Fraud Transaction Detection Project

Overview

This project focuses on detecting fraudulent transactions using machine learning techniques. Fraud detection is a critical issue in financial transactions, and building an accurate model can help reduce losses for financial institutions. In this project, we use a real dataset containing information about credit card transactions and apply Random Forest, along with feature scaling, class imbalance handling, and model evaluation techniques.

Libraries Used

We utilize the following Python libraries:

- **pandas**: For data loading and manipulation.
- **scikit-learn (sklearn)**: For machine learning algorithms and data preprocessing.
- **imblearn**: To handle imbalanced datasets using Synthetic Minority Over-sampling Technique (SMOTE).
- **matplotlib & seaborn**: For data visualization.

The dataset used is a credit card transactions dataset with highly imbalanced classes, making it a suitable case for applying techniques to handle imbalance and improve model performance.

---

1. Loading the Dataset

We first import the necessary libraries and load the dataset using pandas. The dataset is stored in CSV format, and we use `pd.read_csv()` to load it into a pandas DataFrame. We also inspect the data using `info()` and `describe()` functions to understand its structure.

```
python
Copy code
import pandas as pd

# Define the path to the dataset
data_path = r"path_to_your_dataset/creditcard.csv"

# Load the dataset
data = pd.read_csv(data_path)

# Display basic information about the dataset
print(data.info())
print(data.describe())
```

Explanation:

- `data.info()` gives us insights into the data types, non-null values, and memory usage.

- `data.describe()` provides statistical summaries, such as mean, standard deviation, and percentiles for numeric features.

---

2. Data Preprocessing

Before applying machine learning algorithms, we preprocess the data:

1. **Converting the 'Class' column to integer**: This ensures the target column is in the correct format.
2. **Handling Missing Values**: We check for missing values and ensure they are properly handled.
3. **Separating Features and Target**: We split the dataset into `X` (features) and `y` (target) where 'Class' is the target, indicating whether a transaction is fraudulent (1) or not (0).

```
python
Copy code
# Convert 'Class' column to integer if needed
data['Class'] = data['Class'].astype(int)

# Handle missing values if any
print(data.isnull().sum())

# Separate features and target variable
X = data.drop(['Class'], axis=1)
y = data['Class']
```

Explanation:

- `data['Class'].astype(int)` ensures that the 'Class' column is in integer format.
- `X = data.drop(['Class'], axis=1)` removes the 'Class' column from `X`, which will be used as input features.
- `y = data['Class']` assigns the target variable (fraudulent or non-fraudulent) to `y`.

Next, we split the data into training and testing sets using `train_test_split` from `sklearn.model_selection`.

```
python
Copy code
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

Explanation:

- We split the dataset with an 80-20 ratio: 80% of the data for training and 20% for testing. The `random_state` ensures reproducibility.

---

3. Feature Scaling

Credit card transaction data often contain variables with different scales. To standardize the feature set, we use `StandardScaler` to transform the data so that each feature has a mean of 0 and a standard deviation of 1.

```
python
Copy code
from sklearn.preprocessing import StandardScaler

# Initialize the scaler
scaler = StandardScaler()

# Fit and transform the training data, transform the test data
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Explanation:

- `scaler.fit_transform(X_train)` learns the scaling parameters from the training set and applies the transformation.
- `scaler.transform(X_test)` uses the learned parameters to transform the test set. This ensures that both training and testing sets are scaled consistently.

---

4. Handling Class Imbalance with SMOTE

The dataset is highly imbalanced, with fraudulent transactions being far fewer than non-fraudulent ones. To handle this, we use **Synthetic Minority Over-sampling Technique (SMOTE)**, which creates synthetic samples for the minority class to balance the dataset.

```
python
Copy code
from imblearn.over_sampling import SMOTE

# Initialize SMOTE
smote = SMOTE(random_state=42)

# Apply SMOTE to the scaled training data
X_train_res, y_train_res = smote.fit_resample(X_train_scaled, y_train)
```

Explanation:

- **SMOTE** generates new instances of the minority class (fraud) by creating synthetic data points, ensuring the model doesn't learn to be biased toward the majority class.

---

5. Model Training

We use a **Random Forest Classifier** to train our model. Random Forest is an ensemble learning method that builds multiple decision trees and merges them together for a more accurate and stable prediction.

```
python
Copy code
from sklearn.ensemble import RandomForestClassifier
from imblearn.pipeline import Pipeline

# Initialize the RandomForestClassifier
model = RandomForestClassifier(random_state=42)

# Create a pipeline
pipeline = Pipeline([
    ('classifier', model)
])

# Fit the model
pipeline.fit(X_train_res, y_train_res)

# Print a message indicating the model has been trained
print("Model training complete.")
```

Explanation:

- **Pipeline** is used to chain steps together, but in this case, we are only using the Random Forest classifier.
- **RandomForestClassifier** is used because of its robustness, ability to handle large datasets, and its feature importance capability.

---

6. Model Evaluation

Once the model is trained, we evaluate its performance on the test set. The key metrics used include the confusion matrix, classification report, and accuracy score.

```
python
Copy code
from sklearn.metrics import classification_report, confusion_matrix

# Make predictions on the test set
y_pred = pipeline.predict(X_test_scaled)

# Evaluate the model
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

Explanation:

- The **confusion matrix** helps in visualizing the performance by comparing the predicted and actual labels.

- The **classification report** provides metrics like precision, recall, f1-score, and accuracy for both classes (fraud and non-fraud).

---

7. Feature Importance

Random Forests provide an easy way to measure the importance of each feature in making predictions. We visualize this using a horizontal bar plot.

```
python
Copy code
import matplotlib.pyplot as plt
import numpy as np

# Extract feature importances
importances = pipeline.named_steps['classifier'].feature_importances_
indices = np.argsort(importances)

# Plot feature importances
plt.figure(figsize=(10, 6))
plt.title('Feature Importances')
plt.barh(range(len(indices)), importances[indices], align='center')
plt.yticks(range(len(indices)), X.columns[indices])
plt.xlabel('Feature Importance')
plt.show()
```

Explanation:

- Feature importance tells us which variables had the most influence in predicting fraudulent transactions, helping in understanding the model's behavior.

---

8. Plotting ROC and Precision-Recall Curves

We use **ROC Curve** and **Precision-Recall Curve** to evaluate the performance of the classifier.

ROC Curve

```
python
Copy code
from sklearn.metrics import roc_curve, auc

# Compute ROC curve
fpr, tpr, thresholds = roc_curve(y_test,
pipeline.predict_proba(X_test_scaled)[:, 1])
roc_auc = auc(fpr, tpr)

# Plot ROC curve
plt.figure(figsize=(10, 6))
plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (area =
{roc_auc:.2f})')
```

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

Precision-Recall Curve

```
python
Copy code
from sklearn.metrics import precision_recall_curve

# Compute Precision-Recall curve
precision, recall, _ = precision_recall_curve(y_test,
pipeline.predict_proba(X_test_scaled)[:, 1])

# Plot Precision-Recall curve
plt.figure(figsize=(10, 6))
plt.plot(recall, precision, color='green', lw=2)
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.show()
```

Explanation:

- **ROC Curve** plots the true positive rate (TPR) vs. false positive rate (FPR), with the **AUC** representing the area under the curve, which summarizes the model's performance.
- **Precision-Recall Curve** focuses on the trade-off between precision and recall, which is particularly useful in imbalanced datasets like fraud detection.

---

9. Confusion Matrix Visualization

We visualize the confusion matrix using a heatmap for better interpretation.

```
python
Copy code
import seaborn as sns
from sklearn.metrics import confusion_matrix

# Generate confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Plot confusion matrix heatmap
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=["Non-
Fraud", "Fraud"], yticklabels=["Non-Fraud", "Fraud"])
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Confusion Matrix')
plt.show()
```

Explanation:

- **Confusion Matrix Heatmap** visually shows how many fraudulent transactions were correctly and incorrectly classified, allowing for an intuitive understanding of the classifier’s performance.

Conclusion

This project successfully applies a Random Forest Classifier to detect fraudulent transactions. By handling class imbalance with SMOTE and using feature scaling, we improved model performance. Furthermore, the evaluation through ROC, precision-recall curves, and feature importance helped us understand the model's effectiveness and behavior.