Model Code And Documentation:

Technology Stack:

The following tools and technologies were used for model development, evaluation, and deployment:

1. **Programming Language:**
   - Python: The primary programming language used for data manipulation, model training, and evaluation.

2. **Libraries/Frameworks:**
   - Pandas: For data manipulation, including loading datasets and handling missing values.
   - NumPy: For numerical operations and handling arrays efficiently.
   - XGBoost: For building and training the classification model. XGBoost is a powerful library for gradient boosting and is optimized for speed and performance.
   - Matplotlib & Seaborn: For data visualization and plotting graphs such as the AUC-ROC curve, confusion matrix, and feature importance.
   - Scikit-learn:
     - For various model evaluation metrics such as accuracy, precision, recall, F1-score, and AUC-ROC.
     - For hyperparameter tuning and implementing machine learning workflows.
   - Joblib: For saving and loading the trained model for future use or deployment.

3. **Hyperparameter Optimization Tool:**
    - Optuna: For automated hyperparameter optimization, helping to find the best combination of parameters for the XGBoost model.

4. **Development Environment:**
    - Jupyter Notebook: Used as the primary environment for writing and running Python code, providing an interactive platform for data exploration, model development, and analysis.
    - Anaconda: For managing the Python environment and installing necessary libraries.

5. **Deployment Tools (Optional):**
    - Joblib: Used for saving the trained model, which can then be deployed in a production environment for predictions on new data.
    - Flask/Django: These frameworks could be used to build an API around the model (though not implemented in this phase).

Model Code:

```
# Import necessary libraries for data processing, model building, and evaluation
import pandas as pd
import numpy as np
import xgboost as xgb
import matplotlib.pyplot as plt
import seaborn as sns
import joblib  # For saving the model
from sklearn.metrics import (roc_curve, confusion_matrix, roc_auc_score,
precision_recall_curve,
                        classification_report, accuracy_score, precision_score, recall_score,
                        f1_score, log_loss, balanced_accuracy_score)
```

```
# Step 1: Load the dataset
# Reading the input training and test datasets
df = pd.read_csv('X_Train_Data_Input.csv')
dft = pd.read_csv('Y_Train_Data_Target.csv')
dtest = pd.read_csv('X_Test_Data_Input.csv')
dftest = pd.read_csv('Y_Test_Data_Target.csv')

# Step 2: Prepare the data for training
# Dropping the ID column from both training and testing sets, and copying the target variable
x_train = df.drop(['ID'], axis=1)
y_train = dft['target'].copy()
x_test = dtest.drop(['ID'], axis=1)
y_test = dftest['target']

# Step 3: Define the best parameters for XGBoost classifier after hyperparameter tuning
# These parameters were determined using Optuna for optimal performance
best_params = {
    'subsample': 0.8,  # Randomly sample 80% of the data for each tree
    'n_estimators': 500,  # Use 500 trees for boosting
    'min_child_weight': 7,  # Minimum sum of weights of all observations for a child node
    'max_depth': 7,  # Maximum depth of a tree
    'learning_rate': 0.05,  # Step size shrinkage to prevent overfitting
    'lambda': 0,  # L2 regularization (0 means no regularization)
    'gamma': 0.1,  # Minimum loss reduction required to make a further partition
    'colsample_bytree': 0.9,  # Randomly sample 90% of columns for each tree
    'alpha': 0  # L1 regularization (0 means no regularization)
}

# Step 4: Initialize the XGBoost classifier using the best parameters
model = xgb.XGBClassifier(
    **best_params,  # Unpack the parameters into the model
    scale_pos_weight=9.61,  # Scale positive class to handle class imbalance
    use_label_encoder=False,  # Use sklearn style label encoding
    eval_metric='logloss'  # Log loss is the evaluation metric
)

# Step 5: Train the model on the training dataset
model.fit(x_train, y_train)
```

```
# Step 6: Make predictions on the test set
y_pred = model.predict(x_test)

# Predict probabilities for precision-recall curve and threshold adjustments
y_pred_prob = model.predict_proba(x_test)[:, 1]

# Step 7: Calculate precision, recall, and the thresholds
precision, recall, thresholds = precision_recall_curve(y_test, y_pred_prob)

# Step 8: Find the best threshold based on the F1-score
f1_scores = 2 * (precision * recall) / (precision + recall)
best_threshold_index = f1_scores.argmax()
best_threshold = thresholds[best_threshold_index]

# Output the best threshold
print(f"Best threshold: {best_threshold}")

# Step 9: Adjust predictions based on the optimal threshold
y_pred_adjusted = (y_pred_prob >= best_threshold).astype(int)

# Step 10: Evaluate the model with the adjusted threshold
# Generate classification report
print(classification_report(y_test, y_pred_adjusted))

# Step 11: Create and display the confusion matrix
cm = confusion_matrix(y_test, y_pred_adjusted)

# Step 12: Calculate the AUC-ROC score
roc_auc = roc_auc_score(y_test, y_pred_prob)

# Output the confusion matrix and AUC-ROC score
print("Confusion Matrix:\n", cm)
print("AUC-ROC:", roc_auc)

# Step 13: Calculate additional performance metrics
accuracy = (cm[0, 0] + cm[1, 1]) / cm.sum()  # Accuracy formula
precision = cm[1, 1] / (cm[1, 1] + cm[0, 1])  # Precision formula
recall = cm[1, 1] / (cm[1, 1] + cm[1, 0])  # Recall formula
f1_sco = 2 * (precision * recall) / (precision + recall)  # F1-score formula
logloss = log_loss(y_test, y_pred_prob)  # Log loss calculation
```

```
balanced_acc = balanced_accuracy_score(y_test, y_pred_adjusted)  # Balanced accuracy
formula

# Output additional metrics
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1-score:", f1_sco)
print("Log Loss:", logloss)
print("Balanced Accuracy:", balanced_acc)

# Step 14: Save the trained model for future use
joblib.dump(model, 'gstn_model.pkl')
```

Explanation of Key Methodology and Steps:

1. **Data Preprocessing**:
    - The dataset is loaded from CSV files, and we remove the ID column, which is irrelevant for model training.
    - The target column is separated to be used for classification.
2. **Model Selection**:
    - XGBoost is chosen for its performance in classification tasks and its ability to handle large datasets and imbalanced data.
3. **Hyperparameter Tuning**:
    - The model's hyperparameters (subsample, n_estimators, max_depth, etc.) are optimized using a tuning process to find the best combination for performance. Optuna was used for tuning.
4. **Model Training**:
    - The model is trained on the prepared training dataset using the XGBoost classifier with the tuned parameters.
5. **Model Evaluation**:
    - Predictions are made using both hard classification (class labels) and soft classification (probabilities).

    - A precision-recall curve is calculated, and the best threshold for maximizing the F1-score is identified.
    - Model evaluation metrics such as accuracy, precision, recall, F1-score, AUC-ROC, and confusion matrix are calculated and printed.
    - Additional metrics like log loss and balanced accuracy are also calculated.
6. **Saving the Model**:
    - After training and evaluating the model, it is saved using joblib.dump() for potential deployment or further testing.