Salman Irfan

2024-Phd-Cs-5

CS-608 Advanced Techniques in Data Science

Course Project: Credit Card Fraud Detection

Submitted To

Prof. Dr. Muhammad Awais Hassan

1. Select a Project Domain and Area: -

Domain: Financial services (Credit Card Transactions).

Area: Fraud detection in credit card transactions using machine learning.

2. Identify the Problem: -

Credit card companies need to detect fraudulent transactions with high accuracy while minimizing the number of false positives (legitimate transactions marked as fraud).

3. Ask Questions to Answer: -

i. Which transaction patterns or behaviors typically indicate fraud?

- **Why Ask**: Fraudulent transactions often follow certain unusual patterns, such as unusual spending amounts or multiple transactions within a short time frame.
- **Expected Insight**: Identify behaviors like high transaction volumes in a short time, geographical anomalies, or sudden changes in spending patterns.

ii. What role does the transaction time play in identifying fraud?

- **Why Ask:** Fraud may occur during specific hours when detection is harder (e.g., late-night hours).
- **Expected Insight**: Certain times of day may show higher fraud rates, which could inform temporal-based fraud detection methods.

iii. Are there specific amounts of transactions that are more prone to fraud?

- **Why Ask**: Fraudsters might target small amounts to evade detection or large amounts for bigger payouts.
- **Expected Insight:** By analyzing transaction amounts, we can determine if certain ranges are more susceptible to fraud.

iv. How can we detect fraud without overwhelming the system with false positives?

- **Why Ask**: False positives (legitimate transactions marked as fraud) can frustrate customers. We need to balance fraud detection with accuracy.

- **Expected Insight**: Use techniques like precision-recall trade-offs or resampling methods to improve model performance on rare events like fraud.

4. Collect or Find Data: -

- **Data Source:** Kaggle's credit card fraud detection dataset.
- **Source:** https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud?select=creditcard.csv

5. Data Wrangling (Preprocessing): -

```
# 2. Data Wrangling (Preprocessing)
# Check for missing values
print("Check for missing values")
print(df.isnull().sum())
```

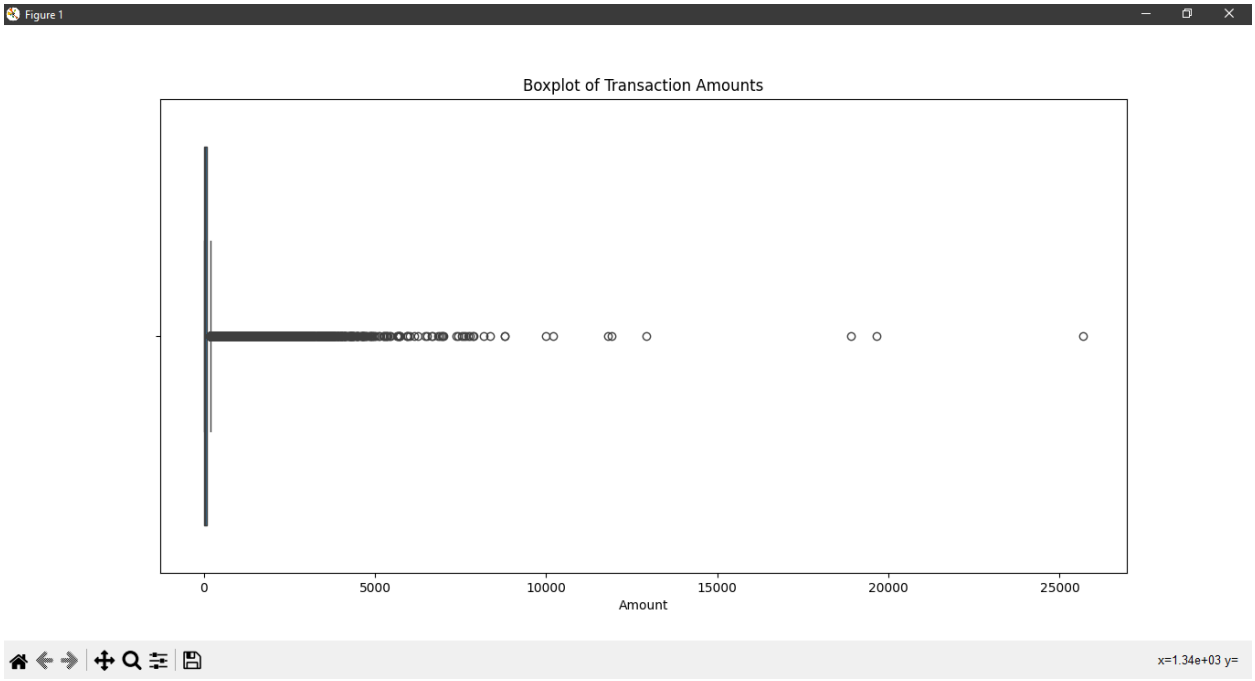
Output: -

```
Check for missing values
Time      0
V1        0
V2        0
V3        0
V4        0
V5        0
V6        0
V7        0
V8        0
V9        0
V10       0
V11       0
V12       0
```

```
V21       0
V22       0
V23       0
...
V28       0
Amount    0
Class     0
dtype: int64
```

Handle Outliers: -

```
# handle Outliers
print("handle outliers")
# Analyze 'Amount' for outliers using a boxplot
plt.figure(figsize=(10, 5))
sns.boxplot(x=df["Amount"])
plt.title("Boxplot of Transaction Amounts")
plt.show()
```



Outliers are important because they depict fraudulent transaction.

6. EDA (Exploratory Data Analysis): -

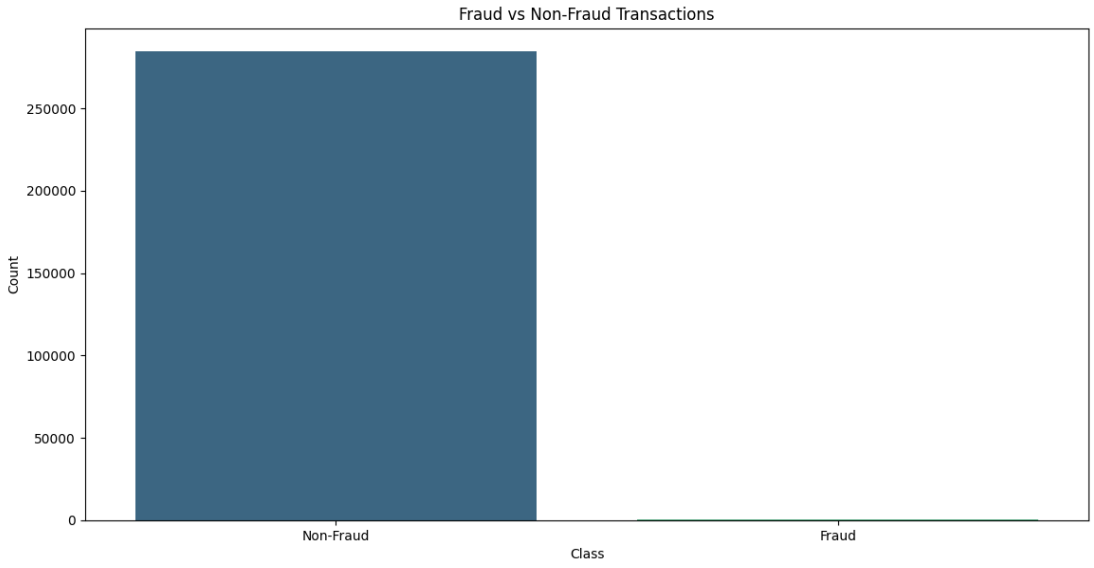
```
# Standardize Features
# Standardize 'Time' and 'Amount'
scaler = StandardScaler()
df["scaled_time"] = scaler.fit_transform(df[["Time"]])
df["scaled_amount"] = scaler.fit_transform(df[["Amount"]])

# Drop original 'Time' and 'Amount' columns
df = df.drop(["Time", "Amount"], axis=1)


# EDA
# fraud vs non-fraud transaction
print("fraud vs non-fraud transaction")
```

```python
# Check distribution of target variable
fraud_counts = df["Class"].value_counts()
print(fraud_counts)
```

```python
# Plot fraud vs. non-fraud distribution
plt.figure(figsize=(5, 5))
sns.barplot(x=fraud_counts.index, y=fraud_counts.values, palette="viridis")
plt.title("Fraud vs Non-Fraud Transactions")
plt.xticks([0, 1], ["Non-Fraud", "Fraud"])
plt.ylabel("Count")
plt.show()
```

Output: -

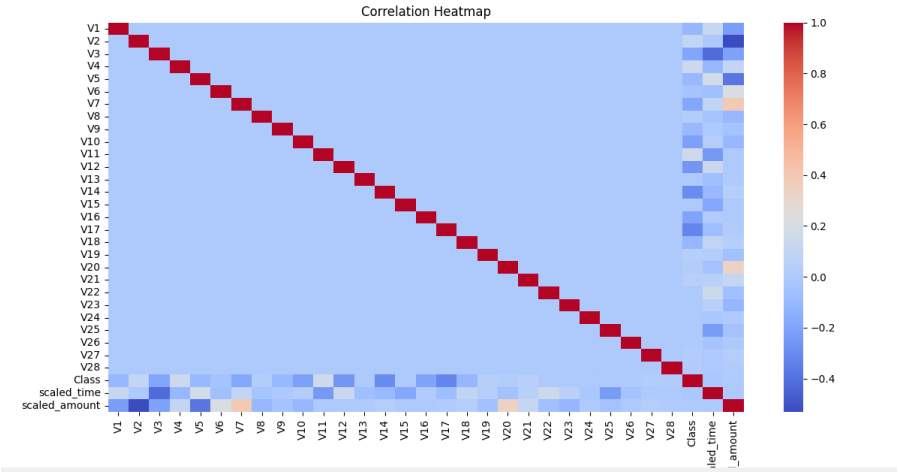
```
fraud vs non-fraud transaction
Class
0    284315
1       492
Name: count, dtype: int64
d:\uet\phd\sem1\ads\assignments\project\code\credit_card_fraud_detection.py:57: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `
legend=False` for the same effect.

  sns.barplot(x=fraud_counts.index, y=fraud_counts.values, palette="viridis")
```




```python
# Feature-Target Relationship
print ("Feature-Target Relationship")
# Plot correlation heatmap
```

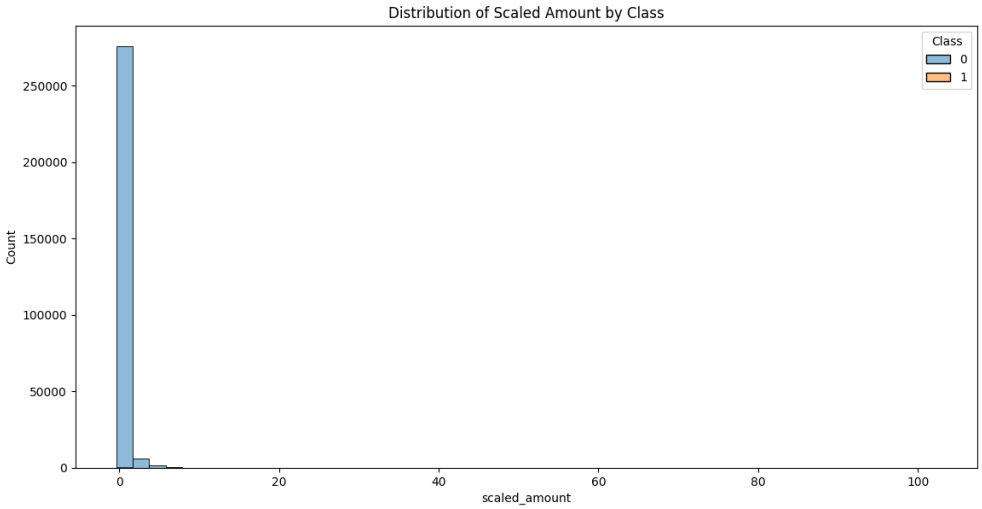
```
plt.figure(figsize=(15, 10))
correlation = df.corr()
sns.heatmap(correlation, cmap="coolwarm", annot=False)
plt.title("Correlation Heatmap")
plt.show()
```

Output: -



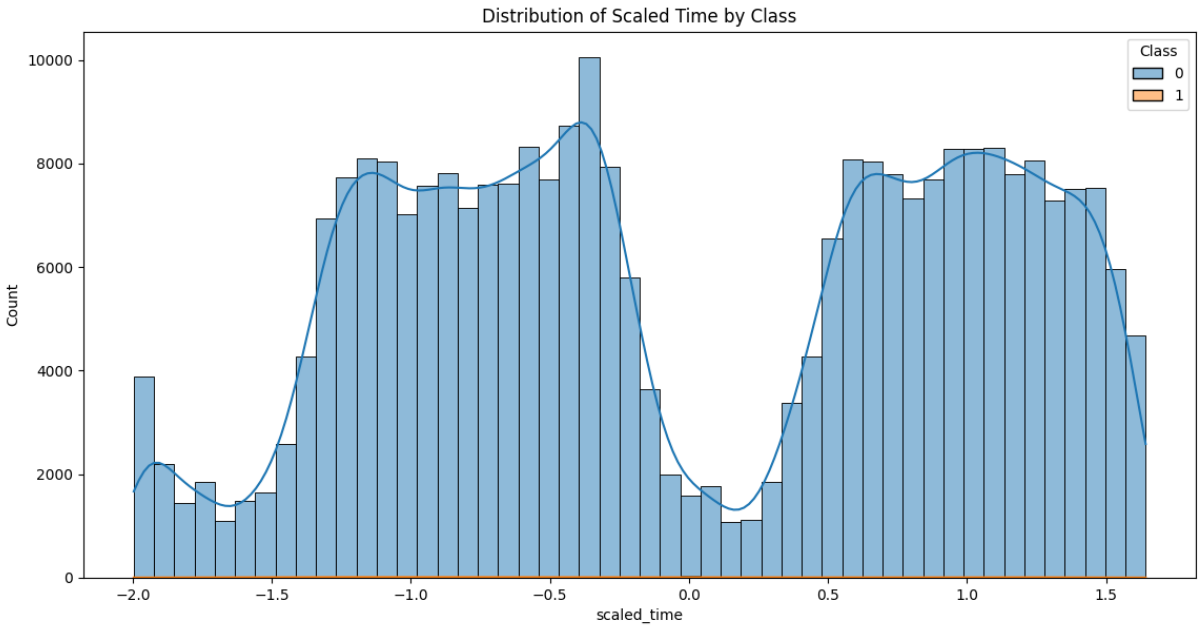

```
# Feature Analysis
print ("Feature Analysis")
# Analyze the relationship between scaled features and target
sns.histplot(data=df, x="scaled_amount", hue="Class", bins=50)
plt.title("Distribution of Scaled Amount by Class")
plt.show()
```

Output: -



```
sns.histplot(data=df, x="scaled_time", hue="Class", kde=True, bins=50)
plt.title("Distribution of Scaled Time by Class")
plt.show()
```

Output: -



7. Predictive Analysis: -

SMOTE: -

The code using **SMOTE (Synthetic Minority Oversampling Technique)** was applied to handle the **class imbalance** in the dataset.

Understanding Class Imbalance

- In the credit card fraud detection dataset, the target variable (Class) is highly imbalanced:
  - Non-fraudulent transactions (Class = 0) account for over 99.8%.
  - Fraudulent transactions (Class = 1) make up only 0.172%.
- This imbalance poses a challenge for machine learning models, which tend to be biased toward the majority class and perform poorly on the minority class (fraud).

Why Handle Class Imbalance?

Without addressing the class imbalance:

- **Models tend to prioritize accuracy**: A model could achieve over 99% accuracy by predicting only the majority class (non-fraudulent transactions) but fail to identify frauds.
- **Poor recall for the minority class**: The model would miss most fraudulent transactions, leading to high **false negatives**, which are costly in fraud detection.

Why SMOTE?

- **SMOTE (Synthetic Minority Oversampling Technique)** generates synthetic samples for the minority class (fraudulent transactions) by interpolating between existing samples.
- Unlike simple oversampling, which duplicates minority class samples, SMOTE creates synthetic data points, reducing the risk of overfitting.

4. How SMOTE Helps

- After applying SMOTE, the training set has a balanced number of samples for both classes, allowing the model to:
    - Learn patterns associated with fraud more effectively.
    - Improve recall and precision for the minority class.

When to Use SMOTE

- SMOTE is particularly useful when:
    - The dataset is heavily imbalanced.
    - The minority class is important, such as in fraud detection or medical diagnoses.
- SMOTE should be applied **only to the training set** to prevent data leakage into the test set.

Model: Random Forest Classifier: -

The Random Forest algorithm is an ensemble learning method primarily used for classification and regression tasks. It combines multiple decision trees (weak learners) to create a stronger predictive model.

- **Decision Tree Building**: Each subset is used to train a separate decision tree.
- **Feature Randomization**: At each split, only a random subset of features is considered to reduce correlation between trees.
- **Random Forest** is chosen here because it can handle imbalanced datasets with class_weight.

```
print ("predictive analysis")
# Separate features and target
X = df.drop("Class", axis=1)
y = df["Class"]

# Split into training and test sets
# Ensures that the training and test sets have the same distribution of
fraudulent and non-fraudulent transactions (stratify=y).
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 handle class imbalance
smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)
# Train a Random Forest model
rf_model = RandomForestClassifier(
    random_state=42, n_estimators=100, class_weight="balanced"
)
rf_model.fit(X_train_resampled, y_train_resampled)

# Predict on the test set
# This method applies the trained Random Forest model to the input data (X_test)
and predicts the output labels (y_pred).
y_pred = rf_model.predict(X_test)

# Evaluate model performance
print(classification_report(y_test, y_pred))

# Precision-Recall Curve
# Compute precision-recall values
y_pred_proba = rf_model.predict_proba(X_test)[:, 1]
precision, recall, thresholds = precision_recall_curve(y_test, y_pred_proba)
```

Output: -

```
predictive analysis
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     85295
           1       0.86      0.76      0.81       148

    accuracy                           1.00     85443
   macro avg       0.93      0.88      0.90     85443
weighted avg       1.00      1.00      1.00     85443

AUC-PR: 0.9618226805855037
```

Precision-Recall Curve: -

```
# Precision-Recall Curve
# Compute precision-recall values
y_pred_proba = rf_model.predict_proba(X_test)[:, 1]
precision, recall, thresholds = precision_recall_curve(y_test, y_pred_proba)

# Plot Precision-Recall Curve
plt.figure(figsize=(10, 5))
plt.plot(recall, precision, marker=".", label="Random Forest")
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.title("Precision-Recall Curve")
plt.legend()
plt.show()

# Compute AUC-PR
auc_pr = roc_auc_score(y_test, y_pred_proba)
print(f"AUC-PR: {auc_pr}")
```

Output: -

```
AUC-PR: 0.9618226805855037
```