People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research
University Benyoucef Benkhdda Algiers 1

Faculty of Sciences

Department of Computer Science

Specialty: Intelligent Information Systems Engineering (IISE)

TOPIC

Machine learning project

Prepared by:
- Nour ElHouda MESBAH
- Hebatallah CHEBLAL
- Sara HADDOUDA
- Farah AGOUILAL

Academic Year: 2024/2025

CONTENTS

LIST OF FIGURES

GENERAL INTRODUCTION

Machine learning, a branch of artificial intelligence, allows computers to learn from data and make predictions without explicit programming. With the growing volume of data generated by technology, it offers solutions to complex problems in fields like healthcare, finance, and marketing.

This project focuses on supervised learning, where models are trained with labeled data to predict outcomes. It includes two main approaches: Regression and Classification. Regression predicts numerical values, like stock prices or energy production, while Classification categorizes data into discrete classes, such as determining whether an auction bid is valid or fraudulent.

We used the Auction Verification Dataset from the UCI Machine Learning Repository, which addresses two machine learning problems: a Classification problem for verifying auction bid legitimacy, and a Regression problem for predicting "verification.time" values.

Data preprocessing was performed to improve dataset quality, and we applied various machine learning techniques to train and optimize models. Data visualization helped analyze results and compare model performance.

Google Colab was used for its interactive platform, enabling efficient collaboration through step-by-step code, visualizations, and explanations. Team members contributed to preprocessing, data visualization, and modeling, ensuring an organized workflow.

The following sections describe the dataset, preprocessing steps, methodologies, and results, showcasing how machine learning can solve real-world problems effectively.

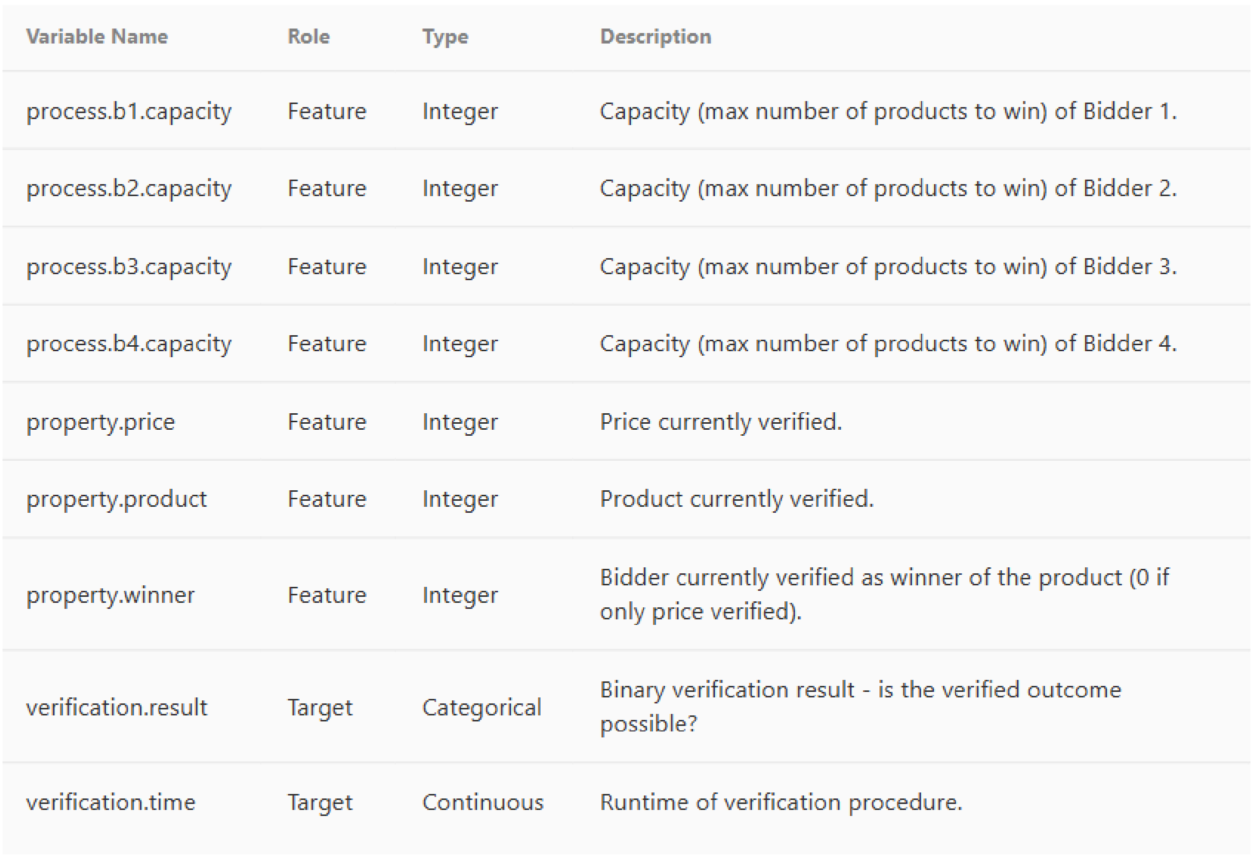
CHAPTER 1

DATA PREPROCESSING

1.1 Data-set

We chose the Auction Verification Dataset, which contains a collection of data points gathered from various online auctions. it contains 2043 instances.
hyperref Link to dataset : https://archive.ics.uci.edu/dataset/713/auction+verification Link to related paper : hhttps://ieeexplore.ieee.org/document/9721192

1.1.1 Dataset information



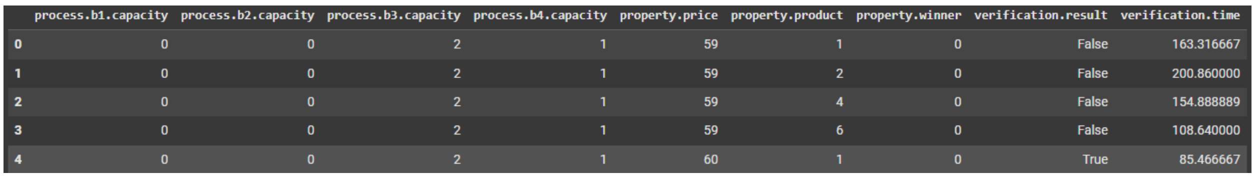
Variable Name	Role	Type	Description
process.b1.capacity	Feature	Integer	Capacity (max number of products to win) of Bidder 1.
process.b2.capacity	Feature	Integer	Capacity (max number of products to win) of Bidder 2.
process.b3.capacity	Feature	Integer	Capacity (max number of products to win) of Bidder 3.
process.b4.capacity	Feature	Integer	Capacity (max number of products to win) of Bidder 4.
property.price	Feature	Integer	Price currently verified.
property.product	Feature	Integer	Product currently verified.
property.winner	Feature	Integer	Bidder currently verified as winner of the product (0 if only price verified).
verification.result	Target	Categorical	Binary verification result - is the verified outcome possible?
verification.time	Target	Continuous	Runtime of verification procedure.

Figure 1.1: Dataset information

1.2 Preprocessing

- We imported the dataset and displayed the head:

```python
# Load the dataset
data = pd.read_csv('/content/drive/MyDrive/data.csv')
dframe=pd.DataFrame(data)
dframe.head()
```

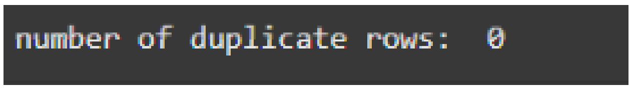


	process.b1.capacity	process.b2.capacity	process.b3.capacity	process.b4.capacity	property.price	property.product	property.winner	verification.result	verification.time
0	0	0	2	1	59	1	0	False	163.316667
1	0	0	2	1	59	2	0	False	200.860000
2	0	0	2	1	59	4	0	False	154.888889
3	0	0	2	1	59	6	0	False	108.640000
4	0	0	2	1	60	1	0	True	85.466667

Figure 1.2: load-head

- Then, we checked for duplicated rows, and the results indicate that there are no duplicate rows:

```
duplicate_rows_df = data[data.duplicated()]
print("number of duplicate rows: ", duplicate_rows_df.shape[0])
```

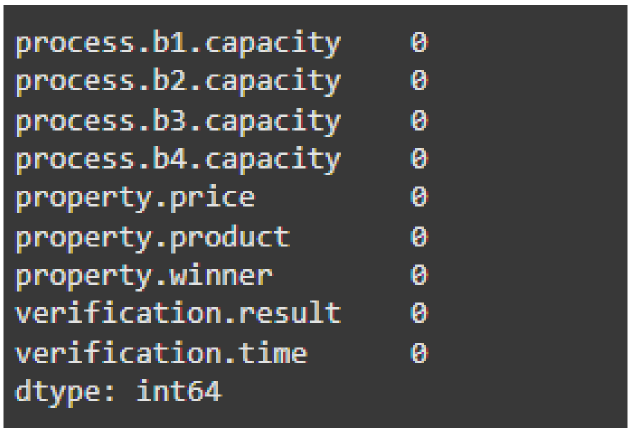


```
number of duplicate rows:  0
```

Figure 1.3: Duplicate rows

- Next, we checked for missing or null values :

```
print(data.isnull().sum())
```



```
process.b1.capacity    0
process.b2.capacity    0
process.b3.capacity    0
process.b4.capacity    0
property.price         0
property.product       0
property.winner        0
verification.result    0
verification.time      0
dtype: int64
```

Figure 1.4: Null values

- Pairplot :

```
sns.pairplot(data, diag_kind='kde')
```

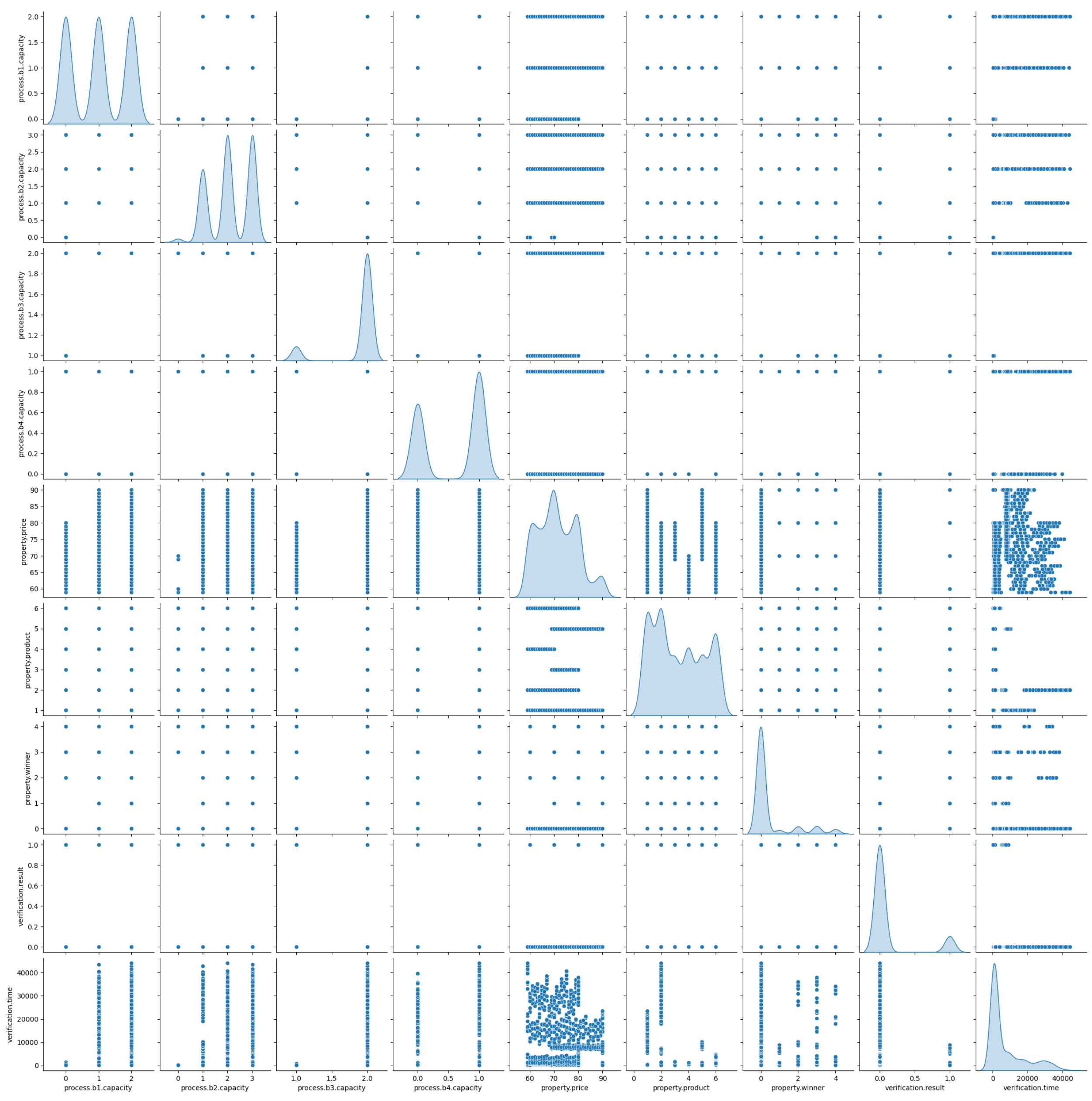
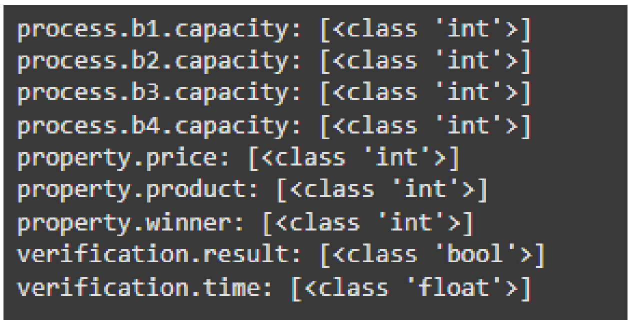


Figure 1.5: Pair plot

- Data type :

```
for col in data.columns:
    print(f"{col}: {data[col].apply(type).unique()}")
```



```
process.b1.capacity: [<class 'int'>]
process.b2.capacity: [<class 'int'>]
process.b3.capacity: [<class 'int'>]
process.b4.capacity: [<class 'int'>]
property.price: [<class 'int'>]
property.product: [<class 'int'>]
property.winner: [<class 'int'>]
verification.result: [<class 'bool'>]
verification.time: [<class 'float'>]
```

Figure 1.6: Data type

- Now, search for outliers :

```python
# Visualize outliers using boxplots
num_columns = ['process.b1.capacity', 'process.b2.capacity',
'process.b3.capacity', 'process.b4.capacity', 'property.price',
'property.product', 'property.winner', 'verification.time']
for col in num_columns:
    sns.boxplot(x=data[col])
    plt.title(col)
    plt.show()
```

- Histograms :

```python
num_columns = ['process.b1.capacity', 'process.b2.capacity',
'process.b3.capacity', 'process.b4.capacity', 'property.price',
'property.product', 'property.winner', 'verification.time']

# Create subplots
fig, axes = plt.subplots(nrows=2, ncols=4, figsize=(15, 8))

# Flatten the axes array to easily loop through
axes = axes.flatten()

# Loop through each column and plot its histogram
for i, column in enumerate(num_columns):
    axes[i].hist(data[column], bins=30)
    axes[i].set_title(column)

# Adjust layout to prevent overlap
```

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

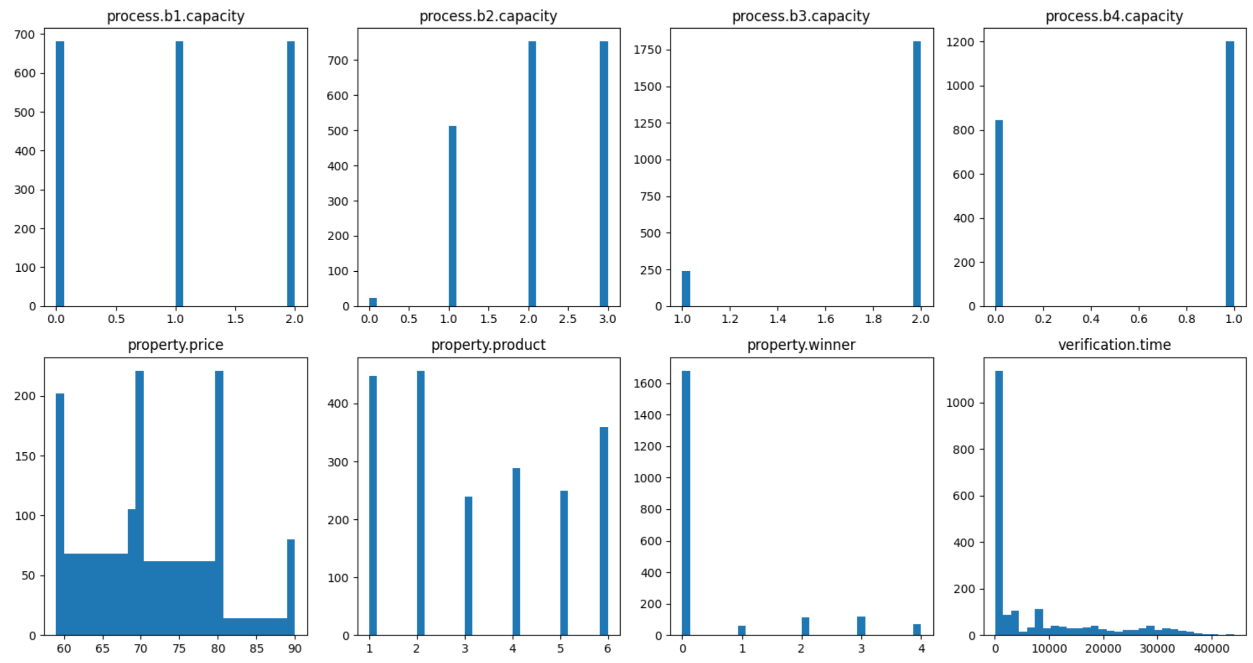


Figure 1.7: Histograms

- Then , features scaling with min-max method :

```python
num_columns = ['process.b1.capacity', 'process.b2.capacity',
'process.b3.capacity', 'process.b4.capacity', 'property.price',
'property.product', 'property.winner']

# Initialize the MinMaxScaler
scaler = MinMaxScaler()

# Fit and transform the features
data[num_columns] = scaler.fit_transform(data[num_columns])
```

- Correlation matrix visualization :

```python
# Correlation matrix
corr_matrix = data.corr()

# Visualize
```

```python
sns.heatmap(corr_matrix, annot=True, cmap="coolwarm")
plt.show()
```

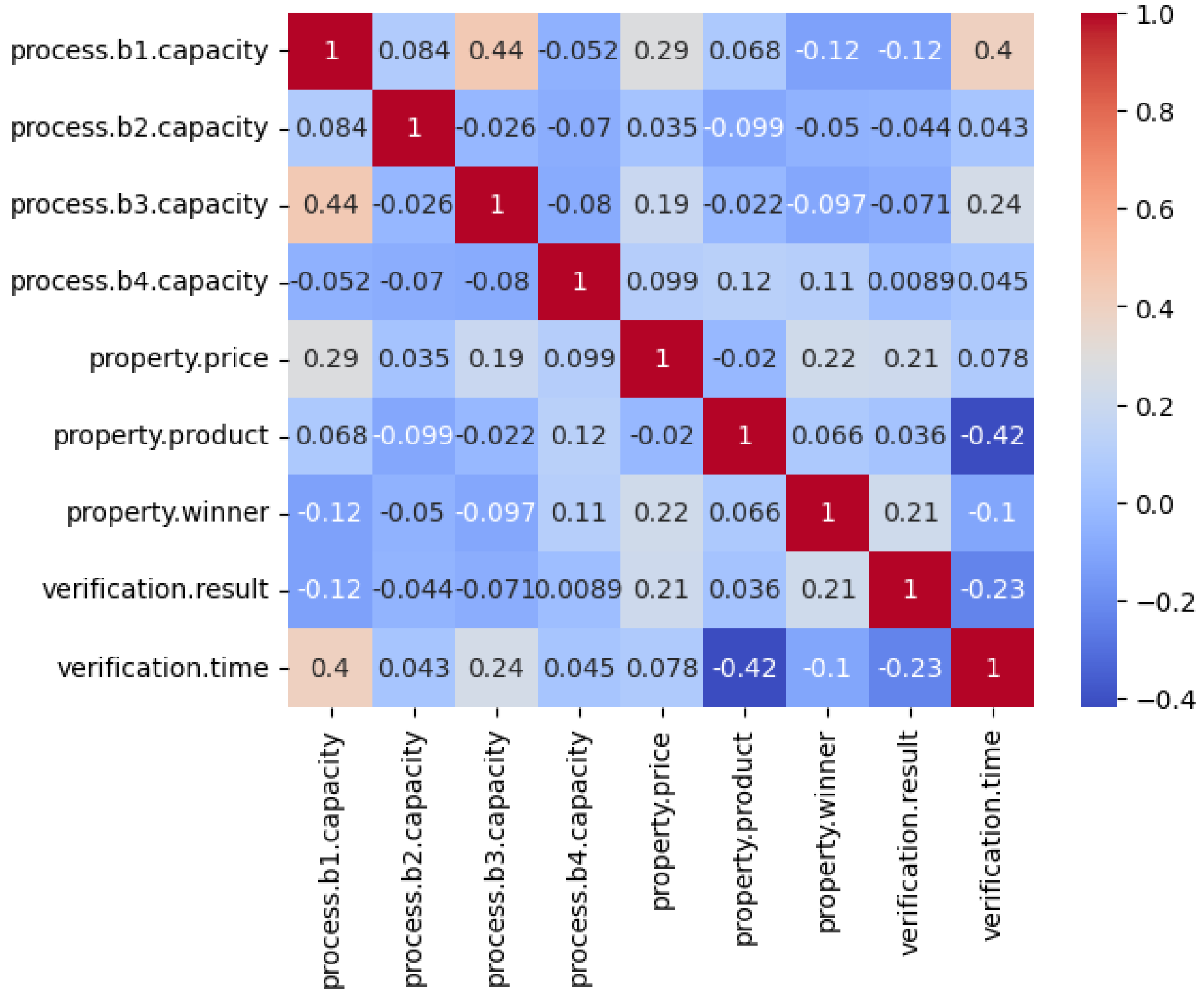


Figure 1.8: Correlation matrix

- Features selection for regression :

```python
# Define the numerical columns (excluding the target)
num_columns = data.select_dtypes(include=['number']).columns.tolist()
num_columns.remove('verification.time')  # Exclude the target column

# Step 1: Recursive Feature Elimination (RFE) with RandomForestRegressor
model = RandomForestRegressor(random_state=42)
rfe = RFE(estimator=model, n_features_to_select=5)  # Select top 5 features
rfe.fit(data[num_columns], data['verification.time'])

```

```
# Step 2: Select important features
selected_featuresreg = [num_columns[i] for i in range(len(num_columns)) if rfe.suppor
print("Selected Features:", selected_featuresreg)

# Step 3: Reduced dataset with selected features
reduced_datareg = data[selected_featuresreg]
```

And this is the result:
Selected Features: ['process.b1.capacity', 'process.b4.capacity', 'property.price', 'property.product', 'verification.result']

- Features selection for classification :

```
from sklearn.feature_selection import RFE
from sklearn.ensemble import RandomForestClassifier

# Ensure your target column exists
target_column = 'verification.result'

# Step 1: Recursive Feature Elimination (RFE)
model = RandomForestClassifier(random_state=42)
rfe = RFE(estimator=model, n_features_to_select=5)  # Select top 5 features
rfe.fit(data[num_columns], data[target_column])

# Step 2: Select important features
selected_features = [num_columns[i] for i in range(len(num_columns)) if rfe.support_[
print("Selected Features:", selected_features)

# Step 3: Reduced dataset with selected features
reduced_data = data[selected_features]
```

And this is the result:
Selected Features: [Selected Features: ['process.b1.capacity', 'process.b2.capacity', 'property.price', 'property.product', 'property.winner']

- Verification for imbalanced classification :

```
#imbalenced data
data['verification.result'].value_counts().plot(kind='bar')
```

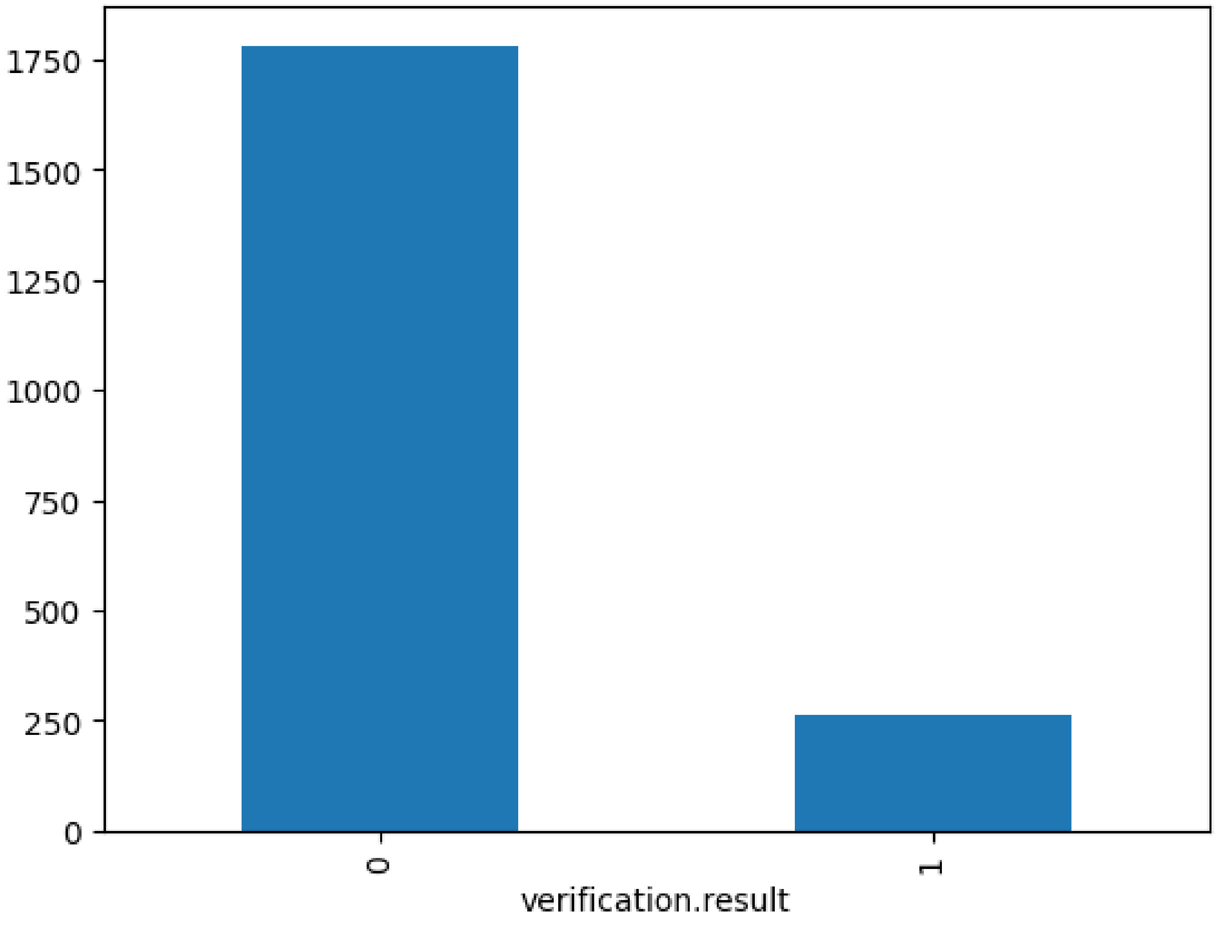
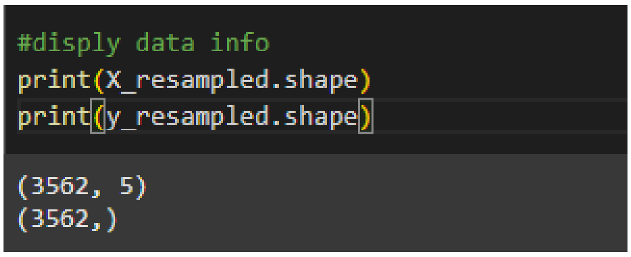


Figure 1.9: Classification histogram

- Fixing imbalanced data using SMOTE algorithm :
  SMOTE: creates new data points for the smaller (minority) class by taking two similar existing points in that class and generating a new point somewhere in between them.

```
#imbalenced data
data['verification.result'].value_counts().plot(kind='bar')
```



```
#disply data info
print(X_resampled.shape)
print(y_resampled.shape)

(3562, 5)
(3562,)
```

Figure 1.10: data info

- Afterward, we saved two datasets: one for regression and the other for classification.

CHAPTER 2 REGRESSION

2.1 Regression Problem

Regression in machine learning refers to a supervised learning technique where the goal is to predict a continuous numerical value based on one or more independent features. In the context of the Auction Verification Dataset, regression can be interpreted as the process of predicting numerical outcomes related to auction transactions.

2.2 Data Preparation

- Load Data: The preprocessed dataset is loaded from a CSV file into a pandas DataFrame. We tried using only the selected features, but the accuracy was poor, so we decided to use all the original features instead.

- Separate Features and Target: The features (input variables) are separated from the target variable (verification.result), which we aim to predict.

```
# Separate features (X) and target (y)
X = data.drop(['verification.time'], axis=1).values  # All columns except the target
y = data['verification.time'].values  #  Target
```

- Here, we added the Bias:

```
# Add intercept term to X (for the bias term in the model)
X = np.c_[np.ones(X.shape[0]), X]  # Add a column of ones
```

2.3 Normal equation

2.3.1 Defining the Normal Equation Function

```
def normal_equation(X, y):
    return np.linalg.inv(X.T @ X) @ X.T @ y
```

- This function calculates the theta parameters for a regression model using the normal equation:

$$\theta = \left(X^\top X\right)^{-1} X^\top y$$

2.3.2 Setting Up K-Fold Cross-Validation

```python
kf = KFold(n_splits=5, shuffle=True, random_state=42)
mse_scores = []
```

- We choose to splits the dataset into 5 folds for cross-validation, shuffling the data to ensure randomness.

2.3.3 Cross-Validation Loop

```python
for train_index, test_index in kf.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
```

- So, for each fold, the data is split into training and test sets.

```python
theta = normal_equation(X_train, y_train)
```

- The theta parameters are calculated using the training data.

```python
y_pred = X_test @ theta
```

- Predictions are made on the test data using the computed theta.

```python
mse = np.mean((y_test - y_pred) ** 2)
mse_scores.append(mse)

```

- The MSE is calculated as the mean of squared differences between the actual (y-test) and predicted (y-pred) values.

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2$$

2.3.4 Visualization of MSE Scores

```python
plt.figure(figsize=(8, 5))
plt.plot(range(1, len(mse_scores) + 1), mse_scores, marker='o',
linestyle='-', color='b', label='MSE per Fold')
plt.axhline(y=np.mean(mse_scores), color='r', linestyle='--',
label='Mean MSE')
plt.title("K-Fold Cross-Validation MSE")
plt.xlabel("Fold")
plt.ylabel("Mean Squared Error")
plt.legend()
plt.grid(True)
plt.show()
```

- The MSE scores for each fold are plotted, and a horizontal dashed line represents the average MSE across folds.

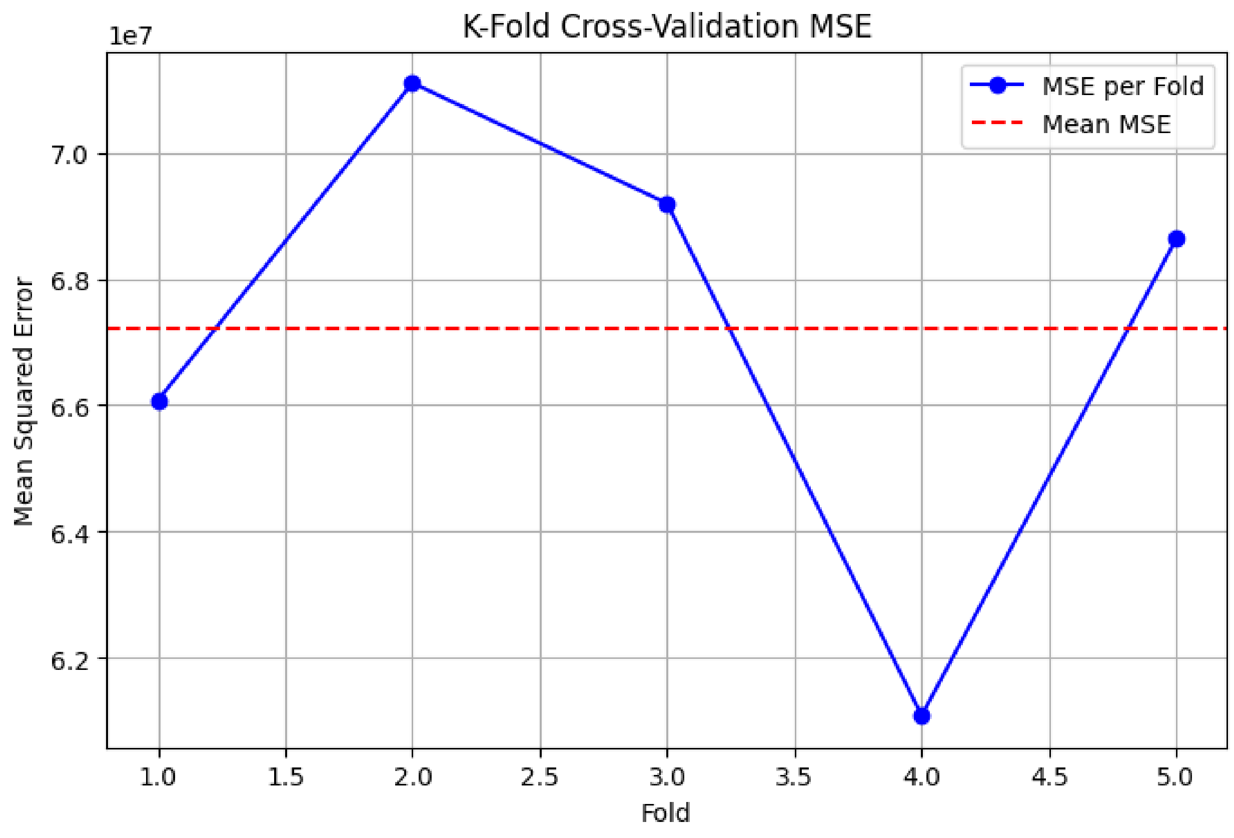


Figure 2.1: K-Fold Cross-Validation MSE

```python
print(f"Average MSE across folds: {np.mean(mse_scores):.4f}")
```

- The mean of all fold MSE values is computed and displayed as the final performance metric.
  Average MSE across folds: 67217681.9325

2.4 Gradient Descent

2.4.1 Gradient Descent Function

```python
def gradient_descent(X, y, alpha, num_iters):
    m = len(y)
    thetas = np.zeros(X.shape[1])  # Initialize thetas
    cost_history = []  # To track cost function values

    for i in range(num_iters):
        predictions = X @ thetas  # Compute predictions
        errors = predictions - y
        gradient = (1 / m) * (X.T @ errors)  # Compute gradients
        thetas -= alpha * gradient  # Update thetas

        # Compute the cost function (Mean Squared Error)
        cost = (1 / (2 * m)) * np.sum(errors**2)
        cost_history.append(cost)

    return thetas, cost_history
```

- Gradient Descent: The function optimizes the weights (θ) of a linear regression model by minimizing the cost function. It uses: Predictions: Computed using the formula:

$$\hat{y} = X\theta$$

- Gradient: The gradient of the cost function with respect to the weights is used to update them iteratively:

$$\theta := \theta - \alpha \cdot \nabla J(\theta)$$

- Cost Function (MSE): The cost function is updated after each iteration, and it tracks the Mean Squared Error for convergence analysis.

- Learning Rate (α): Controls the step size in each iteration.

2.4.2 Adjusted R-squared Function

```python
def adjusted_r2(y_true, y_pred, p):
    n = len(y_true)
    ss_res = np.sum((y_true - y_pred) ** 2)
    ss_tot = np.sum((y_true - np.mean(y_true)) ** 2)
    r2 = 1 - (ss_res / ss_tot)
    adj_r2 = 1 - ((1 - r2) * (n - 1)) / (n - p - 1)
    return adj_r2

```

2.4.3 k-Fold Cross-Validation with Cost Function and Adjusted R-squared

```python
def k_fold_cross_validation_with_cost_and_r2_plot(X, y, k, alpha, num_iters):
    kf = KFold(n_splits=k, shuffle=True, random_state=42)
    # Shuffle ensures randomness
    metrics = []  # To store Mean Squared Error for each fold
    adj_r2_scores = []  # To store Adjusted R^2 for each fold
    cost_histories = []  # To track cost function history for each fold

    for fold, (train_index, test_index) in enumerate(kf.split(X)):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        # Train the model using Gradient Descent
        thetas, cost_history = gradient_descent(X_train, y_train, alpha,
        num_iters)
        cost_histories.append(cost_history)

        # Make predictions
        y_pred = X_test @ thetas

        # Compute Mean Squared Error
        mse = mean_squared_error(y_test, y_pred)
        metrics.append(mse)

        # Calculate Adjusted R-squared
        p = X_train.shape[1]  # Number of features
        adj_r2 = adjusted_r2(y_test, y_pred, p)
        adj_r2_scores.append(adj_r2)

        print(f"Fold {fold + 1} MSE: {mse}, Adjusted R^2: {adj_r2}")

    return metrics, adj_r2_scores, cost_histories

```

- Metrics Collected: The MSE and Adjusted R-squared scores for each fold are stored, along with the cost history for plotting the cost function convergence.

2.4.4 Running k-Fold Cross-Validation

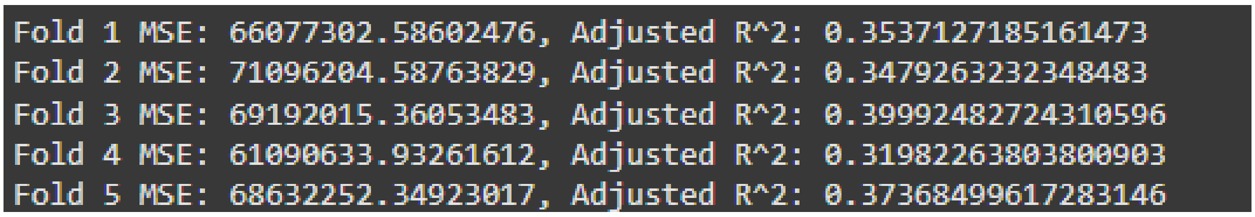
```
k = 5  # Number of folds
alpha = 0.1  # Learning rate
num_iters = 5000  # Number of iterations

# Perform k-fold cross-validation
mse_scores, adj_r2_scores, cost_histories = k_fold_cross_validation_with_
cost_and_r2_plot(X, y, k, alpha, num_iters)
```

- We specified the model's parameters, including the number of folds for cross-validation (k=5), the learning rate (α=0.1), and the number of iterations for the gradient descent optimization process.

2.4.5 Plotting Cost Function Convergence

```
# Plot cost function for each fold
plt.figure(figsize=(10, 6))
for fold, cost_history in enumerate(cost_histories):
    plt.plot(range(len(cost_history)), cost_history, label=f"Fold {fold + 1}")
plt.xlabel("Iterations")
plt.ylabel("Cost (MSE)")
plt.title("Cost Function Convergence for Each Fold")
plt.legend()
plt.show()
```

- This part visualizes how the cost function (MSE) converges over iterations for each fold, allowing us to observe the learning process of the model.



```
Fold 1 MSE: 66077302.58602476, Adjusted R^2: 0.3537127185161473
Fold 2 MSE: 71096204.58763829, Adjusted R^2: 0.3479263232348483
Fold 3 MSE: 69192015.36053483, Adjusted R^2: 0.39992482724310596
Fold 4 MSE: 61090633.93261612, Adjusted R^2: 0.31982263803800903
Fold 5 MSE: 68632252.34923017, Adjusted R^2: 0.37368499617283146
```

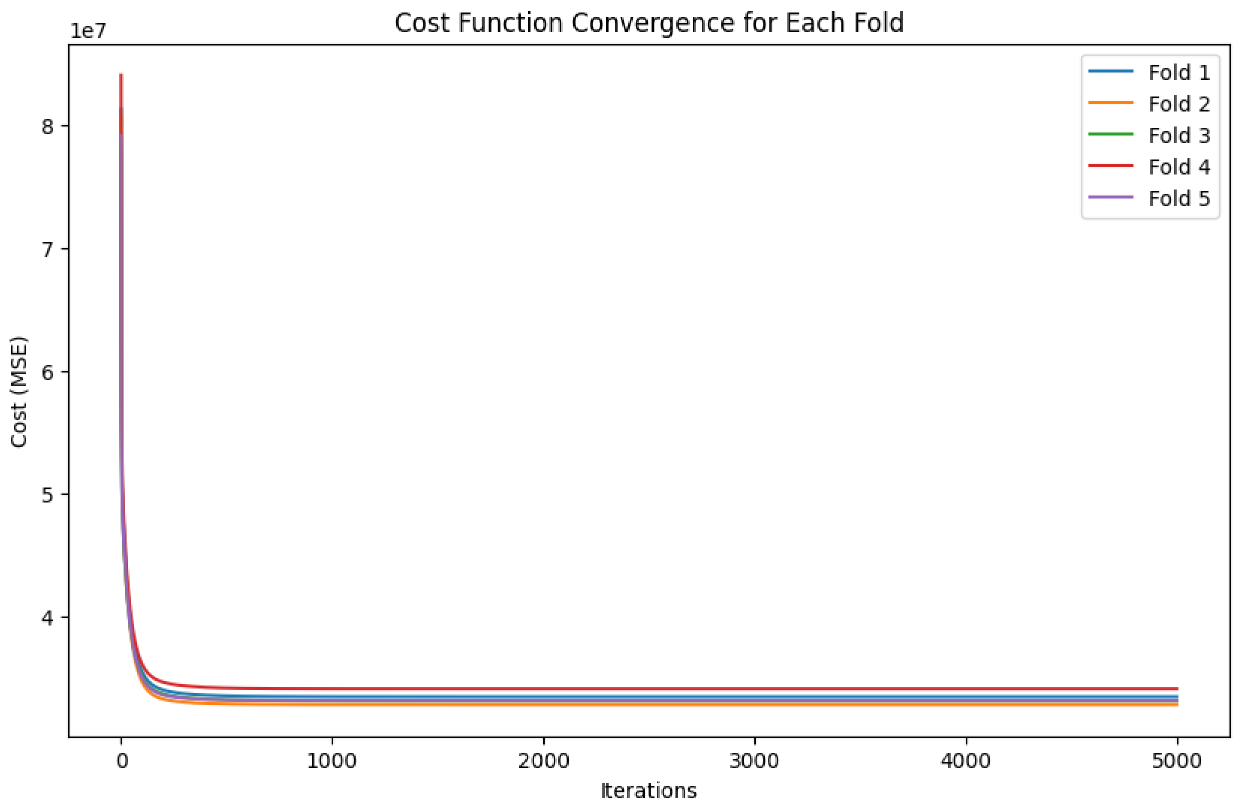
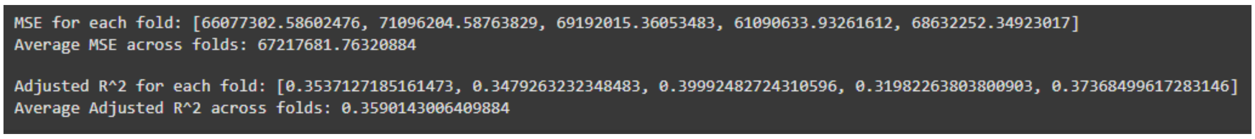


Figure 2.2: Cost Function Convergence for Each Fold

2.4.6 Running k-Fold Cross-Validation

```python
# Print MSE and Adjusted R^2 results
print("\nMSE for each fold:", mse_scores)
print("Average MSE across folds:", np.mean(mse_scores))
print("\nAdjusted R^2 for each fold:", adj_r2_scores)
print("Average Adjusted R^2 across folds:", np.mean(adj_r2_scores))
```



```
MSE for each fold: [66077302.58602476, 71096204.58763829, 69192015.36053483, 61090633.93261612, 68632252.34923017]
Average MSE across folds: 67217681.76320884

Adjusted R^2 for each fold: [0.3537127185161473, 0.3479263232348483, 0.39992482724310596, 0.31982263803800903, 0.37368499617283146]
Average Adjusted R^2 across folds: 0.3590143006409884
```

2.5 Model evaluation

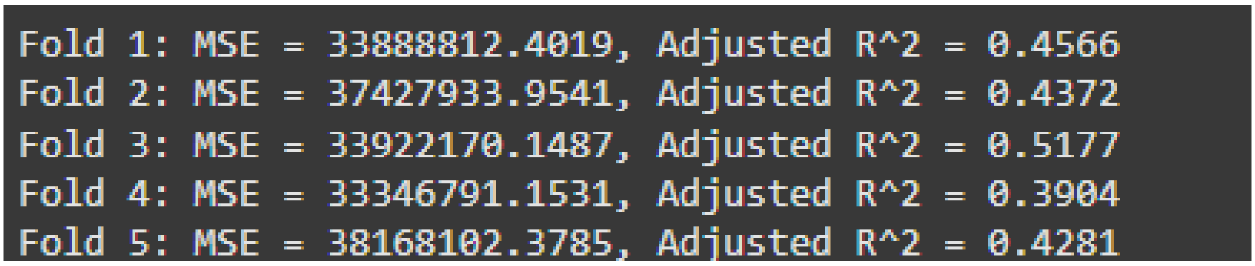
After these results, we made several adjustments to improve the accuracy of the model.Polynomial feature expansion increases the model's ability to capture complex, non-linear patterns. Ridge regression with regularization prevents over-fitting and improves generalization.

```python
def k_fold_ridge_with_scaling(X, y, k, alpha, degree):
    kf = KFold(n_splits=k, shuffle=True, random_state=42)
    mse_scores = []
    adj_r2_scores = []

    for fold, (train_index, test_index) in enumerate(kf.split(X)):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        # Build a pipeline with polynomial features, scaling, and Ridge regression
        pipeline = Pipeline([
            ("poly", PolynomialFeatures(degree=degree, include_bias=False)),
            ("scaler", StandardScaler()),
            ("ridge", Ridge(alpha=alpha))
        ])

        # Train the pipeline
        pipeline.fit(X_train, y_train)
        y_pred = pipeline.predict(X_test)

        # Calculate MSE and Adjusted R²
        mse = mean_squared_error(y_test, y_pred)
        mse_scores.append(mse)

        p = pipeline.named_steps['poly'].n_output_features_
        # Get number of features after polynomial expansion
        adj_r2 = adjusted_r2(y_test, y_pred, p)
        adj_r2_scores.append(adj_r2)

```

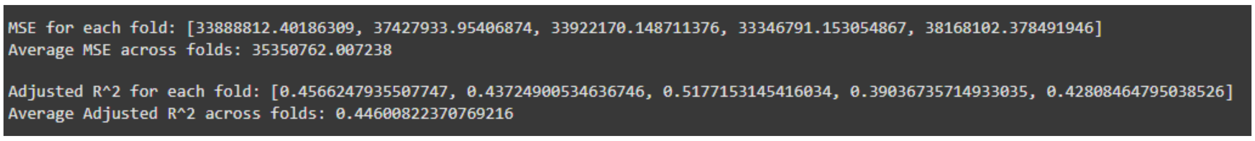
```
            print(f"Fold {fold + 1}: MSE = {mse:.4f}, Adjusted R^2 = {adj_r2:.4f}")

        return mse_scores, adj_r2_scores
```

```
# Run the updated Ridge regression with scaling
k = 5  # Number of folds
alpha = 0.1  # Regularization parameter
degree = 3  # Polynomial degree
```

- In an effort to improve the model's performance, we experimented with different values for the regularization parameter (α) and the polynomial degree. Various combinations of these hyper-parameters were tested to find the optimal values that minimized the Mean Squared Error (MSE) and maximized the Adjusted R^2 score. After evaluating the model with several settings, the final choice was to set α=0.1 and the polynomial degree to 3. However, despite these adjustments, the model's performance remained poor, indicating that further improvements or different techniques may be required.



```
Fold 1: MSE = 33888812.4019, Adjusted R^2 = 0.4566
Fold 2: MSE = 37427933.9541, Adjusted R^2 = 0.4372
Fold 3: MSE = 33922170.1487, Adjusted R^2 = 0.5177
Fold 4: MSE = 33346791.1531, Adjusted R^2 = 0.3904
Fold 5: MSE = 38168102.3785, Adjusted R^2 = 0.4281
```



```
MSE for each fold: [33888812.40186309, 37427933.95406874, 33922170.148711376, 33346791.153054867, 38168102.378491946]
Average MSE across folds: 35350762.007238

Adjusted R^2 for each fold: [0.4566247935507747, 0.43724900534636746, 0.5177153145416034, 0.39036735714933035, 0.42808464795038526]
Average Adjusted R^2 across folds: 0.44600822370769216
```

CHAPTER 3 CLASSIFICATION

3.1 Classification Problem

The second problem we address is a classification problem, where the goal is to predict a category based on a set of input features. Specifically, this is a binary classification task, meaning the model must choose between two possible outcomes. In our case, the task is to determine whether the verification result of an auction is possible or impossible, based on different characteristics of the auction. A prediction of "1" signifies that the verification result is possible, while a prediction of "0" indicates that it is impossible. This helps in assessing the validity of an auction's outcome based on available data.

3.2 Data Preparation and Cross-Validation

As the first step of classification, this code implements K-Fold cross-validation and visualizes the class distribution in the training and test sets.

- Load Data: The preprocessed dataset is loaded from a CSV file into a pandas DataFrame.
- Separate Features and Target: The features (input variables) are separated from the target variable (verification.result), which we aim to predict.
- up K-Fold Cross-Validation: A 5-fold cross-validation is applied, shuffling the data and splitting it into training and test sets.
- Visualize Class Distribution: For the first fold, pie charts are created to visualize the distribution of classes in both the training and test sets.
- Display Charts: The pie charts are displayed to assess the balance of classes across the training and test sets.

```
import pandas as pd
from sklearn.model_selection import KFold
import matplotlib.pyplot as plt

# Load the dataset
processed_data = pd.read_csv('/content/processed_data_with_target.csv')
X = processed_data.drop('verification.result', axis=1)
y = processed_data['verification.result']
# KFold cross-validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)
for train_index, test_index in kf.split(X):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]
```

3.3 Logistic Regression

The second step focuses on applying Logistic Regression, evaluating its performance through accuracy and a confusion matrix, and visualizing the results.

- **Initialize the Model:** A Logistic Regression model is instantiated with a maximum iteration limit of 1000 to ensure convergence.
- **Training:** The model is trained using the training set (Xtrain and ytrain).
- **Predictions:** Predictions are made on the test set (Xtest) to evaluate model performance.
- **Evaluation:** The model's accuracy is computed, and a confusion matrix is generated to assess prediction results.
- **Visualization:**
  - A heatmap of the confusion matrix is created to illustrate the distribution of predicted and actual classes.
  - A bar chart displays the accuracy score for Logistic Regression.

```python
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Initialize the Logistic Regression model
log_reg = LogisticRegression(max_iter=1000)

# Train the model
log_reg.fit(X_train, y_train)

# Make predictions
y_pred_log_reg = log_reg.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred_log_reg)
conf_matrix = confusion_matrix(y_test, y_pred_log_reg)

print("Accuracy (Logistic Regression):", accuracy)
print("Confusion Matrix (Logistic Regression):\n", conf_matrix)

```

```
# Visualize the confusion matrix
plt.figure(figsize=(6, 5))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Class 0', 'Class 1'],
            yticklabels=['Class 0', 'Class 1'])
plt.title('Confusion Matrix - Logistic Regression')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()

# Visualize the accuracy
plt.figure(figsize=(5, 4))
plt.bar(['Logistic Regression'], [accuracy], color='skyblue')
plt.title('Accuracy - Logistic Regression')
plt.ylim(0, 1)
plt.ylabel('Accuracy')
plt.show()
```

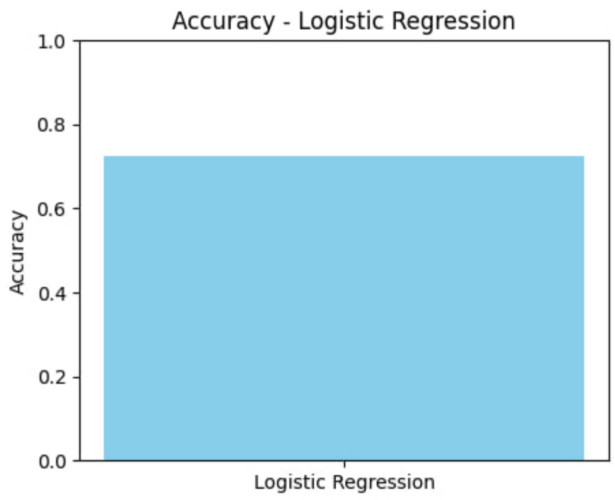


Figure 3.1: Logistic Regression Accuracy

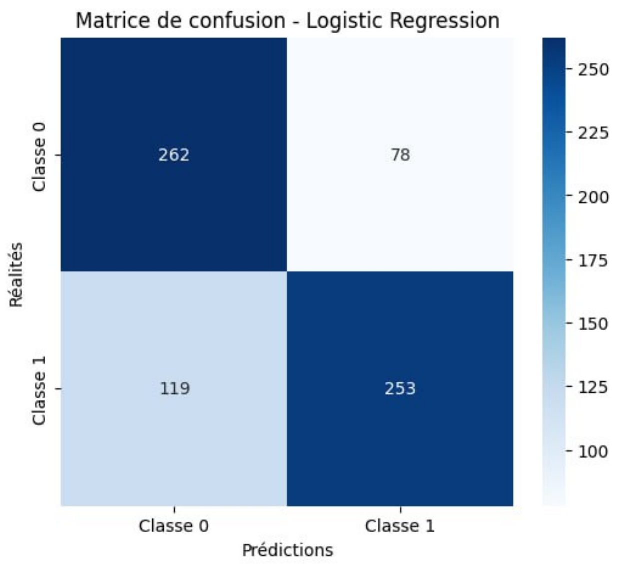


Figure 3.2: Logistic Regression matrix of confusion

3.4 Neural Network

The third step involves using a Neural Network (Multi-Layer Perceptron) for classification. The process includes training the model, evaluating its performance through accuracy and a confusion matrix, and visualizing the results.

- **Initialize the Model:** A neural network classifier (MLPClassifier) is instantiated with one hidden layer of 100 neurons and a random seed for reproducibility.
- **Training:** The model is trained on the training dataset, specifically the features and the target variable.
- **Predictions:** Predictions are made on the test dataset to evaluate the model's performance.
- **Evaluation:**
  - Compute the model's accuracy to measure performance.
  - Generate a confusion matrix to analyze prediction results.
- **Visualization:**
  - A heatmap of the confusion matrix is created for a visual overview of prediction and actual class distributions.
  - A bar chart is used to present the accuracy score of the neural network model.

```python
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Initialize the neural network model
mlp_model = MLPClassifier(hidden_layer_sizes=(100,), random_state=42)

# Train the model
mlp_model.fit(X_train, y_train)

# Make predictions
y_pred_mlp = mlp_model.predict(X_test)

# Evaluate the model
accuracy_mlp = accuracy_score(y_test, y_pred_mlp)
conf_matrix_mlp = confusion_matrix(y_test, y_pred_mlp)
```

```python

print("Accuracy (MLP):", accuracy_mlp)
print("Confusion Matrix (MLP):\n", conf_matrix_mlp)

# Visualize the confusion matrix
plt.figure(figsize=(6, 5))
sns.heatmap(conf_matrix_mlp, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Class 0', 'Class 1'],
            yticklabels=['Class 0', 'Class 1'])
plt.title('Confusion Matrix - MLP')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()

# Visualize the accuracy
plt.figure(figsize=(5, 4))
plt.bar(['MLP'], [accuracy_mlp], color='lightcoral')
plt.title('Accuracy - MLP')
plt.ylim(0, 1)
plt.ylabel('Accuracy')
plt.show()
```

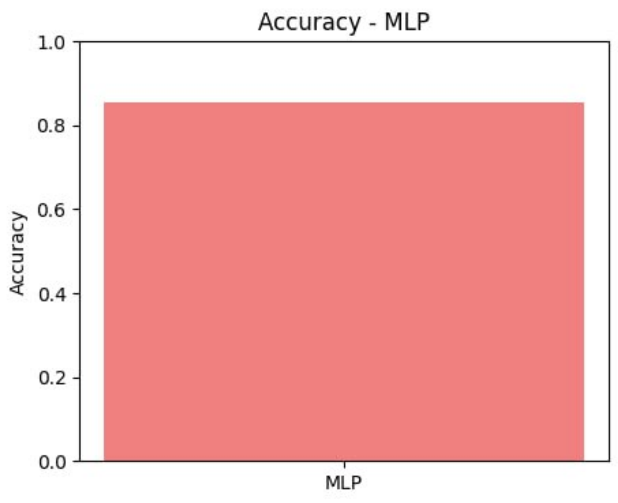


Figure 3.3: MLP Accuracy

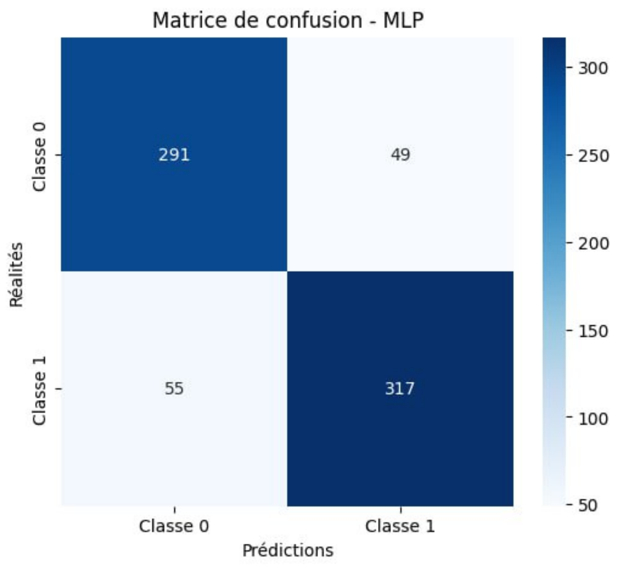


Figure 3.4: MLP matrix of confusion

CHAPTER 4

ENSEMBLE LEARNING FOR CLASSIFICATION

4.1 Introduction to Ensemble Learning

Ensemble learning is a powerful technique that combines multiple individual models to create a stronger model for making predictions. The idea is to leverage the strengths of various models, which may have different strengths and weaknesses, to improve overall accuracy and robustness.

In this chapter, we will apply various ensemble techniques such as voting, stacking, bagging, boosting, and random forests to solve a classification problem. The goal is to demonstrate how these methods can enhance the accuracy and robustness of classification models.

4.2 Voting Classifier

The Voting Classifier is an ensemble method that combines multiple machine learning models to make a final prediction based on a majority vote from each individual model. In this section, we implement a Voting Classifier using three base models: Logistic Regression, Random Forest, and Multi-Layer Perceptron (MLP). These models are trained on the preprocessed dataset, and their predictions are aggregated to determine the final classification.

- Define Base Models: We define three base models — Logistic Regression, Random Forest, and MLP Classifier — as the individual classifiers in the ensemble.

- Create Voting Classifier: A hard voting mechanism is used, meaning the majority vote from the individual classifiers determines the final prediction.

- Train the Model: The Voting Classifier is trained on the training set.

- Predictions: The trained Voting Classifier is used to make predictions on the test set.

- Evaluate the Model: The accuracy of the model is calculated and visualized **88.90%**, along with the confusion matrix to assess classification performance.

```python
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Define the base models
log_reg = LogisticRegression(max_iter=1000)
rf_model = RandomForestClassifier(random_state=42)
mlp_model = MLPClassifier(
    hidden_layer_sizes=(100,), random_state=42)

# Create the Voting Classifier model
voting_model = VotingClassifier(
    estimators=[('log_reg', log_reg),
                ('rf_model', rf_model),
                ('mlp_model', mlp_model)],
    voting='hard')  # Hard voting for majority classification

# Train the model
voting_model.fit(X_train, y_train)

# Predictions
y_pred_voting = voting_model.predict(X_test)

# Evaluation
accuracy_voting = accuracy_score(y_test, y_pred_voting)
conf_matrix_voting = confusion_matrix(y_test, y_pred_voting)

print("Accuracy (Voting):", accuracy_voting)
print("Confusion Matrix (Voting):\n", conf_matrix_voting)

# Visualize the confusion matrix
plt.figure(figsize=(6, 5))
sns.heatmap(conf_matrix_voting, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Class 0', 'Class 1'],
            yticklabels=['Class 0', 'Class 1'])
```

```
plt.title('Confusion Matrix - Voting Classifier')
plt.xlabel('Predictions')
plt.ylabel('True Values')
plt.show()

# Visualize the accuracy
plt.figure(figsize=(5, 4))
plt.bar(['Voting Classifier'], [accuracy_voting],
        color='lightgreen')
plt.title('Accuracy - Voting Classifier')
plt.ylim(0, 1)
plt.ylabel('Accuracy')
plt.show()
```

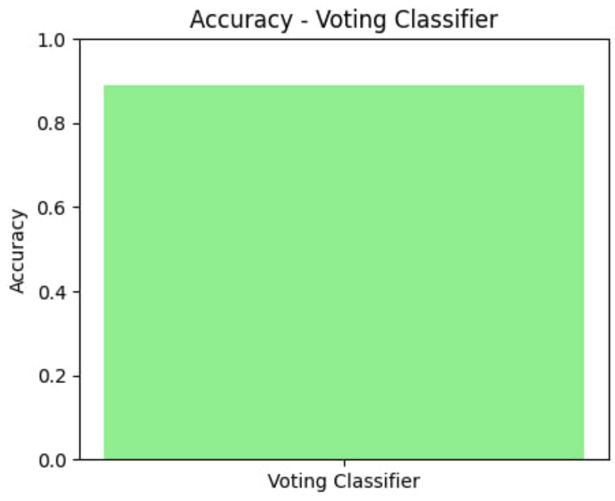


Figure 4.1: Voting Accuracy

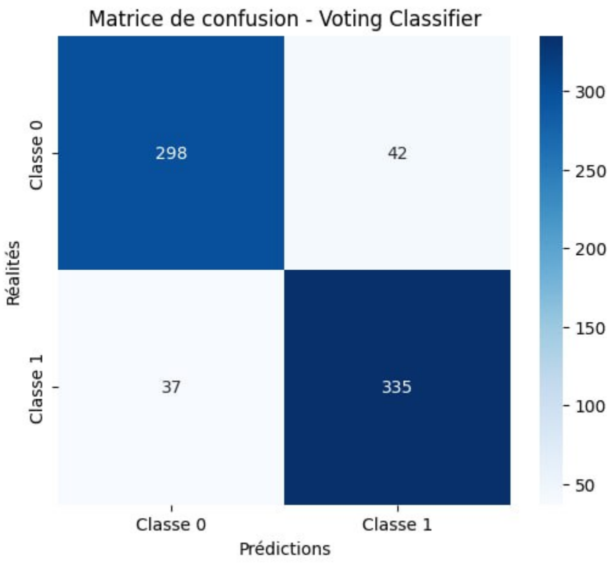


Figure 4.2: Voting matrix of confusion

4.3 Stacking Classifier

The Stacking Classifier is an ensemble learning method that combines multiple base models and uses another model (known as the meta-model) to make the final prediction based on the outputs of the base models. In this section, we implement a Stacking Classifier using three base models: Logistic Regression, Random Forest, and Multi-Layer Perceptron (MLP). The predictions from these base models are then combined using a Logistic Regression meta-model to provide the final classification.

- Define Base Models: We define three base models — Logistic Regression, Random Forest, and MLP Classifier — as the individual classifiers in the ensemble.
- Create Stacking Classifier: A Logistic Regression model is used as the meta-model that aggregates the predictions from the base models.
- Train the Model: The Stacking Classifier is trained on the training set.
- Predictions: The trained Stacking Classifier is used to make predictions on the test set.
- Evaluate the Model: The accuracy of the model is calculated and visualized **98.88%**, along with the confusion matrix to assess classification performance.

```python
from sklearn.ensemble import StackingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Define the base models
base_learners = [
    ('log_reg', LogisticRegression(max_iter=1000)),
    ('rf_model', RandomForestClassifier(random_state=42)),
    ('mlp_model', MLPClassifier(
        hidden_layer_sizes=(100,), random_state=42))
]

# Create the Stacking Classifier model with Logistic Regression as meta-model
stacking_model = StackingClassifier(
    estimators=base_learners, final_estimator=LogisticRegression())

```

```python
# Train the model
stacking_model.fit(X_train, y_train)

# Predictions
y_pred_stacking = stacking_model.predict(X_test)

# Evaluation
accuracy_stacking = accuracy_score(y_test, y_pred_stacking)
conf_matrix_stacking = confusion_matrix(y_test, y_pred_stacking)

print("Accuracy (Stacking):", accuracy_stacking)
print("Confusion Matrix (Stacking):\n", conf_matrix_stacking)

# Visualize the confusion matrix
plt.figure(figsize=(6, 5))
sns.heatmap(conf_matrix_stacking, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Class 0', 'Class 1'],
            yticklabels=['Class 0', 'Class 1'])
plt.title('Confusion Matrix - Stacking Classifier')
plt.xlabel('Predictions')
plt.ylabel('True Values')
plt.show()

# Visualize the accuracy
plt.figure(figsize=(5, 4))
plt.bar(['Stacking Classifier'], [accuracy_stacking],
        color='lightblue')
plt.title('Accuracy - Stacking Classifier')
plt.ylim(0, 1)
plt.ylabel('Accuracy')
plt.show()
```

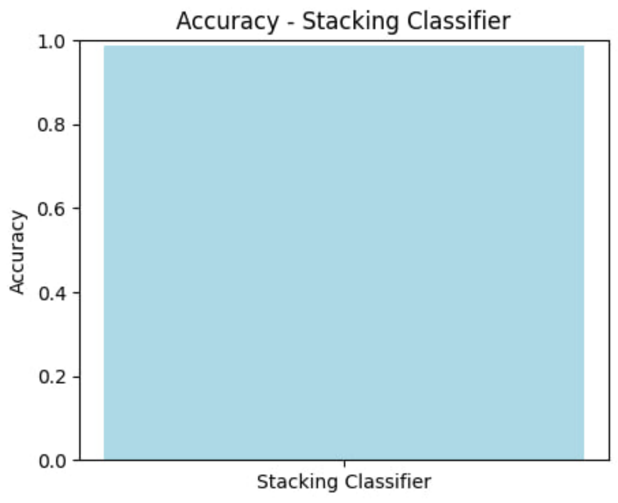


Figure 4.3: Stacking accuracy

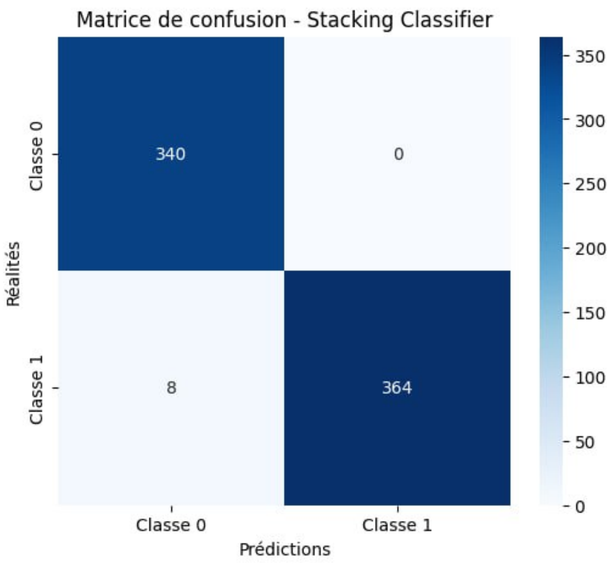


Figure 4.4: Stacking matrix of confusion

4.4 Bagging Classifier

Bagging, short for Bootstrap Aggregating, is an ensemble method that creates multiple subsets of the training data using bootstrapping (random sampling with replacement). Each subset is used to train a base model, and their predictions are aggregated, usually by voting, to make the final prediction. In this section, we implement a Bagging Classifier using a Random Forest as the base estimator.

- Define Base Model: We define a RandomForestClassifier as the base estimator for the Bagging Classifier.
- Create Bagging Classifier: A Bagging Classifier is created with the RandomForestClassifier as the base model.
- Train the Model: The Bagging Classifier is trained on the training set.
- Predictions: The trained Bagging Classifier is used to make predictions on the test set.
- Evaluate the Model: The accuracy of the model is calculated and visualized **99.44%**, along with the confusion matrix to assess classification performance.

```python
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Initialize the Bagging model with RandomForestClassifier as base estimator
bagging_model = BaggingClassifier(
    estimator=RandomForestClassifier(), random_state=42
)

# Train the model
bagging_model.fit(X_train, y_train)

# Predictions
y_pred_bagging = bagging_model.predict(X_test)

# Evaluation
accuracy_bagging = accuracy_score(y_test, y_pred_bagging)
conf_matrix_bagging = confusion_matrix(y_test, y_pred_bagging)

print("Accuracy (Bagging):", accuracy_bagging)
print("Confusion Matrix (Bagging):\n", conf_matrix_bagging)

# Visualize the confusion matrix
plt.figure(figsize=(6, 5))
sns.heatmap(
    conf_matrix_bagging, annot=True, fmt='d', cmap='Blues',
    xticklabels=['Class 0', 'Class 1'],
    yticklabels=['Class 0', 'Class 1']
)
plt.title('Confusion Matrix - Bagging Classifier')
plt.xlabel('Predictions')
plt.ylabel('True Values')
plt.show()

# Visualize the accuracy
plt.figure(figsize=(5, 4))
plt.bar(['Bagging Classifier'], [accuracy_bagging], color='lightgreen')
plt.title('Accuracy - Bagging Classifier')
plt.ylim(0, 1)
plt.ylabel('Accuracy')
plt.show()
```

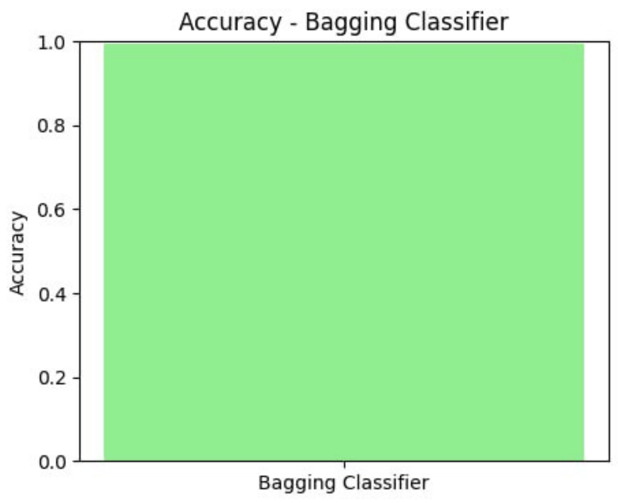


Figure 4.5: Bagging accuracy

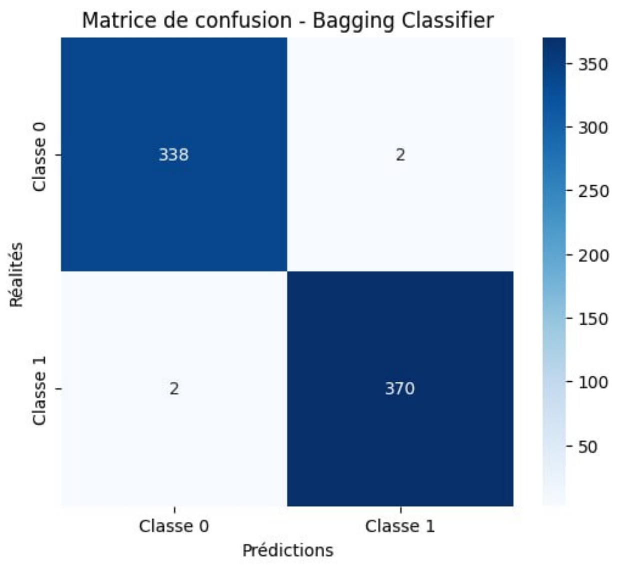


Figure 4.6: Bagging matrix of confusion

4.5 Boosting Classifier

Boosting is an ensemble technique that combines multiple weak models to create a strong model. In this section, we implement a Boosting Classifier using Gradient Boosting.

- Define Base Model: We define a GradientBoostingClassifier as the base model for the boosting ensemble.
- Train the Model: The Boosting Classifier is trained on the training set.
- Predictions: The trained Boosting Classifier is used to make predictions on the test set.
- Evaluate the Model: The accuracy of the model is calculated and visualized **98.03%**, along with the confusion matrix to assess classification performance.

```python
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Initialize the Boosting model
boosting_model = GradientBoostingClassifier(random_state=42)

# Train the model
boosting_model.fit(X_train, y_train)

# Predictions
y_pred_boosting = boosting_model.predict(X_test)

# Evaluation
accuracy_boosting = accuracy_score(y_test, y_pred_boosting)
conf_matrix_boosting = confusion_matrix(y_test, y_pred_boosting)

print("Accuracy (Boosting):", accuracy_boosting)
print("Confusion Matrix (Boosting):\n", conf_matrix_boosting)

# Visualize the confusion matrix
plt.figure(figsize=(6, 5))
sns.heatmap(
    conf_matrix_boosting, annot=True, fmt='d', cmap='Blues',
    xticklabels=['Class 0', 'Class 1'],
    yticklabels=['Class 0', 'Class 1']
)
plt.title('Confusion Matrix - Boosting Classifier')
plt.xlabel('Predictions')
plt.ylabel('True Values')
plt.show()

# Visualize the accuracy
plt.figure(figsize=(5, 4))
plt.bar(['Boosting Classifier'], [accuracy_boosting], color='lightcoral')
plt.title('Accuracy - Boosting Classifier')
plt.ylim(0, 1)
plt.ylabel('Accuracy')
plt.show()
```

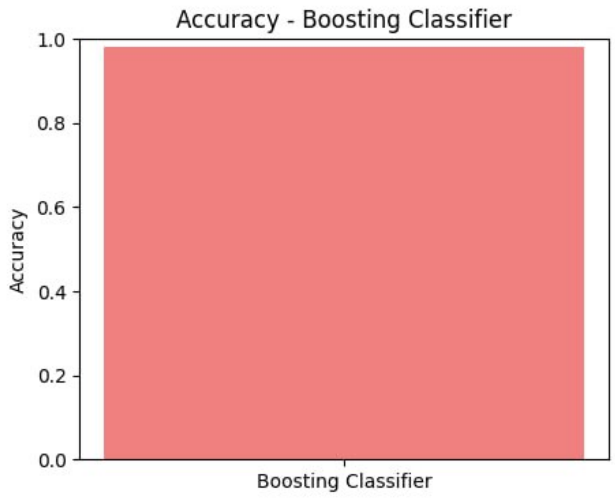


Figure 4.7: Boosting accuracy

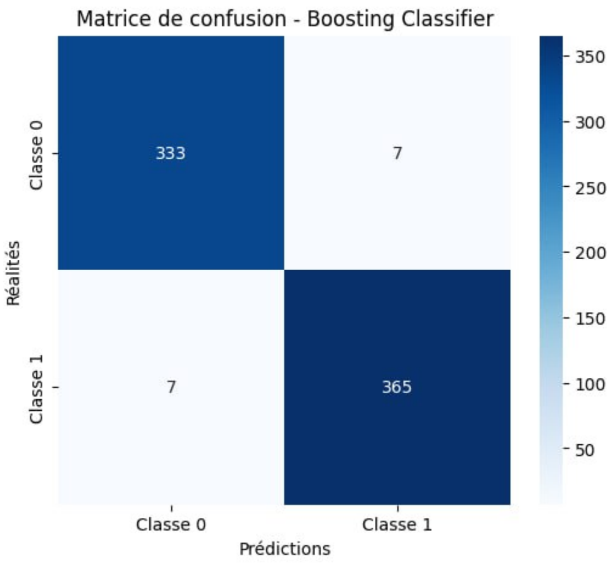


Figure 4.8: Boosting matrix of confusion

4.6 Boost Classifier

AdaBoost, or Adaptive Boosting, is an ensemble method that combines weak learners (e.g., decision trees) into a strong model. In this section, we implement an AdaBoost Classifier with Decision Trees as the base model.

- Define Base Model: We define a shallow DecisionTreeClassifier (with a max depth of 1) as the base model for AdaBoost.
- Train the Model: The AdaBoost model is trained using the base model and the training data.
- Predictions: The trained AdaBoost Classifier is used to make predictions on the test set.
- Evaluate the Model: The accuracy of the model is calculated and visualized **92.69%**, along with the confusion matrix to assess classification performance.

```python
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Initialize the Boosting model (AdaBoost)
base_model = DecisionTreeClassifier(max_depth=1)  # Shallow decision trees
boost_model = AdaBoostClassifier(
    estimator=base_model, n_estimators=50, random_state=42
)

# Train the model
boost_model.fit(X_train, y_train)

# Predictions
y_pred_boost = boost_model.predict(X_test)

# Evaluation
accuracy_boost = accuracy_score(y_test, y_pred_boost)
conf_matrix_boost = confusion_matrix(y_test, y_pred_boost)

print("Accuracy (Boost):", accuracy_boost)
print("Confusion Matrix (Boost):\n", conf_matrix_boost)

```

```python
# Visualize the confusion matrix
plt.figure(figsize=(6, 5))
sns.heatmap(
    conf_matrix_boost, annot=True, fmt='d', cmap='Blues',
    xticklabels=['Class 0', 'Class 1'],
    yticklabels=['Class 0', 'Class 1']
)
plt.title('Confusion Matrix - AdaBoost Classifier')
plt.xlabel('Predictions')
plt.ylabel('True Values')
plt.show()

# Visualize the accuracy
plt.figure(figsize=(5, 4))
plt.bar(['AdaBoost Classifier'], [accuracy_boost], color='lightseagreen')
plt.title('Accuracy - AdaBoost Classifier')
plt.ylim(0, 1)
plt.ylabel('Accuracy')
plt.show()
```

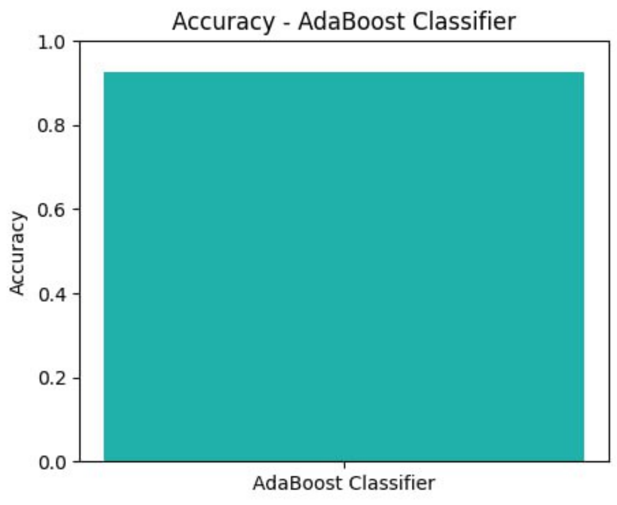


Figure 4.9: Boost accuracy

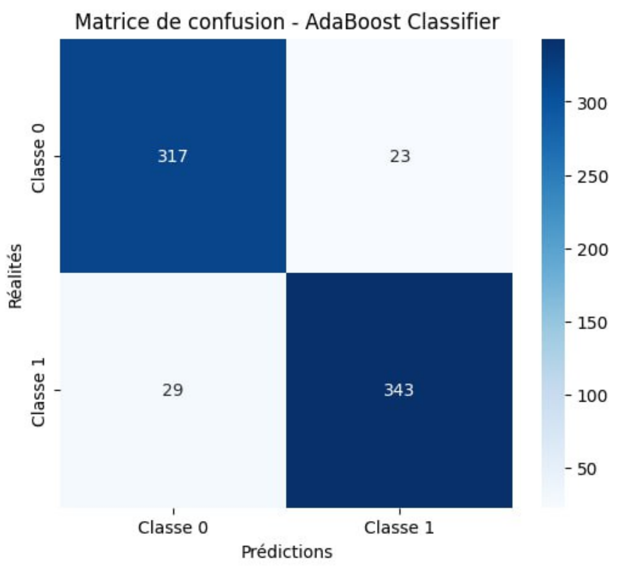


Figure 4.10: Boost matrix of confusion

4.7 Random Forest Classifier

Random Forest is an ensemble learning method that builds multiple decision trees and merges them together to improve accuracy and reduce overfitting. In this section, we implement a Random Forest Classifier.

- Initialize Model: A RandomForestClassifier is created with a fixed random seed for reproducibility.
- Train the Model: The Random Forest model is trained using the training data.
- Predictions: The trained Random Forest model is used to make predictions on the test set.
- Evaluate the Model: The accuracy of the model is calculated and visualized **99.58%**, along with the confusion matrix to assess classification performance.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Initialize the Random Forest model
rf_model = RandomForestClassifier(random_state=42)

# Train the model
rf_model.fit(X_train, y_train)

# Predictions
y_pred_rf = rf_model.predict(X_test)

# Evaluation
accuracy_rf = accuracy_score(y_test, y_pred_rf)
conf_matrix_rf = confusion_matrix(y_test, y_pred_rf)

print("Accuracy (Random Forest):", accuracy_rf)
print("Confusion Matrix (Random Forest):\n", conf_matrix_rf)

# Visualize the confusion matrix
plt.figure(figsize=(6, 5))
sns.heatmap(
    conf_matrix_rf, annot=True, fmt='d', cmap='Blues',
    xticklabels=['Class 0', 'Class 1'],
    yticklabels=['Class 0', 'Class 1']
)
plt.title('Confusion Matrix - Random Forest Classifier')
plt.xlabel('Predictions')
plt.ylabel('True Values')
plt.show()

# Visualize the accuracy
plt.figure(figsize=(5, 4))
plt.bar(['Random Forest Classifier'], [accuracy_rf], color='lightseagreen')
plt.title('Accuracy - Random Forest Classifier')
plt.ylim(0, 1)
plt.ylabel('Accuracy')
plt.show()
```

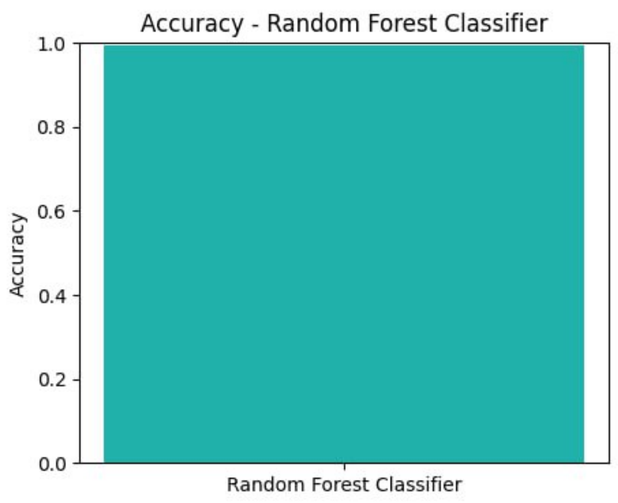


Figure 4.11: Random Forest accuracy

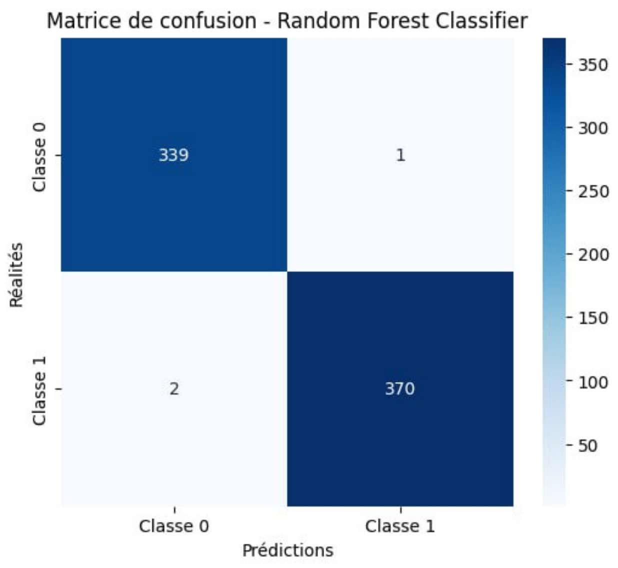


Figure 4.12: Random Forest matrix of confusion

CHAPTER 5 CROSS VALIDATION AND MODELS EVALUATION

5.1 Introduction to Model Evaluation

In this chapter, we aim to evaluate several machine learning models to assess their performance using two key techniques: cross-validation and the ROC curve. Cross-validation is an essential technique that provides a more reliable estimate of a model's performance by using different subsets of the data for training and testing. The ROC curve, on the other hand, helps visualize the trade-off between the true positive rate and false positive rate and is particularly useful in binary classification tasks.

This chapter covers:

- The implementation of cross-validation for model evaluation.
- The calculation and interpretation of ROC curves and AUC scores.
- A comparison of different models based on these evaluation methods.

The following sections will detail the results from both cross-validation and ROC analysis, along with conclusions drawn from these results.

5.2 Cross Validation

Cross-validation is an essential method for assessing the performance of a machine learning model. It involves dividing the dataset into multiple subsets and using some of them for training and others for testing. This helps to mitigate overfitting and gives a more reliable estimate of the model's performance. In this section, we apply cross-validation to evaluate several models.

- Initialize Models: We define a list of models to evaluate, including Logistic Regression, Random Forest, MLP, and others.

- Cross-Validation: We perform cross-validation using StratifiedKFold to ensure that each fold has a proportional representation of the target classes.
- Results Visualization: We display the mean and standard deviation of the accuracy scores from cross-validation for each model.

The cross-validation process allows us to observe how well each model generalizes to unseen data, minimizing bias and variance. Based on the results, we can conclude which models perform consistently well across different subsets of the dataset.

```python
import matplotlib.pyplot as plt
from sklearn.model_selection import cross_val_score, StratifiedKFold
import numpy as np

# Fonction pour effectuer la validation croisée
def cross_val_evaluation(model, X, y, cv_folds=5):
    cv = StratifiedKFold(n_splits=cv_folds, shuffle=True, random_state=42)
    try:
        cv_scores = cross_val_score(model, X, y, cv=cv, scoring='accuracy')
        mean_score = np.mean(cv_scores)
        std_score = np.std(cv_scores)
    except Exception as e:
        print(f"Erreur lors de la validation croisée pour {model}: {e}")
        mean_score = np.nan
        std_score = np.nan
    return mean_score, std_score

# Liste des modèles à évaluer
models = [
    ('Logistic Regression', log_reg),
    ('Random Forest', rf_model),
    ('MLP', mlp_model),
    ('Voting Classifier', voting_model),
    ('Stacking Classifier', stacking_model),
    ('Bagging Classifier', bagging_model),
    ('Boosting Classifier', boosting_model),
    ('Boost Model (AdaBoost)', boost_model),
    ('XGBoost', xgb_model)
]

model_names = []
mean_scores = []
```

```python
std_scores = []

# Appliquer la validation croisée pour chaque modèle
for name, model in models:
    mean_score, std_score = cross_val_evaluation(model, X, y)
    model_names.append(name)
    mean_scores.append(mean_score)
    std_scores.append(std_score)

# Création du plot
plt.figure(figsize=(10, 6))
plt.barh(model_names, mean_scores, xerr=std_scores, capsize=5, color='skyblue',
edgecolor='black')

plt.xlabel('Accuracy')
plt.title('Cross-Validation Accuracy of Different Models')
plt.grid(True, axis='x', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```

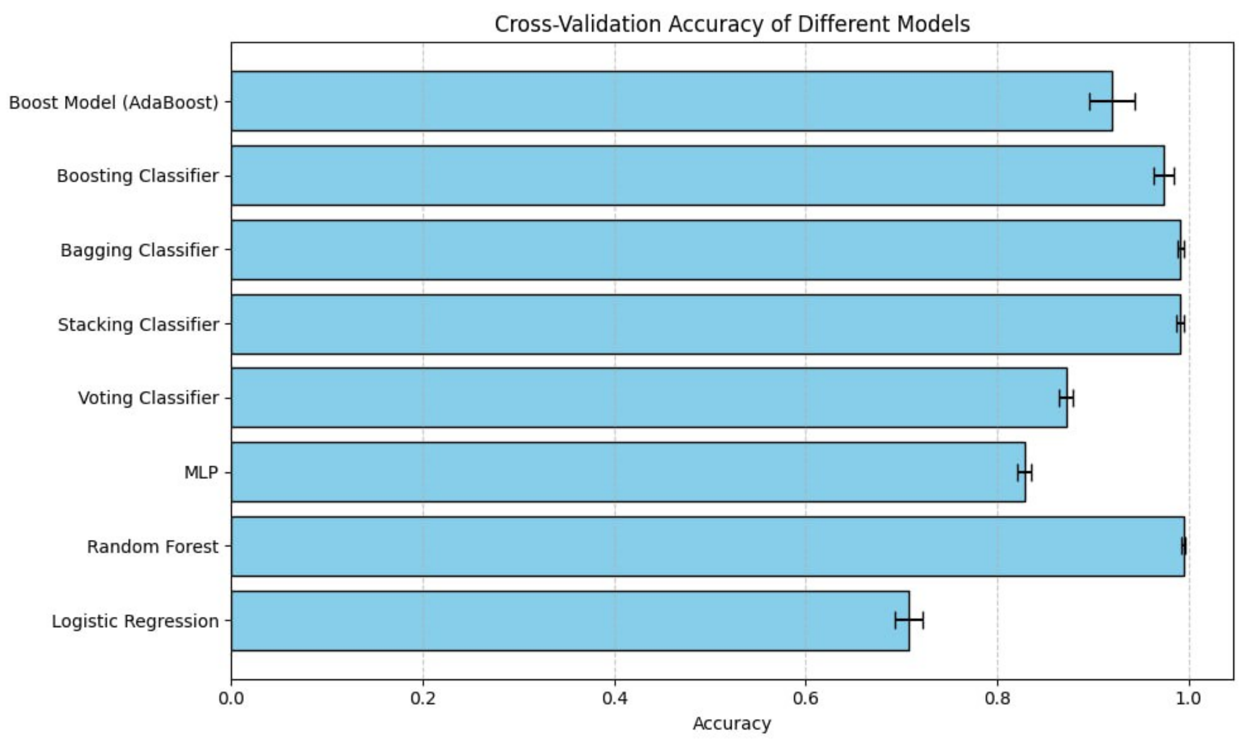


Figure 5.1: Cross validation accuracy of different models

Conclusion from Cross-Validation Results:

- The Random Forest model achieved the highest mean accuracy and is among the most reliable models for this dataset.
- Logistic Regression and MLP showed slightly lower accuracy but still performed reasonably well.

5.3 Models Evaluation

In this section, we evaluate the models using ROC curves and the AUC score, which are essential metrics for assessing the performance of classifiers. The ROC curve helps visualize the trade-off between the true positive rate and false positive rate. The AUC score summarizes this performance, where a higher score indicates better model performance.

- ROC Curve: We plot the ROC curve for each model to visualize its performance.
- AUC Score: We calculate and display the AUC score for each model.
- Visualization: The ROC curves and AUC scores for different models are visualized in a single plot.

The ROC analysis allows us to assess the overall classification performance of each model and compare how well each model distinguishes between the positive and negative classes.

```python
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt
from xgboost import XGBClassifier

# Function to plot ROC curve
def plot_roc_curve(model, X_test, y_test, label):
    try:
        if hasattr(model, "predict_proba"):
            y_pred = model.predict_proba(X_test)[:, 1]
        else:
            y_pred = model.predict(X_test)
            y_pred = (y_pred == 1).astype(int)

        fpr, tpr, _ = roc_curve(y_test, y_pred)
        auc_score = auc(fpr, tpr)
        plt.plot(fpr, tpr, label=f'{label} (AUC = {auc_score:.2f})')
    except Exception as e:
        print(f"Error with model {label}: {e}")
```

```

# Train the XGBoost model
xgb_model = XGBClassifier(objective='binary:logistic', random_state=42)
xgb_model.fit(X_train, y_train)

# Plot ROC for each model
plt.figure(figsize=(10, 8))
plot_roc_curve(voting_model, X_test, y_test, "Voting Classifier")
plot_roc_curve(stacking_model, X_test, y_test, "Stacking Classifier")
plot_roc_curve(bagging_model, X_test, y_test, "Bagging Classifier")
plot_roc_curve(boosting_model, X_test, y_test, "Boosting Classifier")
plot_roc_curve(xgb_model, X_test, y_test, "XGBoost")
plot_roc_curve(rf_model, X_test, y_test, "Random Forest")

plt.plot([0, 1], [0, 1], linestyle='--', color='gray')

plt.title("ROC Curve for Different Models")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.legend(loc="best")
plt.show()
```

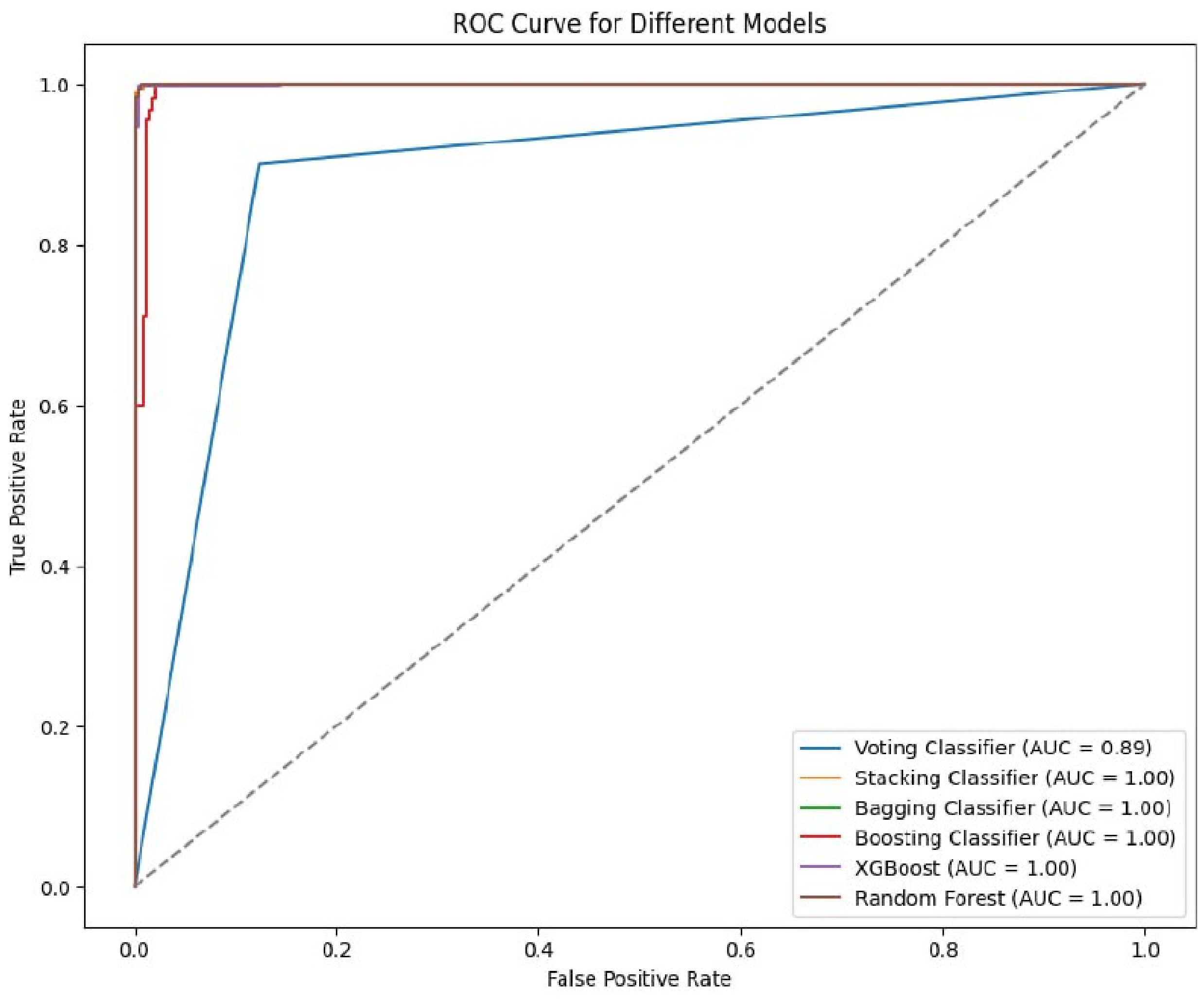


Figure 5.2: ROC curve

Conclusion from ROC Curve Analysis:

- The Random Forest and XGBoost models showed the best AUC scores, indicating their superior ability to distinguish between the two classes.
- The Voting and Stacking classifiers also performed well, with AUC scores above 0.9.
- Logistic Regression showed lower AUC scores compared to the ensemble models, but still provided reasonable separation between classes.

CHAPTER 6 CONCLUSION

To conclude, we can say that model evaluation is a crucial step in machine learning, enabling us to assess the performance of different algorithms and select the best ones for our tasks. Cross-validation and the ROC curve are two essential techniques that help us understand how well our models perform in terms of generalization and classification accuracy.

By applying cross-validation, we are able to obtain a more reliable estimate of model performance, reducing the risk of overfitting and ensuring that the model can generalize well to unseen data. The ROC curve and AUC score, on the other hand, provide a detailed visualization of a model's ability to distinguish between classes, which is particularly useful for evaluating binary classification tasks.

Through this analysis, we found that ensemble methods like Random Forest and XGBoost performed exceptionally well, showing high accuracy and strong AUC scores. Other models, such as Logistic Regression and MLP, also demonstrated competitive performance, though they were outperformed by the ensemble approaches.

In the end, model evaluation not only helps us choose the right model for a given task but also provides deeper insights into how well a model is likely to perform in real-world applications. By using these techniques, organizations can make more informed decisions, ensuring that their machine learning systems are reliable, efficient, and effective in solving complex problems.