_curve(y_test_binary, y_probs_top)

axes[1].plot(fpr_top, tpr_top, color='green', label='GNB Top Features')
```

```
axes[1].plot([0, 1], [0, 1], 'k--')
axes[1].set_title("ROC Curve - GNB Top Features")
axes[1].set_xlabel("False Positive Rate")
axes[1].set_ylabel("True Positive Rate")
axes[1].legend()

plt.tight_layout()
plt.show()

print("AUC Score (GNB Top Features):", roc_auc_score(y_test_binary,
    y_probs_top))
```

```
Top 20 features selected based on highest correlation:
 ['Population Served', 'Water Pump Age', 'Water Source Type_Lake', 'Water Source
Type_River', 'Water Source Type_Well', 'Water Quality_Contaminated', 'Pump
Type_Motorized Pump', 'Pump Type_Solar Pump']
Training Accuracy (GaussianNB with Top Features): 0.7224
Testing Accuracy (GaussianNB with Top Features): 0.7142
```



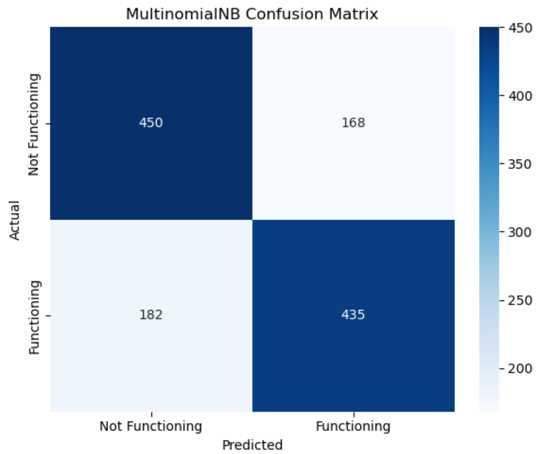

```
AUC Score (GNB Top Features): 0.7375441246662785
```

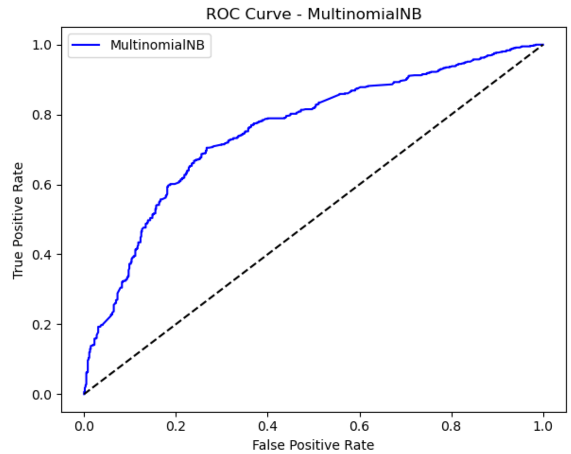
[61]:
```
# Model 3: Multinomial Naive Bayes
#----------------------------------

# Clip negatives (MultinomialNB requires non-negative features)
X_res_mnb = X_res.clip(lower=0)
X_test_mnb = X_test.clip(lower=0)

# Train MultinomialNB
mnb = MultinomialNB()
mnb.fit(X_res_mnb, y_res)
```

```python
# Predict outcomes on training and testing datasets
y_train_pred_mnb = mnb.predict(X_res_mnb)
y_test_pred_mnb = mnb.predict(X_test_mnb)

# Evaluate performance by checking training and testing accuracies
train_acc_mnb = accuracy_score(y_res, y_train_pred_mnb)
test_acc_mnb = accuracy_score(y_test, y_test_pred_mnb)

print(f"Training Accuracy (MultinomialNB): {train_acc_mnb:.4f}")
print(f"Testing Accuracy (MultinomialNB): {test_acc_mnb:.4f}")

# Overfitting check
if abs(train_acc_mnb - test_acc_mnb) < 0.05:
    print("No major overfitting detected for MultinomialNB.")
else:
    print("Possible overfitting detected for MultinomialNB.")

# Visualization: Confusion Matrix + ROC Curve Side-by-Side
#------------------------------------
fig, axes = plt.subplots(ncols=2, figsize=(12, 5))

# Confusion matrix
cm = confusion_matrix(y_test, y_test_pred_mnb)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Not Functioning', 'Functioning'],
            yticklabels=['Not Functioning', 'Functioning'],
            ax=axes[0])
axes[0].set_title("MultinomialNB Confusion Matrix")
axes[0].set_xlabel("Predicted")
axes[0].set_ylabel("Actual")

# ROC Curve
# Important: For ROC curve, make sure y_test is binary (0/1)
y_test_binary = y_test.astype(int)  # Corrected

y_probs_mnb = mnb.predict_proba(X_test_mnb)[:, 1]
fpr, tpr, _ = roc_curve(y_test_binary, y_probs_mnb)

axes[1].plot(fpr, tpr, color='blue', label='MultinomialNB')
axes[1].plot([0, 1], [0, 1], 'k--')
axes[1].set_title("ROC Curve - MultinomialNB")
axes[1].set_xlabel("False Positive Rate")
axes[1].set_ylabel("True Positive Rate")
axes[1].legend()

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

```python
print("AUC Score (MultinomialNB):", roc_auc_score(y_test_binary, y_probs_mnb))
```

```
Training Accuracy (MultinomialNB): 0.7200
Testing Accuracy (MultinomialNB): 0.7166
No major overfitting detected for MultinomialNB.
```




```
AUC Score (MultinomialNB): 0.7549238669205312
```

[62]:
```python
#Neural network
```

[63]:
```python
# 1. Regularization-Focused Neural Network

# 1. Define early stopping
early_stop = callbacks.EarlyStopping(
    monitor='val_loss',
    patience=10,
    restore_best_weights=True
)

# 2. Define the model
model_b = keras.Sequential([
    layers.Input(shape=(X_res.shape[1],)),
    layers.Dense(128, activation='relu', kernel_regularizer=regularizers.l2(0.
 ↪01)),
    layers.BatchNormalization(),
    layers.Dropout(0.3),

    layers.Dense(64, activation='relu', kernel_regularizer=regularizers.l2(0.
 ↪01)),
    layers.BatchNormalization(),
```

```python
    layers.Dense(1, activation='sigmoid')
])

# 3. Compile the model
model_b.compile(
    optimizer=keras.optimizers.Adam(learning_rate=0.001),
    loss='binary_crossentropy',
    metrics=['accuracy', keras.metrics.AUC(name='auc')]
)

# 4. Train the model
history_b = model_b.fit(
    X_res, y_res,
    validation_data=(X_test, y_test),
    epochs=100,
    batch_size=32,
    callbacks=[early_stop],
    verbose=1
)

# 5. Evaluate the model on Train set
train_loss_b, train_accuracy_b, train_auc_b = model_b.evaluate(X_res, y_res,
 ↪verbose=0)

# 6. Evaluate the model on Test set
test_loss_b, test_accuracy_b, test_auc_b = model_b.evaluate(X_test, y_test,
 ↪verbose=0)

# 7. Print Results
print("\n--- Final Evaluation: Model -Regularization-Focused ---")
print(f"Train Loss: {train_loss_b:.4f}")
print(f"Train Accuracy: {train_accuracy_b:.4f}")
print(f"Train AUC: {train_auc_b:.4f}")

print(f"\nTest Loss: {test_loss_b:.4f}")
print(f"Test Accuracy: {test_accuracy_b:.4f}")
print(f"Test AUC: {test_auc_b:.4f}")
```

```
Epoch 1/100
129/129            3s 5ms/step -
accuracy: 0.6310 - auc: 0.6779 - loss: 1.5899 - val_accuracy: 0.5822 - val_auc:
0.7757 - val_loss: 1.3093
Epoch 2/100
129/129            0s 3ms/step -
accuracy: 0.7198 - auc: 0.7761 - loss: 1.1740 - val_accuracy: 0.6146 - val_auc:
0.7939 - val_loss: 1.0979
Epoch 3/100
```

```
129/129              0s 3ms/step -
accuracy: 0.7253 - auc: 0.7777 - loss: 1.0004 - val_accuracy: 0.6850 - val_auc:
0.7974 - val_loss: 0.9391
Epoch 4/100
129/129              0s 3ms/step -
accuracy: 0.7323 - auc: 0.7915 - loss: 0.8782 - val_accuracy: 0.7255 - val_auc:
0.8066 - val_loss: 0.8106
Epoch 5/100
129/129              0s 2ms/step -
accuracy: 0.7271 - auc: 0.7919 - loss: 0.7973 - val_accuracy: 0.7457 - val_auc:
0.8060 - val_loss: 0.7385
Epoch 6/100
129/129              0s 2ms/step -
accuracy: 0.7331 - auc: 0.7943 - loss: 0.7411 - val_accuracy: 0.7433 - val_auc:
0.8142 - val_loss: 0.6931
Epoch 7/100
129/129              0s 3ms/step -
accuracy: 0.7358 - auc: 0.7973 - loss: 0.6941 - val_accuracy: 0.7385 - val_auc:
0.8112 - val_loss: 0.6629
Epoch 8/100
129/129              0s 3ms/step -
accuracy: 0.7500 - auc: 0.8063 - loss: 0.6537 - val_accuracy: 0.7449 - val_auc:
0.8127 - val_loss: 0.6320
Epoch 9/100
129/129              0s 3ms/step -
accuracy: 0.7299 - auc: 0.7945 - loss: 0.6457 - val_accuracy: 0.7425 - val_auc:
0.8038 - val_loss: 0.6206
Epoch 10/100
129/129              0s 3ms/step -
accuracy: 0.7394 - auc: 0.7962 - loss: 0.6277 - val_accuracy: 0.7433 - val_auc:
0.8138 - val_loss: 0.5990
Epoch 11/100
129/129              0s 3ms/step -
accuracy: 0.7412 - auc: 0.8024 - loss: 0.6048 - val_accuracy: 0.7490 - val_auc:
0.8127 - val_loss: 0.5856
Epoch 12/100
129/129              0s 2ms/step -
accuracy: 0.7391 - auc: 0.8066 - loss: 0.5899 - val_accuracy: 0.7296 - val_auc:
0.7994 - val_loss: 0.5887
Epoch 13/100
129/129              0s 2ms/step -
accuracy: 0.7263 - auc: 0.7908 - loss: 0.5962 - val_accuracy: 0.7336 - val_auc:
0.7906 - val_loss: 0.5929
Epoch 14/100
129/129              0s 3ms/step -
accuracy: 0.7412 - auc: 0.8009 - loss: 0.5875 - val_accuracy: 0.7393 - val_auc:
0.8024 - val_loss: 0.5699
Epoch 15/100
```

```
129/129              0s 2ms/step -
accuracy: 0.7311 - auc: 0.7889 - loss: 0.5922 - val_accuracy: 0.7474 - val_auc:
0.8105 - val_loss: 0.5574
Epoch 16/100
129/129              0s 3ms/step -
accuracy: 0.7324 - auc: 0.7920 - loss: 0.5833 - val_accuracy: 0.7457 - val_auc:
0.8096 - val_loss: 0.5575
Epoch 17/100
129/129              0s 3ms/step -
accuracy: 0.7354 - auc: 0.7870 - loss: 0.5846 - val_accuracy: 0.7514 - val_auc:
0.8046 - val_loss: 0.5567
Epoch 18/100
129/129              0s 3ms/step -
accuracy: 0.7314 - auc: 0.7932 - loss: 0.5763 - val_accuracy: 0.7457 - val_auc:
0.8010 - val_loss: 0.5580
Epoch 19/100
129/129              0s 3ms/step -
accuracy: 0.7467 - auc: 0.8107 - loss: 0.5514 - val_accuracy: 0.7328 - val_auc:
0.8040 - val_loss: 0.5550
Epoch 20/100
129/129              0s 3ms/step -
accuracy: 0.7416 - auc: 0.8067 - loss: 0.5530 - val_accuracy: 0.7360 - val_auc:
0.7963 - val_loss: 0.5615
Epoch 21/100
129/129              0s 3ms/step -
accuracy: 0.7468 - auc: 0.8029 - loss: 0.5574 - val_accuracy: 0.7263 - val_auc:
0.7793 - val_loss: 0.5798
Epoch 22/100
129/129              0s 3ms/step -
accuracy: 0.7410 - auc: 0.8054 - loss: 0.5551 - val_accuracy: 0.7393 - val_auc:
0.7982 - val_loss: 0.5566
Epoch 23/100
129/129              0s 2ms/step -
accuracy: 0.7392 - auc: 0.8028 - loss: 0.5560 - val_accuracy: 0.7385 - val_auc:
0.8072 - val_loss: 0.5470
Epoch 24/100
129/129              0s 3ms/step -
accuracy: 0.7350 - auc: 0.8017 - loss: 0.5566 - val_accuracy: 0.7417 - val_auc:
0.7999 - val_loss: 0.5522
Epoch 25/100
129/129              0s 4ms/step -
accuracy: 0.7513 - auc: 0.8119 - loss: 0.5427 - val_accuracy: 0.7474 - val_auc:
0.8079 - val_loss: 0.5430
Epoch 26/100
129/129              0s 3ms/step -
accuracy: 0.7439 - auc: 0.8059 - loss: 0.5474 - val_accuracy: 0.7425 - val_auc:
0.8051 - val_loss: 0.5440
Epoch 27/100
```

```
129/129            0s 3ms/step -
accuracy: 0.7550 - auc: 0.8141 - loss: 0.5363 - val_accuracy: 0.7506 - val_auc:
0.8041 - val_loss: 0.5430
Epoch 28/100
129/129            0s 3ms/step -
accuracy: 0.7357 - auc: 0.8010 - loss: 0.5532 - val_accuracy: 0.7360 - val_auc:
0.7968 - val_loss: 0.5537
Epoch 29/100
129/129            0s 3ms/step -
accuracy: 0.7536 - auc: 0.8147 - loss: 0.5407 - val_accuracy: 0.7441 - val_auc:
0.8034 - val_loss: 0.5434
Epoch 30/100
129/129            0s 3ms/step -
accuracy: 0.7498 - auc: 0.8092 - loss: 0.5418 - val_accuracy: 0.7441 - val_auc:
0.8060 - val_loss: 0.5408
Epoch 31/100
129/129            0s 3ms/step -
accuracy: 0.7455 - auc: 0.8116 - loss: 0.5430 - val_accuracy: 0.7425 - val_auc:
0.8048 - val_loss: 0.5413
Epoch 32/100
129/129            0s 3ms/step -
accuracy: 0.7374 - auc: 0.7958 - loss: 0.5563 - val_accuracy: 0.7482 - val_auc:
0.8013 - val_loss: 0.5428
Epoch 33/100
129/129            0s 3ms/step -
accuracy: 0.7464 - auc: 0.7984 - loss: 0.5483 - val_accuracy: 0.7433 - val_auc:
0.7968 - val_loss: 0.5486
Epoch 34/100
129/129            0s 3ms/step -
accuracy: 0.7348 - auc: 0.7891 - loss: 0.5613 - val_accuracy: 0.7506 - val_auc:
0.8039 - val_loss: 0.5380
Epoch 35/100
129/129            0s 2ms/step -
accuracy: 0.7491 - auc: 0.8026 - loss: 0.5469 - val_accuracy: 0.7174 - val_auc:
0.7951 - val_loss: 0.5550
Epoch 36/100
129/129            0s 3ms/step -
accuracy: 0.7572 - auc: 0.8165 - loss: 0.5329 - val_accuracy: 0.7401 - val_auc:
0.8017 - val_loss: 0.5439
Epoch 37/100
129/129            0s 2ms/step -
accuracy: 0.7472 - auc: 0.8058 - loss: 0.5418 - val_accuracy: 0.7514 - val_auc:
0.8045 - val_loss: 0.5416
Epoch 38/100
129/129            0s 3ms/step -
accuracy: 0.7336 - auc: 0.7989 - loss: 0.5501 - val_accuracy: 0.7449 - val_auc:
0.8051 - val_loss: 0.5409
Epoch 39/100
```

```
129/129            0s 3ms/step -
accuracy: 0.7346 - auc: 0.7925 - loss: 0.5576 - val_accuracy: 0.7482 - val_auc:
0.8038 - val_loss: 0.5398
Epoch 40/100
129/129            0s 3ms/step -
accuracy: 0.7526 - auc: 0.8156 - loss: 0.5306 - val_accuracy: 0.7466 - val_auc:
0.8025 - val_loss: 0.5409
Epoch 41/100
129/129            0s 3ms/step -
accuracy: 0.7492 - auc: 0.8099 - loss: 0.5359 - val_accuracy: 0.7466 - val_auc:
0.7983 - val_loss: 0.5447
Epoch 42/100
129/129            0s 3ms/step -
accuracy: 0.7334 - auc: 0.7971 - loss: 0.5507 - val_accuracy: 0.7425 - val_auc:
0.8042 - val_loss: 0.5371
Epoch 43/100
129/129            0s 3ms/step -
accuracy: 0.7399 - auc: 0.7976 - loss: 0.5518 - val_accuracy: 0.7449 - val_auc:
0.8014 - val_loss: 0.5396
Epoch 44/100
129/129            0s 3ms/step -
accuracy: 0.7376 - auc: 0.8043 - loss: 0.5445 - val_accuracy: 0.7425 - val_auc:
0.8008 - val_loss: 0.5405
Epoch 45/100
129/129            0s 3ms/step -
accuracy: 0.7260 - auc: 0.7861 - loss: 0.5624 - val_accuracy: 0.7417 - val_auc:
0.8022 - val_loss: 0.5388
Epoch 46/100
129/129            0s 2ms/step -
accuracy: 0.7290 - auc: 0.7964 - loss: 0.5472 - val_accuracy: 0.7506 - val_auc:
0.8019 - val_loss: 0.5407
Epoch 47/100
129/129            0s 3ms/step -
accuracy: 0.7361 - auc: 0.8028 - loss: 0.5439 - val_accuracy: 0.7474 - val_auc:
0.7999 - val_loss: 0.5405
Epoch 48/100
129/129            0s 3ms/step -
accuracy: 0.7420 - auc: 0.8003 - loss: 0.5460 - val_accuracy: 0.7474 - val_auc:
0.8048 - val_loss: 0.5370
Epoch 49/100
129/129            0s 2ms/step -
accuracy: 0.7322 - auc: 0.7845 - loss: 0.5585 - val_accuracy: 0.7466 - val_auc:
0.8009 - val_loss: 0.5427
Epoch 50/100
129/129            0s 2ms/step -
accuracy: 0.7288 - auc: 0.7865 - loss: 0.5591 - val_accuracy: 0.7344 - val_auc:
0.8012 - val_loss: 0.5404
Epoch 51/100
```

```
129/129              0s 2ms/step -
accuracy: 0.7360 - auc: 0.7916 - loss: 0.5543 - val_accuracy: 0.7482 - val_auc:
0.8003 - val_loss: 0.5409
Epoch 52/100
129/129              0s 3ms/step -
accuracy: 0.7451 - auc: 0.8038 - loss: 0.5419 - val_accuracy: 0.7385 - val_auc:
0.8003 - val_loss: 0.5425
Epoch 53/100
129/129              0s 4ms/step -
accuracy: 0.7350 - auc: 0.8016 - loss: 0.5434 - val_accuracy: 0.7385 - val_auc:
0.7975 - val_loss: 0.5446
Epoch 54/100
129/129              0s 3ms/step -
accuracy: 0.7485 - auc: 0.8079 - loss: 0.5367 - val_accuracy: 0.7441 - val_auc:
0.7981 - val_loss: 0.5408
Epoch 55/100
129/129              0s 3ms/step -
accuracy: 0.7440 - auc: 0.8017 - loss: 0.5424 - val_accuracy: 0.7368 - val_auc:
0.8035 - val_loss: 0.5390
Epoch 56/100
129/129              0s 3ms/step -
accuracy: 0.7529 - auc: 0.8096 - loss: 0.5352 - val_accuracy: 0.7166 - val_auc:
0.7921 - val_loss: 0.5518
Epoch 57/100
129/129              0s 3ms/step -
accuracy: 0.7334 - auc: 0.8018 - loss: 0.5433 - val_accuracy: 0.7441 - val_auc:
0.8035 - val_loss: 0.5366
Epoch 58/100
129/129              0s 3ms/step -
accuracy: 0.7357 - auc: 0.7970 - loss: 0.5480 - val_accuracy: 0.7360 - val_auc:
0.8004 - val_loss: 0.5376
Epoch 59/100
129/129              0s 3ms/step -
accuracy: 0.7442 - auc: 0.8005 - loss: 0.5429 - val_accuracy: 0.7457 - val_auc:
0.7998 - val_loss: 0.5387
Epoch 60/100
129/129              0s 3ms/step -
accuracy: 0.7396 - auc: 0.7984 - loss: 0.5455 - val_accuracy: 0.7449 - val_auc:
0.7984 - val_loss: 0.5384
Epoch 61/100
129/129              0s 3ms/step -
accuracy: 0.7380 - auc: 0.7999 - loss: 0.5442 - val_accuracy: 0.7344 - val_auc:
0.7880 - val_loss: 0.5494
Epoch 62/100
129/129              0s 3ms/step -
accuracy: 0.7309 - auc: 0.8025 - loss: 0.5444 - val_accuracy: 0.7474 - val_auc:
0.8058 - val_loss: 0.5338
Epoch 63/100
```

```
129/129            0s 3ms/step -
accuracy: 0.7465 - auc: 0.8035 - loss: 0.5382 - val_accuracy: 0.7425 - val_auc:
0.7993 - val_loss: 0.5409
Epoch 64/100
129/129            0s 3ms/step -
accuracy: 0.7412 - auc: 0.7998 - loss: 0.5472 - val_accuracy: 0.7433 - val_auc:
0.8030 - val_loss: 0.5364
Epoch 65/100
129/129            0s 3ms/step -
accuracy: 0.7364 - auc: 0.7929 - loss: 0.5527 - val_accuracy: 0.7449 - val_auc:
0.8019 - val_loss: 0.5364
Epoch 66/100
129/129            0s 3ms/step -
accuracy: 0.7429 - auc: 0.8032 - loss: 0.5389 - val_accuracy: 0.7433 - val_auc:
0.8018 - val_loss: 0.5365
Epoch 67/100
129/129            0s 3ms/step -
accuracy: 0.7431 - auc: 0.8033 - loss: 0.5402 - val_accuracy: 0.7425 - val_auc:
0.8012 - val_loss: 0.5391
Epoch 68/100
129/129            0s 3ms/step -
accuracy: 0.7359 - auc: 0.8026 - loss: 0.5396 - val_accuracy: 0.7490 - val_auc:
0.7998 - val_loss: 0.5356
Epoch 69/100
129/129            0s 3ms/step -
accuracy: 0.7440 - auc: 0.8079 - loss: 0.5344 - val_accuracy: 0.7352 - val_auc:
0.7990 - val_loss: 0.5407
Epoch 70/100
129/129            0s 3ms/step -
accuracy: 0.7405 - auc: 0.7997 - loss: 0.5439 - val_accuracy: 0.7498 - val_auc:
0.8034 - val_loss: 0.5348
Epoch 71/100
129/129            0s 3ms/step -
accuracy: 0.7508 - auc: 0.8113 - loss: 0.5312 - val_accuracy: 0.7457 - val_auc:
0.8031 - val_loss: 0.5366
Epoch 72/100
129/129            0s 3ms/step -
accuracy: 0.7397 - auc: 0.8012 - loss: 0.5421 - val_accuracy: 0.7474 - val_auc:
0.8009 - val_loss: 0.5374

--- Final Evaluation: Model -Regularization-Focused ---
Train Loss: 0.5279
Train Accuracy: 0.7482
Train AUC: 0.8147

Test Loss: 0.5338
Test Accuracy: 0.7474
Test AUC: 0.8058
```

[64]:

```python
# 2. High-Dropout Stability Neural Network

# 1. Define early stopping to prevent overfitting
early_stop = callbacks.EarlyStopping(
    monitor='val_loss',
    patience=15,  # Increased patience for better stability
    restore_best_weights=True
)

# 2. Define the model with adjusted architecture
model = keras.Sequential([
    layers.Input(shape=(X_res.shape[1],)),

    # First hidden layer with L2 regularization and increased dropout
    layers.Dense(64, activation='relu', kernel_regularizer=regularizers.l2(0.
 ↪005)),
    layers.BatchNormalization(),
    layers.Dropout(0.5),  # Increased dropout to reduce overfitting

    # Second hidden layer with L2 regularization and increased dropout
    layers.Dense(32, activation='relu', kernel_regularizer=regularizers.l2(0.
 ↪005)),
    layers.BatchNormalization(),
    layers.Dropout(0.4),  # Increased dropout to reduce overfitting

    # Output layer
    layers.Dense(1, activation='sigmoid')  # Single output neuron for binary␣
 ↪classification
])

# 3. Compile the model with a slightly reduced learning rate
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=0.0005),  # Slightly reduced␣
 ↪learning rate
    loss='binary_crossentropy',
    metrics=['accuracy', keras.metrics.AUC(name='auc')]
)

# 4. Train the model
history = model.fit(
    X_res, y_res,
    validation_data=(X_test, y_test),
    epochs=100,
    batch_size=32,
    callbacks=[early_stop],
    verbose=1
)
```

```python
# 5. Evaluate the model on Train set
train_loss, train_accuracy, train_auc = model.evaluate(X_res, y_res, verbose=0)

# 6. Evaluate the model on Test set
test_loss, test_accuracy, test_auc = model.evaluate(X_test, y_test, verbose=0)

# 7. Print Results
print("\n--- Final Evaluation -High-Dropout Stability ---")
print(f"Train Loss: {train_loss:.4f}")
print(f"Train Accuracy: {train_accuracy:.4f}")
print(f"Train AUC: {train_auc:.4f}")

print(f"\nTest Loss: {test_loss:.4f}")
print(f"Test Accuracy: {test_accuracy:.4f}")
print(f"Test AUC: {test_auc:.4f}")
```

```
Epoch 1/100
129/129            3s 5ms/step -
accuracy: 0.5315 - auc: 0.5482 - loss: 1.1245 - val_accuracy: 0.6939 - val_auc:
0.7310 - val_loss: 0.8994
Epoch 2/100
129/129            0s 3ms/step -
accuracy: 0.6278 - auc: 0.6757 - loss: 0.9631 - val_accuracy: 0.7101 - val_auc:
0.7555 - val_loss: 0.8441
Epoch 3/100
129/129            0s 3ms/step -
accuracy: 0.6625 - auc: 0.7127 - loss: 0.8988 - val_accuracy: 0.7166 - val_auc:
0.7649 - val_loss: 0.8074
Epoch 4/100
129/129            0s 3ms/step -
accuracy: 0.6626 - auc: 0.7093 - loss: 0.8804 - val_accuracy: 0.7296 - val_auc:
0.7718 - val_loss: 0.7825
Epoch 5/100
129/129            0s 3ms/step -
accuracy: 0.6826 - auc: 0.7382 - loss: 0.8315 - val_accuracy: 0.7271 - val_auc:
0.7751 - val_loss: 0.7643
Epoch 6/100
129/129            0s 3ms/step -
accuracy: 0.6876 - auc: 0.7400 - loss: 0.8141 - val_accuracy: 0.7304 - val_auc:
0.7767 - val_loss: 0.7485
Epoch 7/100
129/129            0s 3ms/step -
accuracy: 0.7061 - auc: 0.7460 - loss: 0.7923 - val_accuracy: 0.7344 - val_auc:
0.7807 - val_loss: 0.7339
Epoch 8/100
129/129            0s 3ms/step -
accuracy: 0.6943 - auc: 0.7544 - loss: 0.7692 - val_accuracy: 0.7344 - val_auc:
```

```
0.7821 - val_loss: 0.7215
Epoch 9/100
129/129            0s 3ms/step -
accuracy: 0.7002 - auc: 0.7537 - loss: 0.7630 - val_accuracy: 0.7377 - val_auc:
0.7839 - val_loss: 0.7081
Epoch 10/100
129/129            0s 3ms/step -
accuracy: 0.6967 - auc: 0.7460 - loss: 0.7597 - val_accuracy: 0.7360 - val_auc:
0.7843 - val_loss: 0.6978
Epoch 11/100
129/129            0s 3ms/step -
accuracy: 0.7062 - auc: 0.7596 - loss: 0.7291 - val_accuracy: 0.7344 - val_auc:
0.7866 - val_loss: 0.6865
Epoch 12/100
129/129            0s 3ms/step -
accuracy: 0.7189 - auc: 0.7762 - loss: 0.7071 - val_accuracy: 0.7385 - val_auc:
0.7888 - val_loss: 0.6758
Epoch 13/100
129/129            0s 2ms/step -
accuracy: 0.7301 - auc: 0.7807 - loss: 0.6916 - val_accuracy: 0.7409 - val_auc:
0.7902 - val_loss: 0.6659
Epoch 14/100
129/129            0s 2ms/step -
accuracy: 0.7278 - auc: 0.7836 - loss: 0.6794 - val_accuracy: 0.7433 - val_auc:
0.7932 - val_loss: 0.6555
Epoch 15/100
129/129            0s 2ms/step -
accuracy: 0.7216 - auc: 0.7761 - loss: 0.6789 - val_accuracy: 0.7433 - val_auc:
0.7929 - val_loss: 0.6481
Epoch 16/100
129/129            0s 2ms/step -
accuracy: 0.7332 - auc: 0.7879 - loss: 0.6595 - val_accuracy: 0.7433 - val_auc:
0.7955 - val_loss: 0.6390
Epoch 17/100
129/129            0s 2ms/step -
accuracy: 0.7261 - auc: 0.7783 - loss: 0.6616 - val_accuracy: 0.7417 - val_auc:
0.7968 - val_loss: 0.6317
Epoch 18/100
129/129            0s 2ms/step -
accuracy: 0.7239 - auc: 0.7809 - loss: 0.6547 - val_accuracy: 0.7449 - val_auc:
0.7949 - val_loss: 0.6251
Epoch 19/100
129/129            0s 2ms/step -
accuracy: 0.7355 - auc: 0.7887 - loss: 0.6393 - val_accuracy: 0.7457 - val_auc:
0.7953 - val_loss: 0.6188
Epoch 20/100
129/129            0s 2ms/step -
accuracy: 0.7262 - auc: 0.7843 - loss: 0.6384 - val_accuracy: 0.7433 - val_auc:
```

```
0.7972 - val_loss: 0.6118
Epoch 21/100
129/129             0s 2ms/step -
accuracy: 0.7322 - auc: 0.7934 - loss: 0.6199 - val_accuracy: 0.7425 - val_auc:
0.7989 - val_loss: 0.6062
Epoch 22/100
129/129             0s 3ms/step -
accuracy: 0.7277 - auc: 0.7957 - loss: 0.6134 - val_accuracy: 0.7417 - val_auc:
0.7984 - val_loss: 0.6003
Epoch 23/100
129/129             0s 2ms/step -
accuracy: 0.7435 - auc: 0.7936 - loss: 0.6100 - val_accuracy: 0.7417 - val_auc:
0.7987 - val_loss: 0.5947
Epoch 24/100
129/129             0s 3ms/step -
accuracy: 0.7425 - auc: 0.7968 - loss: 0.6025 - val_accuracy: 0.7417 - val_auc:
0.7998 - val_loss: 0.5897
Epoch 25/100
129/129             0s 3ms/step -
accuracy: 0.7310 - auc: 0.7953 - loss: 0.5988 - val_accuracy: 0.7385 - val_auc:
0.8001 - val_loss: 0.5851
Epoch 26/100
129/129             1s 4ms/step -
accuracy: 0.7250 - auc: 0.7841 - loss: 0.6096 - val_accuracy: 0.7449 - val_auc:
0.7995 - val_loss: 0.5816
Epoch 27/100
129/129             0s 3ms/step -
accuracy: 0.7332 - auc: 0.7939 - loss: 0.5980 - val_accuracy: 0.7433 - val_auc:
0.8003 - val_loss: 0.5764
Epoch 28/100
129/129             0s 3ms/step -
accuracy: 0.7446 - auc: 0.8025 - loss: 0.5808 - val_accuracy: 0.7441 - val_auc:
0.8010 - val_loss: 0.5735
Epoch 29/100
129/129             0s 3ms/step -
accuracy: 0.7318 - auc: 0.7878 - loss: 0.5963 - val_accuracy: 0.7466 - val_auc:
0.8005 - val_loss: 0.5711
Epoch 30/100
129/129             0s 3ms/step -
accuracy: 0.7345 - auc: 0.7938 - loss: 0.5838 - val_accuracy: 0.7425 - val_auc:
0.7972 - val_loss: 0.5715
Epoch 31/100
129/129             0s 3ms/step -
accuracy: 0.7241 - auc: 0.7966 - loss: 0.5778 - val_accuracy: 0.7474 - val_auc:
0.8014 - val_loss: 0.5653
Epoch 32/100
129/129             0s 3ms/step -
accuracy: 0.7381 - auc: 0.7900 - loss: 0.5830 - val_accuracy: 0.7466 - val_auc:
```

```
0.8000 - val_loss: 0.5622
Epoch 33/100
129/129            0s 3ms/step -
accuracy: 0.7343 - auc: 0.7989 - loss: 0.5696 - val_accuracy: 0.7433 - val_auc:
0.8013 - val_loss: 0.5590
Epoch 34/100
129/129            0s 3ms/step -
accuracy: 0.7459 - auc: 0.7996 - loss: 0.5681 - val_accuracy: 0.7457 - val_auc:
0.7972 - val_loss: 0.5608
Epoch 35/100
129/129            0s 3ms/step -
accuracy: 0.7332 - auc: 0.7902 - loss: 0.5809 - val_accuracy: 0.7449 - val_auc:
0.8021 - val_loss: 0.5563
Epoch 36/100
129/129            0s 2ms/step -
accuracy: 0.7370 - auc: 0.8022 - loss: 0.5620 - val_accuracy: 0.7490 - val_auc:
0.8011 - val_loss: 0.5535
Epoch 37/100
129/129            0s 2ms/step -
accuracy: 0.7423 - auc: 0.7956 - loss: 0.5704 - val_accuracy: 0.7466 - val_auc:
0.8002 - val_loss: 0.5538
Epoch 38/100
129/129            0s 3ms/step -
accuracy: 0.7351 - auc: 0.7983 - loss: 0.5651 - val_accuracy: 0.7441 - val_auc:
0.8015 - val_loss: 0.5526
Epoch 39/100
129/129            0s 3ms/step -
accuracy: 0.7363 - auc: 0.7953 - loss: 0.5670 - val_accuracy: 0.7401 - val_auc:
0.8010 - val_loss: 0.5514
Epoch 40/100
129/129            0s 3ms/step -
accuracy: 0.7385 - auc: 0.8014 - loss: 0.5565 - val_accuracy: 0.7409 - val_auc:
0.7997 - val_loss: 0.5509
Epoch 41/100
129/129            0s 3ms/step -
accuracy: 0.7418 - auc: 0.8025 - loss: 0.5539 - val_accuracy: 0.7530 - val_auc:
0.8020 - val_loss: 0.5481
Epoch 42/100
129/129            0s 2ms/step -
accuracy: 0.7434 - auc: 0.7922 - loss: 0.5653 - val_accuracy: 0.7538 - val_auc:
0.8000 - val_loss: 0.5477
Epoch 43/100
129/129            0s 3ms/step -
accuracy: 0.7414 - auc: 0.8029 - loss: 0.5522 - val_accuracy: 0.7441 - val_auc:
0.8020 - val_loss: 0.5463
Epoch 44/100
129/129            0s 2ms/step -
accuracy: 0.7374 - auc: 0.7961 - loss: 0.5564 - val_accuracy: 0.7498 - val_auc:
```

```
0.8009 - val_loss: 0.5428
Epoch 45/100
129/129            0s 2ms/step -
accuracy: 0.7440 - auc: 0.8020 - loss: 0.5511 - val_accuracy: 0.7441 - val_auc:
0.8028 - val_loss: 0.5419
Epoch 46/100
129/129            0s 2ms/step -
accuracy: 0.7498 - auc: 0.7989 - loss: 0.5553 - val_accuracy: 0.7457 - val_auc:
0.8015 - val_loss: 0.5435
Epoch 47/100
129/129            0s 2ms/step -
accuracy: 0.7426 - auc: 0.8029 - loss: 0.5536 - val_accuracy: 0.7474 - val_auc:
0.8007 - val_loss: 0.5423
Epoch 48/100
129/129            0s 2ms/step -
accuracy: 0.7476 - auc: 0.8057 - loss: 0.5465 - val_accuracy: 0.7441 - val_auc:
0.8024 - val_loss: 0.5424
Epoch 49/100
129/129            0s 2ms/step -
accuracy: 0.7327 - auc: 0.7870 - loss: 0.5675 - val_accuracy: 0.7457 - val_auc:
0.8032 - val_loss: 0.5410
Epoch 50/100
129/129            0s 2ms/step -
accuracy: 0.7504 - auc: 0.8022 - loss: 0.5516 - val_accuracy: 0.7433 - val_auc:
0.8040 - val_loss: 0.5396
Epoch 51/100
129/129            0s 3ms/step -
accuracy: 0.7370 - auc: 0.8030 - loss: 0.5501 - val_accuracy: 0.7457 - val_auc:
0.8032 - val_loss: 0.5392
Epoch 52/100
129/129            0s 2ms/step -
accuracy: 0.7237 - auc: 0.7910 - loss: 0.5603 - val_accuracy: 0.7506 - val_auc:
0.8007 - val_loss: 0.5411
Epoch 53/100
129/129            0s 2ms/step -
accuracy: 0.7385 - auc: 0.8000 - loss: 0.5501 - val_accuracy: 0.7433 - val_auc:
0.8018 - val_loss: 0.5398
Epoch 54/100
129/129            0s 2ms/step -
accuracy: 0.7388 - auc: 0.8002 - loss: 0.5503 - val_accuracy: 0.7474 - val_auc:
0.8036 - val_loss: 0.5385
Epoch 55/100
129/129            0s 3ms/step -
accuracy: 0.7334 - auc: 0.7950 - loss: 0.5543 - val_accuracy: 0.7433 - val_auc:
0.8059 - val_loss: 0.5377
Epoch 56/100
129/129            0s 2ms/step -
accuracy: 0.7366 - auc: 0.7951 - loss: 0.5537 - val_accuracy: 0.7514 - val_auc:
```

```
0.8037 - val_loss: 0.5365
Epoch 57/100
129/129            0s 3ms/step -
accuracy: 0.7451 - auc: 0.8043 - loss: 0.5446 - val_accuracy: 0.7482 - val_auc:
0.8045 - val_loss: 0.5379
Epoch 58/100
129/129            0s 2ms/step -
accuracy: 0.7536 - auc: 0.8019 - loss: 0.5458 - val_accuracy: 0.7441 - val_auc:
0.8030 - val_loss: 0.5389
Epoch 59/100
129/129            0s 2ms/step -
accuracy: 0.7440 - auc: 0.8017 - loss: 0.5465 - val_accuracy: 0.7466 - val_auc:
0.8022 - val_loss: 0.5385
Epoch 60/100
129/129            0s 2ms/step -
accuracy: 0.7461 - auc: 0.8055 - loss: 0.5432 - val_accuracy: 0.7474 - val_auc:
0.8020 - val_loss: 0.5384
Epoch 61/100
129/129            0s 2ms/step -
accuracy: 0.7478 - auc: 0.7955 - loss: 0.5520 - val_accuracy: 0.7498 - val_auc:
0.8025 - val_loss: 0.5365
Epoch 62/100
129/129            0s 2ms/step -
accuracy: 0.7403 - auc: 0.8049 - loss: 0.5412 - val_accuracy: 0.7409 - val_auc:
0.8028 - val_loss: 0.5375
Epoch 63/100
129/129            0s 3ms/step -
accuracy: 0.7502 - auc: 0.8044 - loss: 0.5425 - val_accuracy: 0.7360 - val_auc:
0.8025 - val_loss: 0.5388
Epoch 64/100
129/129            0s 3ms/step -
accuracy: 0.7433 - auc: 0.8024 - loss: 0.5457 - val_accuracy: 0.7474 - val_auc:
0.8012 - val_loss: 0.5380
Epoch 65/100
129/129            0s 2ms/step -
accuracy: 0.7434 - auc: 0.7924 - loss: 0.5558 - val_accuracy: 0.7466 - val_auc:
0.8023 - val_loss: 0.5373
Epoch 66/100
129/129            0s 2ms/step -
accuracy: 0.7452 - auc: 0.7996 - loss: 0.5489 - val_accuracy: 0.7498 - val_auc:
0.8036 - val_loss: 0.5358
Epoch 67/100
129/129            0s 2ms/step -
accuracy: 0.7414 - auc: 0.8007 - loss: 0.5445 - val_accuracy: 0.7466 - val_auc:
0.8043 - val_loss: 0.5338
Epoch 68/100
129/129            0s 3ms/step -
accuracy: 0.7447 - auc: 0.8056 - loss: 0.5431 - val_accuracy: 0.7498 - val_auc:
```

```
0.8046 - val_loss: 0.5339
Epoch 69/100
129/129             0s 3ms/step -
accuracy: 0.7257 - auc: 0.7897 - loss: 0.5606 - val_accuracy: 0.7417 - val_auc:
0.8062 - val_loss: 0.5336
Epoch 70/100
129/129             0s 3ms/step -
accuracy: 0.7463 - auc: 0.8075 - loss: 0.5392 - val_accuracy: 0.7393 - val_auc:
0.8046 - val_loss: 0.5358
Epoch 71/100
129/129             0s 3ms/step -
accuracy: 0.7475 - auc: 0.8099 - loss: 0.5392 - val_accuracy: 0.7417 - val_auc:
0.8033 - val_loss: 0.5359
Epoch 72/100
129/129             0s 3ms/step -
accuracy: 0.7410 - auc: 0.8060 - loss: 0.5431 - val_accuracy: 0.7530 - val_auc:
0.8039 - val_loss: 0.5353
Epoch 73/100
129/129             1s 5ms/step -
accuracy: 0.7375 - auc: 0.7962 - loss: 0.5507 - val_accuracy: 0.7506 - val_auc:
0.8033 - val_loss: 0.5345
Epoch 74/100
129/129             1s 5ms/step -
accuracy: 0.7512 - auc: 0.8006 - loss: 0.5453 - val_accuracy: 0.7449 - val_auc:
0.8034 - val_loss: 0.5343
Epoch 75/100
129/129             0s 3ms/step -
accuracy: 0.7492 - auc: 0.8092 - loss: 0.5373 - val_accuracy: 0.7425 - val_auc:
0.8056 - val_loss: 0.5329
Epoch 76/100
129/129             0s 3ms/step -
accuracy: 0.7409 - auc: 0.8032 - loss: 0.5440 - val_accuracy: 0.7490 - val_auc:
0.8041 - val_loss: 0.5342
Epoch 77/100
129/129             0s 3ms/step -
accuracy: 0.7492 - auc: 0.8030 - loss: 0.5417 - val_accuracy: 0.7498 - val_auc:
0.8035 - val_loss: 0.5351
Epoch 78/100
129/129             0s 3ms/step -
accuracy: 0.7326 - auc: 0.7904 - loss: 0.5542 - val_accuracy: 0.7498 - val_auc:
0.8046 - val_loss: 0.5339
Epoch 79/100
129/129             0s 3ms/step -
accuracy: 0.7454 - auc: 0.7999 - loss: 0.5492 - val_accuracy: 0.7466 - val_auc:
0.8048 - val_loss: 0.5346
Epoch 80/100
129/129             0s 3ms/step -
accuracy: 0.7376 - auc: 0.7989 - loss: 0.5476 - val_accuracy: 0.7490 - val_auc:
```

```
0.8074 - val_loss: 0.5332
Epoch 81/100
129/129              0s 3ms/step -
accuracy: 0.7421 - auc: 0.7968 - loss: 0.5494 - val_accuracy: 0.7449 - val_auc:
0.8021 - val_loss: 0.5359
Epoch 82/100
129/129              0s 3ms/step -
accuracy: 0.7403 - auc: 0.8000 - loss: 0.5446 - val_accuracy: 0.7530 - val_auc:
0.8030 - val_loss: 0.5329
Epoch 83/100
129/129              0s 3ms/step -
accuracy: 0.7373 - auc: 0.7967 - loss: 0.5507 - val_accuracy: 0.7490 - val_auc:
0.8045 - val_loss: 0.5326
Epoch 84/100
129/129              0s 3ms/step -
accuracy: 0.7517 - auc: 0.8114 - loss: 0.5310 - val_accuracy: 0.7498 - val_auc:
0.8047 - val_loss: 0.5335
Epoch 85/100
129/129              0s 2ms/step -
accuracy: 0.7339 - auc: 0.7912 - loss: 0.5524 - val_accuracy: 0.7506 - val_auc:
0.8033 - val_loss: 0.5341
Epoch 86/100
129/129              0s 3ms/step -
accuracy: 0.7443 - auc: 0.8045 - loss: 0.5427 - val_accuracy: 0.7417 - val_auc:
0.8038 - val_loss: 0.5347
Epoch 87/100
129/129              0s 3ms/step -
accuracy: 0.7386 - auc: 0.7966 - loss: 0.5497 - val_accuracy: 0.7490 - val_auc:
0.8044 - val_loss: 0.5319
Epoch 88/100
129/129              0s 3ms/step -
accuracy: 0.7415 - auc: 0.8087 - loss: 0.5362 - val_accuracy: 0.7514 - val_auc:
0.8045 - val_loss: 0.5327
Epoch 89/100
129/129              0s 3ms/step -
accuracy: 0.7510 - auc: 0.8095 - loss: 0.5336 - val_accuracy: 0.7457 - val_auc:
0.8058 - val_loss: 0.5311
Epoch 90/100
129/129              0s 3ms/step -
accuracy: 0.7391 - auc: 0.8014 - loss: 0.5439 - val_accuracy: 0.7482 - val_auc:
0.8071 - val_loss: 0.5303
Epoch 91/100
129/129              0s 3ms/step -
accuracy: 0.7513 - auc: 0.8122 - loss: 0.5316 - val_accuracy: 0.7474 - val_auc:
0.8060 - val_loss: 0.5323
Epoch 92/100
129/129              0s 2ms/step -
accuracy: 0.7331 - auc: 0.8017 - loss: 0.5413 - val_accuracy: 0.7506 - val_auc:
```

```
0.8059 - val_loss: 0.5318
Epoch 93/100
129/129            0s 2ms/step -
accuracy: 0.7545 - auc: 0.8184 - loss: 0.5257 - val_accuracy: 0.7482 - val_auc:
0.8064 - val_loss: 0.5307
Epoch 94/100
129/129            0s 2ms/step -
accuracy: 0.7341 - auc: 0.8017 - loss: 0.5451 - val_accuracy: 0.7506 - val_auc:
0.8057 - val_loss: 0.5299
Epoch 95/100
129/129            0s 3ms/step -
accuracy: 0.7411 - auc: 0.8026 - loss: 0.5441 - val_accuracy: 0.7482 - val_auc:
0.8044 - val_loss: 0.5335
Epoch 96/100
129/129            0s 3ms/step -
accuracy: 0.7506 - auc: 0.8022 - loss: 0.5425 - val_accuracy: 0.7514 - val_auc:
0.8066 - val_loss: 0.5303
Epoch 97/100
129/129            0s 3ms/step -
accuracy: 0.7429 - auc: 0.8043 - loss: 0.5425 - val_accuracy: 0.7522 - val_auc:
0.8049 - val_loss: 0.5305
Epoch 98/100
129/129            0s 3ms/step -
accuracy: 0.7365 - auc: 0.8003 - loss: 0.5461 - val_accuracy: 0.7474 - val_auc:
0.8096 - val_loss: 0.5286
Epoch 99/100
129/129            0s 2ms/step -
accuracy: 0.7412 - auc: 0.7982 - loss: 0.5472 - val_accuracy: 0.7490 - val_auc:
0.8076 - val_loss: 0.5293
Epoch 100/100
129/129            0s 3ms/step -
accuracy: 0.7347 - auc: 0.7912 - loss: 0.5545 - val_accuracy: 0.7466 - val_auc:
0.8070 - val_loss: 0.5296

--- Final Evaluation -High-Dropout Stability ---
Train Loss: 0.5194
Train Accuracy: 0.7535
Train AUC: 0.8204

Test Loss: 0.5286
Test Accuracy: 0.7474
Test AUC: 0.8096
```

[65]:

```
# 3. Deep Dropout-Enhanced Neural Network

# 1. Early stopping to prevent overfitting
early_stop = callbacks.EarlyStopping(
```

```python
    monitor='val_loss',
    patience=10,
    restore_best_weights=True
)

# 2. Define the model
model = keras.Sequential([
    layers.Input(shape=(X_res.shape[1],)),
    layers.Dense(128, activation='relu'),
    layers.BatchNormalization(),
    layers.Dropout(0.4),

    layers.Dense(64, activation='relu'),
    layers.BatchNormalization(),
    layers.Dropout(0.3),

    layers.Dense(32, activation='relu'),
    layers.BatchNormalization(),
    layers.Dropout(0.2),

    layers.Dense(1, activation='sigmoid')
])

# 3. Compile the model
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=0.001),
    loss='binary_crossentropy',
    metrics=['accuracy', keras.metrics.AUC(name='auc')]
)

# 4. Train the model
history = model.fit(
    X_res, y_res,
    validation_data=(X_test, y_test),
    epochs=100,
    batch_size=32,
    callbacks=[early_stop],
    verbose=1
)

# 5. Evaluate the model
train_loss, train_accuracy, train_auc = model.evaluate(X_res, y_res, verbose=0)
test_loss, test_accuracy, test_auc = model.evaluate(X_test, y_test, verbose=0)

# 6. Print basic results
print("\n--- Final Evaluation -Deep Dropout-Enhanced Neural Network---")
print(f"Train Loss: {train_loss:.4f}")
```

```python
print(f"Train Accuracy: {train_accuracy:.4f}")
print(f"Train AUC: {train_auc:.4f}")

print(f"\nTest Loss: {test_loss:.4f}")
print(f"Test Accuracy: {test_accuracy:.4f}")
print(f"Test AUC: {test_auc:.4f}")

# Save predictions for visualization
y_pred_prob = model.predict(X_test)
y_pred = (y_pred_prob > 0.5).astype(int)
```

```
Epoch 1/100
129/129              4s 6ms/step -
accuracy: 0.5929 - auc: 0.6308 - loss: 0.7351 - val_accuracy: 0.7231 - val_auc:
0.7653 - val_loss: 0.6185
Epoch 2/100
129/129              1s 4ms/step -
accuracy: 0.6587 - auc: 0.7151 - loss: 0.6440 - val_accuracy: 0.7304 - val_auc:
0.7780 - val_loss: 0.5765
Epoch 3/100
129/129              1s 4ms/step -
accuracy: 0.7022 - auc: 0.7502 - loss: 0.6033 - val_accuracy: 0.7417 - val_auc:
0.7920 - val_loss: 0.5527
Epoch 4/100
129/129              1s 4ms/step -
accuracy: 0.6970 - auc: 0.7532 - loss: 0.5961 - val_accuracy: 0.7482 - val_auc:
0.7980 - val_loss: 0.5390
Epoch 5/100
129/129              1s 4ms/step -
accuracy: 0.7047 - auc: 0.7614 - loss: 0.5896 - val_accuracy: 0.7425 - val_auc:
0.7997 - val_loss: 0.5345
Epoch 6/100
129/129              1s 4ms/step -
accuracy: 0.7191 - auc: 0.7840 - loss: 0.5597 - val_accuracy: 0.7498 - val_auc:
0.8036 - val_loss: 0.5296
Epoch 7/100
129/129              0s 3ms/step -
accuracy: 0.7242 - auc: 0.7861 - loss: 0.5573 - val_accuracy: 0.7506 - val_auc:
0.8015 - val_loss: 0.5302
Epoch 8/100
129/129              0s 3ms/step -
accuracy: 0.7200 - auc: 0.7808 - loss: 0.5638 - val_accuracy: 0.7538 - val_auc:
0.8051 - val_loss: 0.5281
Epoch 9/100
129/129              0s 3ms/step -
accuracy: 0.7386 - auc: 0.8003 - loss: 0.5400 - val_accuracy: 0.7530 - val_auc:
0.8039 - val_loss: 0.5270
Epoch 10/100
```

```
129/129              0s 3ms/step -
accuracy: 0.7519 - auc: 0.8059 - loss: 0.5334 - val_accuracy: 0.7449 - val_auc:
0.8038 - val_loss: 0.5299
Epoch 11/100
129/129              0s 3ms/step -
accuracy: 0.7338 - auc: 0.7958 - loss: 0.5438 - val_accuracy: 0.7530 - val_auc:
0.8061 - val_loss: 0.5272
Epoch 12/100
129/129              0s 3ms/step -
accuracy: 0.7322 - auc: 0.7901 - loss: 0.5496 - val_accuracy: 0.7538 - val_auc:
0.8054 - val_loss: 0.5256
Epoch 13/100
129/129              1s 4ms/step -
accuracy: 0.7362 - auc: 0.7894 - loss: 0.5514 - val_accuracy: 0.7530 - val_auc:
0.8081 - val_loss: 0.5238
Epoch 14/100
129/129              0s 3ms/step -
accuracy: 0.7329 - auc: 0.7912 - loss: 0.5467 - val_accuracy: 0.7555 - val_auc:
0.8068 - val_loss: 0.5247
Epoch 15/100
129/129              0s 3ms/step -
accuracy: 0.7358 - auc: 0.7999 - loss: 0.5387 - val_accuracy: 0.7506 - val_auc:
0.8107 - val_loss: 0.5213
Epoch 16/100
129/129              0s 3ms/step -
accuracy: 0.7325 - auc: 0.7903 - loss: 0.5456 - val_accuracy: 0.7522 - val_auc:
0.8069 - val_loss: 0.5238
Epoch 17/100
129/129              0s 4ms/step -
accuracy: 0.7296 - auc: 0.7891 - loss: 0.5489 - val_accuracy: 0.7522 - val_auc:
0.8067 - val_loss: 0.5214
Epoch 18/100
129/129              1s 4ms/step -
accuracy: 0.7368 - auc: 0.7947 - loss: 0.5438 - val_accuracy: 0.7490 - val_auc:
0.8076 - val_loss: 0.5226
Epoch 19/100
129/129              1s 5ms/step -
accuracy: 0.7507 - auc: 0.8141 - loss: 0.5246 - val_accuracy: 0.7482 - val_auc:
0.8081 - val_loss: 0.5233
Epoch 20/100
129/129              1s 4ms/step -
accuracy: 0.7438 - auc: 0.7948 - loss: 0.5420 - val_accuracy: 0.7498 - val_auc:
0.8088 - val_loss: 0.5218
Epoch 21/100
129/129              1s 4ms/step -
accuracy: 0.7347 - auc: 0.7937 - loss: 0.5458 - val_accuracy: 0.7530 - val_auc:
0.8087 - val_loss: 0.5237
Epoch 22/100
```

```
129/129            1s 4ms/step -
accuracy: 0.7449 - auc: 0.8069 - loss: 0.5304 - val_accuracy: 0.7571 - val_auc:
0.8077 - val_loss: 0.5230
Epoch 23/100
129/129            0s 3ms/step -
accuracy: 0.7464 - auc: 0.8041 - loss: 0.5332 - val_accuracy: 0.7498 - val_auc:
0.8084 - val_loss: 0.5224
Epoch 24/100
129/129            0s 3ms/step -
accuracy: 0.7400 - auc: 0.7984 - loss: 0.5396 - val_accuracy: 0.7498 - val_auc:
0.8107 - val_loss: 0.5210
Epoch 25/100
129/129            0s 3ms/step -
accuracy: 0.7392 - auc: 0.8003 - loss: 0.5386 - val_accuracy: 0.7555 - val_auc:
0.8084 - val_loss: 0.5203
Epoch 26/100
129/129            0s 3ms/step -
accuracy: 0.7475 - auc: 0.8087 - loss: 0.5272 - val_accuracy: 0.7538 - val_auc:
0.8110 - val_loss: 0.5191
Epoch 27/100
129/129            0s 3ms/step -
accuracy: 0.7396 - auc: 0.8096 - loss: 0.5328 - val_accuracy: 0.7522 - val_auc:
0.8137 - val_loss: 0.5168
Epoch 28/100
129/129            0s 3ms/step -
accuracy: 0.7508 - auc: 0.8081 - loss: 0.5275 - val_accuracy: 0.7579 - val_auc:
0.8108 - val_loss: 0.5190
Epoch 29/100
129/129            1s 4ms/step -
accuracy: 0.7318 - auc: 0.7890 - loss: 0.5498 - val_accuracy: 0.7522 - val_auc:
0.8122 - val_loss: 0.5162
Epoch 30/100
129/129            1s 4ms/step -
accuracy: 0.7461 - auc: 0.8199 - loss: 0.5164 - val_accuracy: 0.7579 - val_auc:
0.8131 - val_loss: 0.5173
Epoch 31/100
129/129            1s 4ms/step -
accuracy: 0.7370 - auc: 0.8009 - loss: 0.5346 - val_accuracy: 0.7563 - val_auc:
0.8127 - val_loss: 0.5175
Epoch 32/100
129/129            1s 4ms/step -
accuracy: 0.7451 - auc: 0.8082 - loss: 0.5283 - val_accuracy: 0.7595 - val_auc:
0.8132 - val_loss: 0.5155
Epoch 33/100
129/129            1s 4ms/step -
accuracy: 0.7443 - auc: 0.8042 - loss: 0.5327 - val_accuracy: 0.7530 - val_auc:
0.8114 - val_loss: 0.5173
Epoch 34/100
```

```
129/129             1s 4ms/step -
accuracy: 0.7461 - auc: 0.8006 - loss: 0.5360 - val_accuracy: 0.7538 - val_auc:
0.8111 - val_loss: 0.5167
Epoch 35/100
129/129             1s 5ms/step -
accuracy: 0.7477 - auc: 0.8037 - loss: 0.5316 - val_accuracy: 0.7530 - val_auc:
0.8126 - val_loss: 0.5159
Epoch 36/100
129/129             1s 4ms/step -
accuracy: 0.7536 - auc: 0.8125 - loss: 0.5251 - val_accuracy: 0.7595 - val_auc:
0.8152 - val_loss: 0.5141
Epoch 37/100
129/129             1s 4ms/step -
accuracy: 0.7478 - auc: 0.8028 - loss: 0.5330 - val_accuracy: 0.7530 - val_auc:
0.8140 - val_loss: 0.5142
Epoch 38/100
129/129             1s 4ms/step -
accuracy: 0.7498 - auc: 0.8097 - loss: 0.5263 - val_accuracy: 0.7522 - val_auc:
0.8170 - val_loss: 0.5135
Epoch 39/100
129/129             0s 3ms/step -
accuracy: 0.7454 - auc: 0.8076 - loss: 0.5293 - val_accuracy: 0.7595 - val_auc:
0.8173 - val_loss: 0.5127
Epoch 40/100
129/129             0s 3ms/step -
accuracy: 0.7500 - auc: 0.8055 - loss: 0.5294 - val_accuracy: 0.7547 - val_auc:
0.8140 - val_loss: 0.5154
Epoch 41/100
129/129             0s 4ms/step -
accuracy: 0.7417 - auc: 0.8038 - loss: 0.5329 - val_accuracy: 0.7579 - val_auc:
0.8187 - val_loss: 0.5097
Epoch 42/100
129/129             0s 3ms/step -
accuracy: 0.7369 - auc: 0.8090 - loss: 0.5280 - val_accuracy: 0.7571 - val_auc:
0.8182 - val_loss: 0.5111
Epoch 43/100
129/129             0s 3ms/step -
accuracy: 0.7445 - auc: 0.8074 - loss: 0.5289 - val_accuracy: 0.7514 - val_auc:
0.8188 - val_loss: 0.5121
Epoch 44/100
129/129             0s 3ms/step -
accuracy: 0.7472 - auc: 0.8113 - loss: 0.5232 - val_accuracy: 0.7587 - val_auc:
0.8179 - val_loss: 0.5105
Epoch 45/100
129/129             0s 3ms/step -
accuracy: 0.7540 - auc: 0.8180 - loss: 0.5180 - val_accuracy: 0.7522 - val_auc:
0.8177 - val_loss: 0.5113
Epoch 46/100
```

```
129/129              0s 4ms/step -
accuracy: 0.7470 - auc: 0.8132 - loss: 0.5211 - val_accuracy: 0.7530 - val_auc:
0.8166 - val_loss: 0.5119
Epoch 47/100
129/129              1s 4ms/step -
accuracy: 0.7318 - auc: 0.8079 - loss: 0.5280 - val_accuracy: 0.7611 - val_auc:
0.8195 - val_loss: 0.5080
Epoch 48/100
129/129              0s 4ms/step -
accuracy: 0.7545 - auc: 0.8194 - loss: 0.5131 - val_accuracy: 0.7587 - val_auc:
0.8220 - val_loss: 0.5062
Epoch 49/100
129/129              0s 4ms/step -
accuracy: 0.7602 - auc: 0.8234 - loss: 0.5085 - val_accuracy: 0.7587 - val_auc:
0.8222 - val_loss: 0.5047
Epoch 50/100
129/129              0s 4ms/step -
accuracy: 0.7556 - auc: 0.8180 - loss: 0.5138 - val_accuracy: 0.7595 - val_auc:
0.8207 - val_loss: 0.5071
Epoch 51/100
129/129              0s 4ms/step -
accuracy: 0.7350 - auc: 0.7990 - loss: 0.5351 - val_accuracy: 0.7555 - val_auc:
0.8214 - val_loss: 0.5050
Epoch 52/100
129/129              0s 4ms/step -
accuracy: 0.7487 - auc: 0.8114 - loss: 0.5220 - val_accuracy: 0.7595 - val_auc:
0.8219 - val_loss: 0.5059
Epoch 53/100
129/129              1s 4ms/step -
accuracy: 0.7417 - auc: 0.8051 - loss: 0.5313 - val_accuracy: 0.7547 - val_auc:
0.8203 - val_loss: 0.5073
Epoch 54/100
129/129              1s 4ms/step -
accuracy: 0.7438 - auc: 0.8199 - loss: 0.5147 - val_accuracy: 0.7571 - val_auc:
0.8220 - val_loss: 0.5048
Epoch 55/100
129/129              0s 4ms/step -
accuracy: 0.7438 - auc: 0.8028 - loss: 0.5342 - val_accuracy: 0.7538 - val_auc:
0.8224 - val_loss: 0.5054
Epoch 56/100
129/129              0s 3ms/step -
accuracy: 0.7478 - auc: 0.8134 - loss: 0.5193 - val_accuracy: 0.7514 - val_auc:
0.8224 - val_loss: 0.5083
Epoch 57/100
129/129              0s 4ms/step -
accuracy: 0.7512 - auc: 0.8062 - loss: 0.5282 - val_accuracy: 0.7563 - val_auc:
0.8238 - val_loss: 0.5053
Epoch 58/100
```

```
129/129             1s 4ms/step -
accuracy: 0.7442 - auc: 0.8115 - loss: 0.5229 - val_accuracy: 0.7514 - val_auc:
0.8251 - val_loss: 0.5029
Epoch 59/100
129/129             1s 4ms/step -
accuracy: 0.7483 - auc: 0.8150 - loss: 0.5206 - val_accuracy: 0.7587 - val_auc:
0.8262 - val_loss: 0.5009
Epoch 60/100
129/129             1s 4ms/step -
accuracy: 0.7621 - auc: 0.8239 - loss: 0.5086 - val_accuracy: 0.7603 - val_auc:
0.8228 - val_loss: 0.5090
Epoch 61/100
129/129             1s 4ms/step -
accuracy: 0.7407 - auc: 0.8100 - loss: 0.5271 - val_accuracy: 0.7555 - val_auc:
0.8275 - val_loss: 0.5016
Epoch 62/100
129/129             1s 4ms/step -
accuracy: 0.7533 - auc: 0.8146 - loss: 0.5212 - val_accuracy: 0.7579 - val_auc:
0.8278 - val_loss: 0.5012
Epoch 63/100
129/129             1s 4ms/step -
accuracy: 0.7590 - auc: 0.8141 - loss: 0.5192 - val_accuracy: 0.7579 - val_auc:
0.8253 - val_loss: 0.5032
Epoch 64/100
129/129             1s 4ms/step -
accuracy: 0.7597 - auc: 0.8214 - loss: 0.5132 - val_accuracy: 0.7538 - val_auc:
0.8246 - val_loss: 0.5051
Epoch 65/100
129/129             1s 4ms/step -
accuracy: 0.7537 - auc: 0.8204 - loss: 0.5137 - val_accuracy: 0.7595 - val_auc:
0.8252 - val_loss: 0.5013
Epoch 66/100
129/129             1s 4ms/step -
accuracy: 0.7582 - auc: 0.8231 - loss: 0.5111 - val_accuracy: 0.7538 - val_auc:
0.8246 - val_loss: 0.5017
Epoch 67/100
129/129             1s 4ms/step -
accuracy: 0.7501 - auc: 0.8095 - loss: 0.5238 - val_accuracy: 0.7563 - val_auc:
0.8262 - val_loss: 0.5013
Epoch 68/100
129/129             1s 4ms/step -
accuracy: 0.7458 - auc: 0.8045 - loss: 0.5270 - val_accuracy: 0.7538 - val_auc:
0.8236 - val_loss: 0.5037
Epoch 69/100
129/129             1s 4ms/step -
accuracy: 0.7298 - auc: 0.7998 - loss: 0.5402 - val_accuracy: 0.7522 - val_auc:
0.8264 - val_loss: 0.5023
```

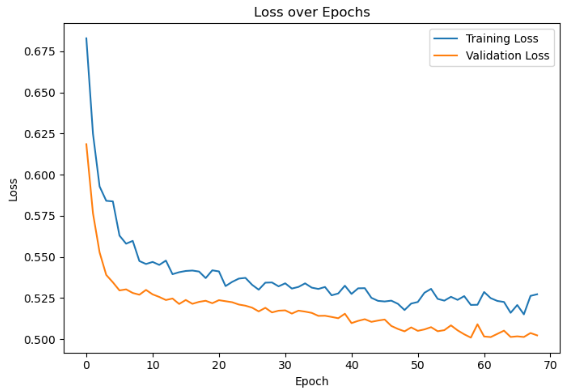
```
--- Final Evaluation -Deep Dropout-Enhanced Neural Network---
Train Loss: 0.4959
Train Accuracy: 0.7630
Train AUC: 0.8321

Test Loss: 0.5009
Test Accuracy: 0.7587
Test AUC: 0.8262
39/39            0s 4ms/step
```

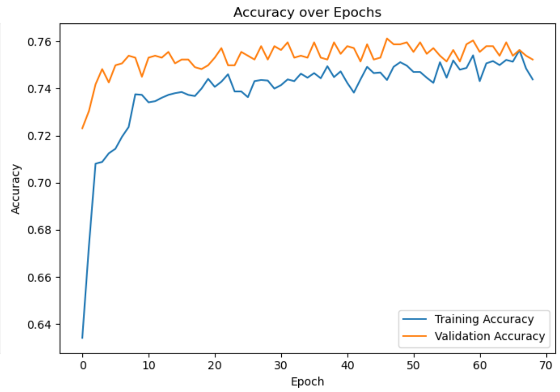
[66]:

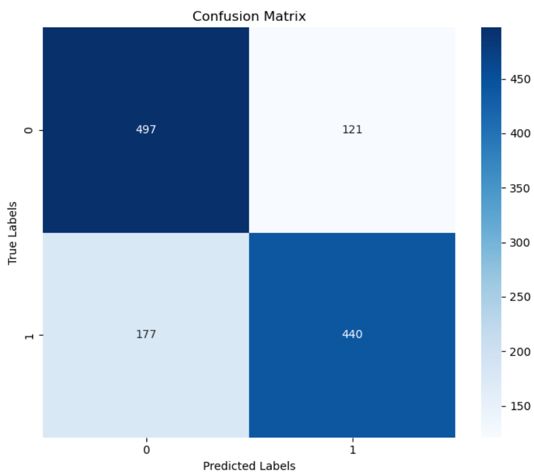
```
# --- PART 2: Visualization shown for Best Neural Network---

# 1. Training vs Validation Loss and Accuracy
fig, axes = plt.subplots(1, 2, figsize=(14,5))

# Plot Loss
axes[0].plot(history.history['loss'], label='Training Loss')
axes[0].plot(history.history['val_loss'], label='Validation Loss')
axes[0].set_title('Loss over Epochs')
axes[0].set_xlabel('Epoch')
axes[0].set_ylabel('Loss')
axes[0].legend()

# Plot Accuracy
axes[1].plot(history.history['accuracy'], label='Training Accuracy')
axes[1].plot(history.history['val_accuracy'], label='Validation Accuracy')
axes[1].set_title('Accuracy over Epochs')
axes[1].set_xlabel('Epoch')
axes[1].set_ylabel('Accuracy')
axes[1].legend()

plt.tight_layout()
plt.show()

# Calculate confusion matrix
cm = confusion_matrix(y_test, y_pred)
# Calculate ROC
fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)
roc_auc = auc(fpr, tpr)
# Create side-by-side plots
fig, axes = plt.subplots(1, 2, figsize=(14,6))

# Confusion Matrix Plot
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ax=axes[0])
axes[0].set_title('Confusion Matrix')
axes[0].set_xlabel('Predicted Labels')
axes[0].set_ylabel('True Labels')
```

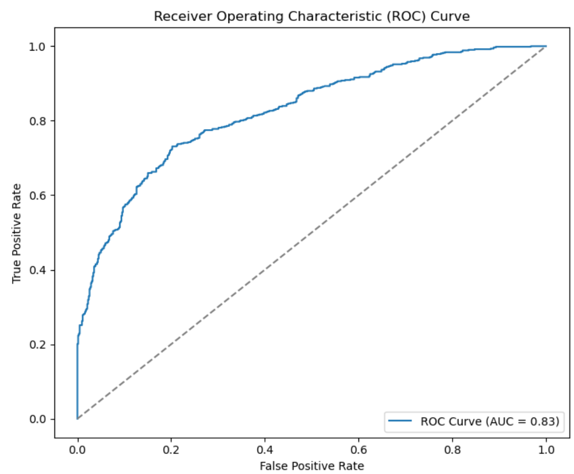
```
# ROC Curve Plot
axes[1].plot(fpr, tpr, label=f'ROC Curve (AUC = {roc_auc:.2f})')
axes[1].plot([0, 1], [0, 1], linestyle='--', color='gray')
axes[1].set_xlabel('False Positive Rate')
axes[1].set_ylabel('True Positive Rate')
axes[1].set_title('Receiver Operating Characteristic (ROC) Curve')
axes[1].legend(loc='lower right')

plt.tight_layout()
plt.show()

# 4. Learning Curve with Generalization Gap
fig, ax = plt.subplots(figsize=(8,5))
ax.plot(history.history['loss'], label='Training Loss')
ax.plot(history.history['val_loss'], label='Validation Loss')
ax.fill_between(
    range(len(history.history['loss'])),
    np.array(history.history['loss']),
    np.array(history.history['val_loss']),
    color='gray', alpha=0.3, label='Generalization Gap'
)
ax.set_title('Learning Curve with Generalization Gap')
ax.set_xlabel('Epoch')
ax.set_ylabel('Loss')
ax.legend()
plt.show()

# 5. Decision Boundary (only for 2D features)
if X_res.shape[1] == 2:
    from matplotlib.colors import ListedColormap

    x_min, x_max = X_res[:, 0].min() - 1, X_res[:, 0].max() + 1
    y_min, y_max = X_res[:, 1].min() - 1, X_res[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                         np.arange(y_min, y_max, 0.01))
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = (Z > 0.5).astype(int)
    Z = Z.reshape(xx.shape)

    plt.figure(figsize=(8,6))
    plt.contourf(xx, yy, Z, cmap=ListedColormap(('lightblue', 'lightcoral')))
    plt.scatter(X_res[:, 0], X_res[:, 1], c=y_res, cmap=ListedColormap(('blue',␣
↪'red')))
    plt.title('Decision Boundary')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
```

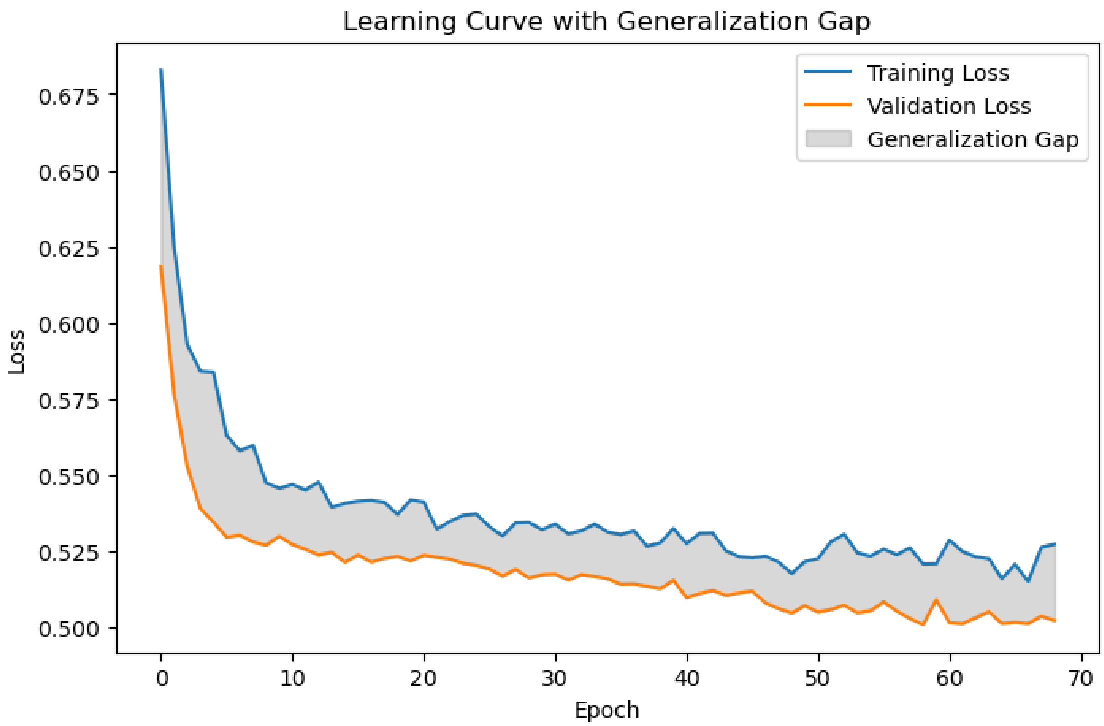
```
    plt.show()
else:
    print("Decision boundary plot skipped (input features > 2D).")
```






```
Decision boundary plot skipped (input features > 2D).
```