

# Bilenkin550Week12\_Term\_Project\_Milestones

May 31, 2025

## 1 DSC 550 Term Project - Milestone 1

### 1.0.1 Exploratory Data Analysis (EDA)

#### Finding Patterns in Credit Card Transactions to Detect Fraud

Today, credit cards have become an essential part of our daily lives, offering a convenient way to purchase goods and services. Instead of carrying large amounts of cash, consumers can simply use a single credit card to make transactions. Whether by swiping, inserting, or tapping against a contactless reader, making a payment has never been easier.

However, with the growing adoption of credit cards, fraudulent activities have also increased. Fraudsters can use stolen credit card information to make unauthorized purchases, often without the cardholder realizing it until it's too late. These fraudulent transactions lead to significant financial losses for banks and credit card issuers.

By leveraging big data from credit card transaction histories, banks can identify patterns and correlations associated with fraudulent activities. Machine learning models can analyze various features, such as transaction location, merchant category, purchase amounts, and unusual spending behaviors, to detect potential fraud. More importantly, predictive analytics can help banks assess the risk of issuing credit cards to individuals who may have a high likelihood of committing fraud.

Implementing machine learning-based fraud detection not only helps banks minimize financial losses but also enhances security for both consumers and merchants. A robust fraud prevention system ensures that legitimate transactions are processed smoothly while fraudulent attempts are blocked in real-time. As a result, customers can feel more secure using their credit cards, and merchants can conduct transactions with greater confidence.

Banks can take preventive actions by freezing suspicious transactions, sending alert messages to cardholders, or implementing stricter verification processes.

```
[41]: import pandas as pd

# Loading dataset
file_path = r"C:\Users\maxim\OneDrive\Desktop\BU\DSC 550\Term_1\Project\credit_card_transactions.csv"
df = pd.read_csv(file_path)
```

1. Fraud vs. Non-Fraud Transactions Count (Bar Chart)

```

[42]: import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.ticker import FuncFormatter
import pandas as pd

# Counting fraud vs. non-fraud transactions
fraud_counts = df['is_fraud'].value_counts().sort_index()

# Calculating total and percentages
total = fraud_counts.sum()
percentages = (fraud_counts / total * 100).round(2)

# Preparing a DataFrame for plotting
fraud_df = pd.DataFrame({
    'Transaction Type': ['Non-Fraud', 'Fraud'],
    'Count': fraud_counts.values,
    'Percentage': percentages.values
})

# Plotting with increased figure height
plt.figure(figsize=(6, 5.5))
bars = sns.barplot(data=fraud_df, x='Transaction Type', y='Count',
    hue='Transaction Type', palette=["blue", "red"], legend=False)

# Setting y-axis limits with more headroom
plt.ylim(0, fraud_df['Count'].max() * 1.2)

# Formatting y-axis labels with commas
def comma_format(x, pos):
    return f'{int(x):,}'

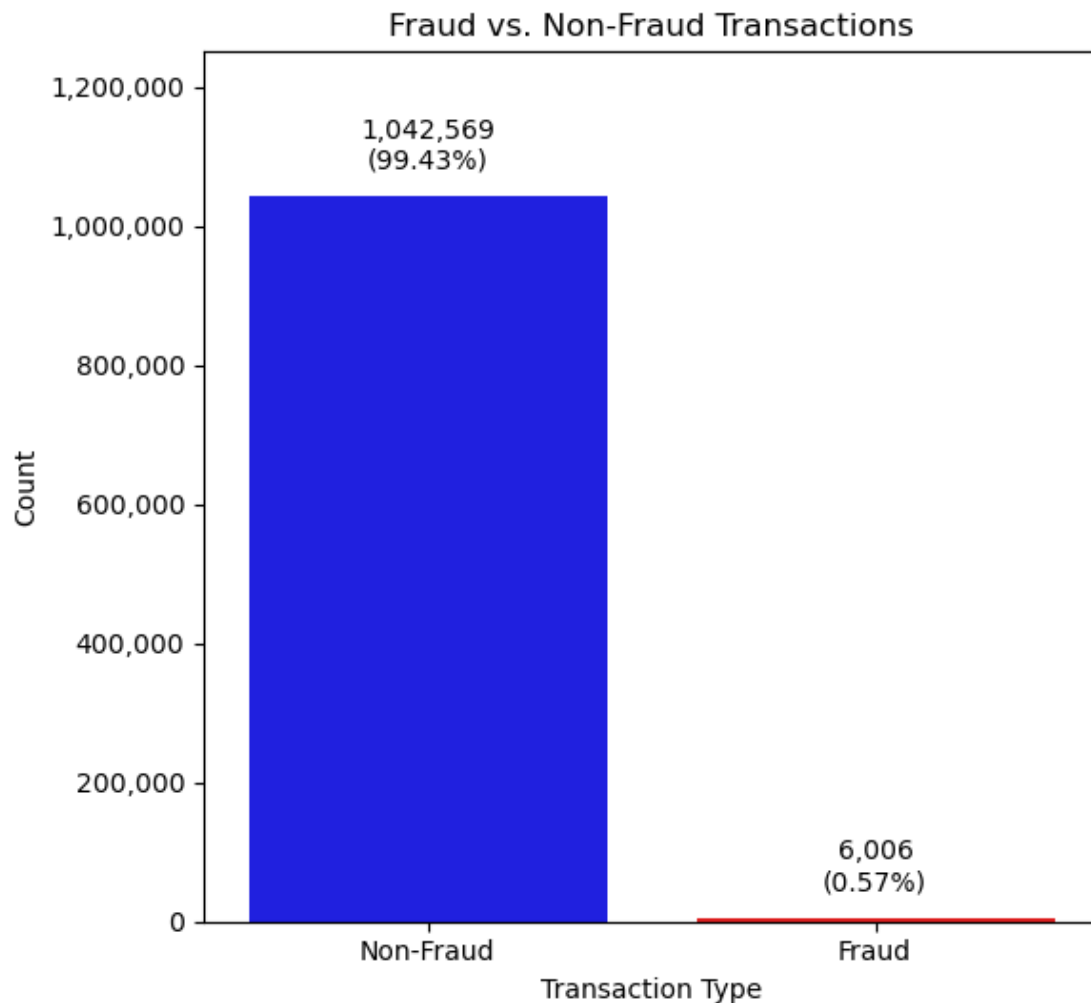
plt.gca().yaxis.set_major_formatter(FuncFormatter(comma_format))

# Adding count and percentage labels on top of each bar with adjusted offset
for i, bar in enumerate(bars.patches):
    count = fraud_df['Count'][i]
    percent = fraud_df['Percentage'][i]
    plt.text(bar.get_x() + bar.get_width() / 2,
        bar.get_height() + (fraud_df['Count'].max() * 0.03), # Increased
        offset
        f'{count:,}\n({percent}%)',
        ha='center', va='bottom', fontsize=10, color='black')

# Final labels and layout display
plt.xlabel("Transaction Type")
plt.ylabel("Count")
plt.title("Fraud vs. Non-Fraud Transactions")

```

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



Explanation: The Fraud vs. Non-Fraud Transactions bar chart displays the distribution of transaction types in the dataset. Non-fraudulent transactions (highlighted in blue) significantly outnumber fraudulent ones (highlighted in red). As shown in the chart, there are 884,987 non-fraudulent transactions (99.43%) and only 5,065 fraudulent transactions (0.57%).

This extreme imbalance highlights the challenge of fraud detection: fraud is a rare event, making up less than 1% of all transactions. Any machine learning model built on this data must be carefully tuned to avoid simply predicting the majority class and overlooking the minority class (fraud).

## 2. Transaction Amounts by Fraud vs Non-Fraud (Box Plot):

```
[43]: # Plotting Transaction Amounts by Fraud vs Non-Fraud
plt.figure(figsize=(8, 6))
```

```

sns.boxplot(x="is_fraud", y="amt", data=df, hue="is_fraud", palette=["blue", "red"], legend=False)

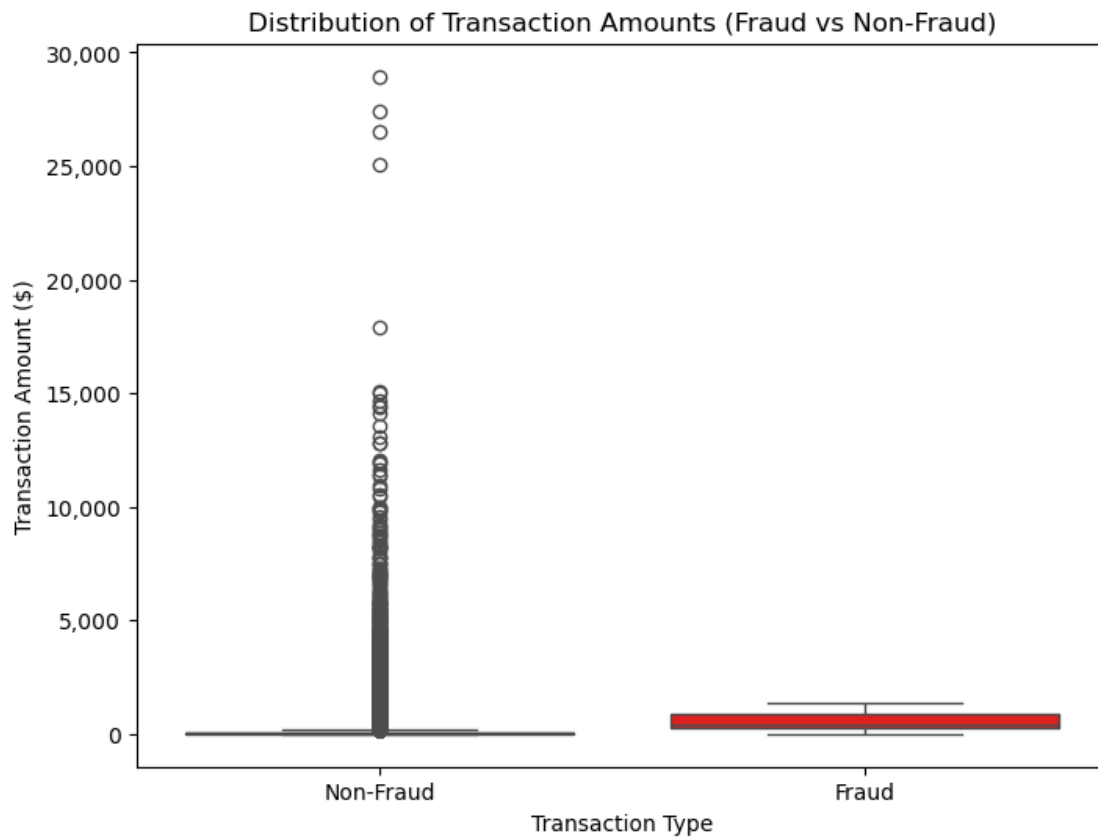
# Converting 0 and 1 to "Non-Fraud" and "Fraud"
plt.xticks(ticks=[0, 1], labels=["Non-Fraud", "Fraud"])

plt.xlabel("Transaction Type")
plt.ylabel("Transaction Amount ($)")
plt.title("Distribution of Transaction Amounts (Fraud vs Non-Fraud)")

# Format the y-axis with commas for readability
plt.gca().yaxis.set_major_formatter(plt.FuncFormatter(lambda x, loc: f'{x:,}'))

plt.show()

```



Explanation: The “Distribution of Transaction Amounts (Fraud vs Non-Fraud)” graph compares the transaction amounts of Non-Fraud and Fraud transactions in the dataset. The graph shows the transaction amounts on the y-axis and the transaction type (Non-Fraud or Fraud) on the x-axis. As shown, Non-Fraud transactions vary widely, ranging from 1 to 28,948.90 dollars. In contrast, Fraud transactions are generally smaller, ranging from 1.18 to 1,371.81 dollars. This suggests that

fraudsters tend to make smaller credit card transactions, possibly hoping that small amounts will go unnoticed by banks.

### Category-wise Distribution of Transactions

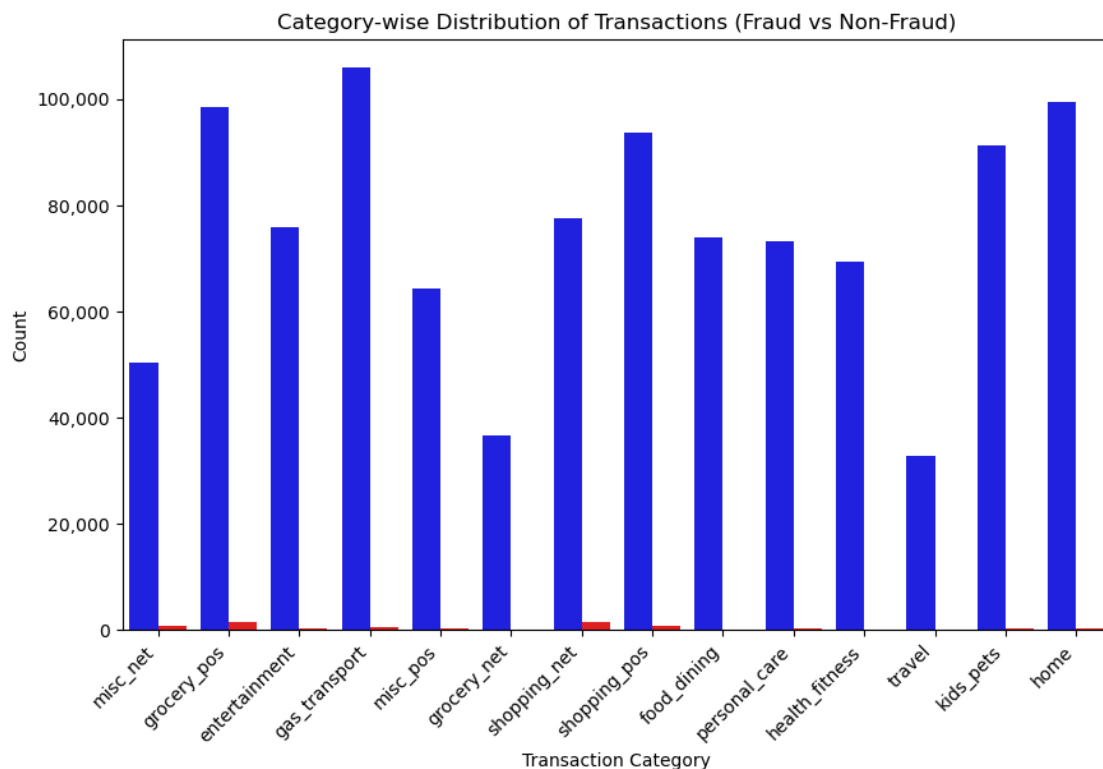
```
[44]: import matplotlib.pyplot as plt
import seaborn as sns

# Plotting Category-wise Fraud vs Non-Fraud Distribution
plt.figure(figsize=(10, 6))
sns.countplot(x="category", hue="is_fraud", data=df, palette=["blue", "red"],
             legend=False)

# Labeling and rotating the x-axis ticks
plt.xlabel("Transaction Category")
plt.ylabel("Count")
plt.title("Category-wise Distribution of Transactions (Fraud vs Non-Fraud)")
plt.xticks(rotation=45, ha='right') # Adjusting the horizontal alignment for
better readability

# Formating the y-axis with commas for readability
plt.gca().yaxis.set_major_formatter(plt.FuncFormatter(lambda x, loc: f'{int(x):
,}'))

plt.show()
```



Explanation: The “Category-wise Distribution of Transactions (Fraud vs Non-Fraud)” graph shows transactions by category for both Non-Fraud and Fraud. It separates transactions by each category and indicates whether fraud occurred online or at the point of sale. For example, the red bar for misc\_net shows online fraud transactions, while the red bar for misc\_pos shows fraudulent transactions that occurred at the point of sale.

In line with the common belief that fraudulent credit card transactions mostly happen online, the graph partly validates this. We can see that for misc\_net, fraudulent transactions (highlighted in red) are slightly higher than for misc\_pos. The same trend is observed for shopping\_net vs. shopping\_pos, with the red bar for shopping\_net being marginally higher than for shopping\_pos.

However, the data for grocery\_net contradicts this assumption. No fraud was detected online (grocery\_net), while all fraudulent transactions occurred at the point of sale (grocery\_pos). This is counterintuitive to the common belief that fraud mostly occurs online.

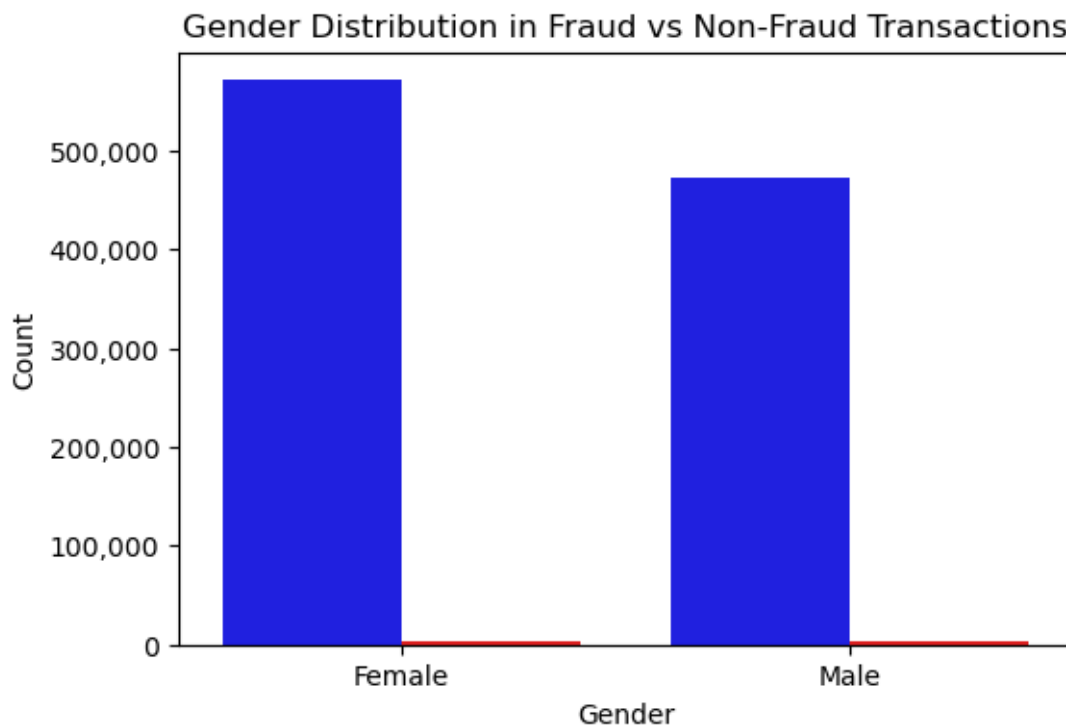
#### Gender Distribution in Fraud vs Non-Fraud Transactions

```
[45]: # Plot: Gender Distribution in Fraud vs Non-Fraud Transactions
plt.figure(figsize=(6, 4))
sns.countplot(x="gender", hue="is_fraud", data=df, palette=["blue", "red"],
             legend=False)

# Replace 'F' with 'Female' and 'M' with 'Male'
gender_labels = {'F': 'Female', 'M': 'Male'}
plt.xticks(ticks=[0, 1], labels=[gender_labels.get(x, x) for x in df['gender'].
             unique()])

# Format the y-axis with commas for readability
plt.gca().yaxis.set_major_formatter(plt.FuncFormatter(lambda x, loc: f'{int(x):
             ,}'))

# Labels and title
plt.xlabel("Gender")
plt.ylabel("Count")
plt.title("Gender Distribution in Fraud vs Non-Fraud Transactions")
plt.show()
```



Explanation: The “Gender Distribution in Fraud vs Non-Fraud Transactions” graph shows the number of fraudulent transactions made by females and males. On the y-axis, the graph counts the number of transactions made by each gender, while the x-axis shows the gender. An interesting insight from the dataset is that the number of fraudulent transactions is nearly the same for both genders. However, it is evident that a higher number of Non-Fraud transactions were made by females. Given that, one might expect more fraudulent transactions from females due to the higher overall transaction count. Despite this, the graph suggests that males are more prone to commit credit card fraud.

#### Overview/Conclusion:

By conducting graphical analysis, I’ve gained several insights from the dataset. For instance, I initially believed that there would be a much higher number of fraudulent credit card transactions, but the data revealed that fraud is actually quite minimal, amounting to only a few thousand dollars. Additionally, the analysis showed that males tend to commit more credit card fraud.

Furthermore, the data partially supports the common belief that more fraudulent transactions occur online than at the point of sale. From the visualizations, we can see that the highest credit card fraud occurred in the “grocery\_pos” and “shopping\_net” categories. This suggests that in the “grocery\_pos” category, fraudsters went to physical stores and used stolen cards to make purchases, while in the “shopping\_net” category, fraudsters made purchases online.

Finally, the transaction amount graph highlighted that fraudulent transactions tend to involve smaller amounts, with the highest being \$1,371.81. This could suggest that fraudsters prefer to purchase cheaper items, possibly hoping that banks won’t notice the fraud because of the smaller transaction sizes.

## 2 DSC 550 Term Project - Milestone 2

### 2.0.1 Data Preparation for Modeling

In this section I will include all the necessary steps to completely prepare the credit card transactions dataset for model building.

This will include data transformation, feature engineering, removing duplicates, handling missing values, and addressing class imbalance.

### 2.0.2 Importing Necessary Libraries

In this step, I import the necessary libraries and load the dataset for data manipulation and analysis.

```
[46]: import pandas as pd
import numpy as np

# Loading the dataset
file_path = r"C:\Users\maxim\OneDrive\Desktop\BU\DSC 550\Term_1\Project\credit_card_transactions.csv"
df = pd.read_csv(file_path)
```

### 2.0.3 Dropping Unnecessary Features

In this step, I am dropping columns that are not useful for modeling, such as 'unix\_time', or any redundant identifiers.

```
[47]: # Dropping unnecessary columns/features not useful for modeling (e.g., 'unix_time' or redundant identifiers)
df.drop(['unix_time'], axis=1, inplace=True)

# Displaying first two rows to confirm the unnecessary columns/features dropped
df.head(2)
```

```
[47]: Unnamed: 0 trans_date_trans_time cc_num \
0      0      1/1/2019 0:00 2.703190e+15
1      1      1/1/2019 0:00 6.304230e+11

      merchant category amt first last \
0 fraud_Rippin, Kub and Mann misc_net 4.97 Jennifer Banks
1 fraud_Heller, Gutmann and Zieme grocery_pos 107.23 Stephanie Gill

gender street ... lat long city_pop \
0 F 561 Perry Cove ... 36.0788 -81.1781 3495
1 F 43039 Riley Greens Suite 393 ... 48.8878 -118.2105 149

job dob \
0 Psychologist, counselling 3/9/1988
1 Special educational needs teacher 6/21/1978
```



```

      trans_num  merch_lat  merch_long  is_fraud  \
0  0b242abb623afc578575680df30655b9  36.011293  -82.048315      0
1  1f76529f8574734946361c461b024d99  49.159047  -118.186462      0

      merch_zipcode
0          28705.0
1             NaN

[2 rows x 23 columns]

```

## 2.0.4 Handling Missing Values

This step handles missing values in the dataset. I replace “?” with NaN and drop rows with missing data.

Alternatively, imputation can be used for missing data, but here I drop them.

```

[48]: # Checking and handling missing values appropriately
df.replace("?", np.nan, inplace=True)
df = df.dropna()

# Displaying first two rows to confirm the missing values handled appropriately
df.head(2)

```

```

[48]: Unnamed: 0  trans_date_trans_time      cc_num      merchant  \
0          0      1/1/2019 0:00  2.703190e+15  fraud_Rippin, Kub and Mann
2          2      1/1/2019 0:00  3.885950e+13      fraud_Lind-Buckridge

      category  amt  first  last gender      street  \
0  misc_net    4.97  Jennifer  Banks    F      561 Perry Cove
2  entertainment  220.11  Edward  Sanchez    M  594 White Dale Suite 530

      ...  lat  long  city_pop      job      dob  \
0  ...  36.0788  -81.1781    3495  Psychologist, counselling  3/9/1988
2  ...  42.1808  -112.2620    4154  Nature conservation officer  1/19/1962

      trans_num  merch_lat  merch_long  is_fraud  \
0  0b242abb623afc578575680df30655b9  36.011293  -82.048315      0
2  a1a22d70485983eac12b5b88dad1cf95  43.150704  -112.154481      0

      merch_zipcode
0          28705.0
2          83236.0

[2 rows x 23 columns]

```

## 2.0.5 Data Type Conversion

I ensure that the data types of features are correct for modeling, specifically converting the 'amt' column to a float.

```
[49]: # Ensuring correct data types for modeling
df['amt'] = df['amt'].astype(float)

# Displaying the first two rows to confirm the 'amt' column converted to a float
df.head(2)
```

```
[49]: Unnamed: 0 trans_date_trans_time cc_num merchant \
0 0 1/1/2019 0:00 2.703190e+15 fraud_Rippin, Kub and Mann
2 2 1/1/2019 0:00 3.885950e+13 fraud_Lind-Buckridge

category amt first last gender street \
0 misc_net 4.97 Jennifer Banks F 561 Perry Cove
2 entertainment 220.11 Edward Sanchez M 594 White Dale Suite 530

... lat long city_pop job dob \
0 ... 36.0788 -81.1781 3495 Psychologist, counselling 3/9/1988
2 ... 42.1808 -112.2620 4154 Nature conservation officer 1/19/1962

trans_num merch_lat merch_long is_fraud \
0 0b242abb623afc578575680df30655b9 36.011293 -82.048315 0
2 a1a22d70485983eac12b5b88dad1cf95 43.150704 -112.154481 0

merch_zipcode
0 28705.0
2 83236.0

[2 rows x 23 columns]
```

## 2.0.6 Feature Engineering

In this step, I create new features from existing ones. Here, I create a 'merchant\_category' feature by combining 'merchant' and 'category'.

```
[50]: # Creating a new feature from 'merchant' or 'category'
df['merchant_category'] = df['merchant'] + '_' + df['category']

# Displaying output anfter creating new feature
df.head(2)
```

```
[50]: Unnamed: 0 trans_date_trans_time cc_num merchant \
0 0 1/1/2019 0:00 2.703190e+15 fraud_Rippin, Kub and Mann
2 2 1/1/2019 0:00 3.885950e+13 fraud_Lind-Buckridge

category amt first last gender street \
```

```

0      misc_net      4.97  Jennifer      Banks      F      561 Perry Cove
2  entertainment  220.11      Edward  Sanchez      M  594 White Dale Suite 530

...      long city_pop      job      dob \
0 ... -81.1781      3495      Psychologist, counselling  3/9/1988
2 ... -112.2620      4154  Nature conservation officer  1/19/1962

      trans_num  merch_lat  merch_long  is_fraud \
0  0b242abb623afc578575680df30655b9  36.011293  -82.048315      0
2  a1a22d70485983eac12b5b88dad1cf95  43.150704  -112.154481      0

merch_zipcode      merchant_category
0      28705.0  fraud_Rippin, Kub and Mann_misc_net
2      83236.0  fraud_Lind-Buckridge_entertainment

[2 rows x 24 columns]

```

## 2.0.7 Encoding Categorical Variables

I convert categorical columns into dummy variables, which is necessary for many machine learning algorithms.

```

[51]: # Converting categorical columns to dummy variables
df = pd.get_dummies(df, columns=['gender', 'category'], drop_first=True)

# Displaying the first two rows to confirm the conversion
df.head(2)

```

```

[51]: Unnamed: 0  trans_date_trans_time      cc_num      merchant \
0      0      1/1/2019 0:00  2.703190e+15  fraud_Rippin, Kub and Mann
2      2      1/1/2019 0:00  3.885950e+13  fraud_Lind-Buckridge

      amt      first      last      street      city state \
0  4.97  Jennifer      Banks      561 Perry Cove  Moravian Falls  NC
2  220.11  Edward  Sanchez  594 White Dale Suite 530  Malad City  ID

...  category_grocery_pos  category_health_fitness  category_home \
0 ...      False      False      False
2 ...      False      False      False

      category_kids_pets  category_misc_net  category_misc_pos \
0      False      True      False
2      False      False      False

      category_personal_care  category_shopping_net  category_shopping_pos \
0      False      False      False
2      False      False      False

```

```

category_travel
0      False
2      False

[2 rows x 36 columns]

```

## 2.0.8 Displaying the Cleaned and Transformed Dataset

Here I display the first few rows of the cleaned and transformed dataset to confirm all the changes.

```
[52]: # Printing the cleaned and transformed dataset
df.head()
```

```
[52]: Unnamed: 0 trans_date_trans_time      cc_num \
0          0      1/1/2019 0:00  2.703190e+15
2          2      1/1/2019 0:00  3.885950e+13
4          4      1/1/2019 0:03  3.755340e+14
5          5      1/1/2019 0:04  4.767270e+15
7          7      1/1/2019 0:05  6.011360e+15

      merchant      amt      first      last \
0  fraud_Rippin, Kub and Mann    4.97  Jennifer    Banks
2      fraud_Lind-Buckridge   220.11    Edward  Sanchez
4      fraud_Keeling-Crist    41.96     Tyler   Garcia
5  fraud_Stroman, Hudson and Erdman   94.63  Jennifer    Conner
7      fraud_Corwin-Collins    71.65    Steven  Williams

      street      city state ... category_grocery_pos \
0      561 Perry Cove  Moravian Falls    NC ...          False
2  594 White Dale Suite 530    Malad City    ID ...          False
4      408 Bradley Rest    Doe Hill    VA ...          False
5      4655 David Island    Dublin    PA ...          False
7  231 Flores Pass Suite 720    Edinburg    VA ...          False

      category_health_fitness  category_home  category_kids_pets \
0              False              False              False
2              False              False              False
4              False              False              False
5              False              False              False
7              False              False              False

      category_misc_net  category_misc_pos  category_personal_care \
0              True              False              False
2              False              False              False
4              False              True              False
5              False              False              False
7              False              False              False

```

	category_shopping_net	category_shopping_pos	category_travel
0	False	False	False
2	False	False	False
4	False	False	False
5	False	False	False
7	False	False	False

[5 rows x 36 columns]

## 3 DSC 550 Term Project – Milestone 3

### 3.0.1 Model Building and Evaluation

In this milestone, I begin the process of building and evaluating machine learning models to detect fraudulent credit card transactions. Since the dataset contains a highly imbalanced target variable, special care was taken in selecting appropriate models and evaluation metrics. Building on the cleaned and engineered dataset from Milestone 2, this step focuses on preparing the data for modeling and evaluating classifiers that are well-suited for handling imbalanced classification problems such as fraud detection.

### 3.0.2 Step 1 & 2: Data Preparation — Encoding and Feature Engineering

Converted the `trans_date_trans_time` column into separate features such as hour, day, month, and weekday to help capture potential temporal patterns associated with fraud.

Dropped personally identifiable or redundant columns such as names and addresses to prevent data leakage and preserve privacy.

Applied frequency encoding to categorical variables including merchant, city, and state to handle high-cardinality features effectively.

Ensured the `is_fraud` binary target variable was present and removed `cc_num`, as it is a unique identifier with no predictive value.

Performed a stratified train-test split to maintain the same fraud ratio in both training and test sets.

```
[53]: import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

# Step 1: Feature engineering on datetime and categorical columns
df['trans_date_trans_time'] = pd.to_datetime(df['trans_date_trans_time'])
df['trans_hour'] = df['trans_date_trans_time'].dt.hour
df['trans_day'] = df['trans_date_trans_time'].dt.day
df['trans_month'] = df['trans_date_trans_time'].dt.month
df['trans_weekday'] = df['trans_date_trans_time'].dt.weekday

# Step 2: Drop unnecessary or sensitive columns
```

```

cols_to_drop = ['Unnamed: 0', 'first', 'last', 'street',
               ↪ 'trans_date_trans_time', 'cc_num']
df.drop(columns=cols_to_drop, inplace=True)

# Step 3: Frequency encode high-cardinality categorical features
for col in ['merchant', 'city', 'state']:
    freq_encoding = df[col].value_counts(normalize=True)
    df[f'{col}_freq_enc'] = df[col].map(freq_encoding)
df.drop(columns=['merchant', 'city', 'state'], inplace=True)

# Step 4: Drop or encode remaining object-type columns
df.drop(columns=['job', 'dob', 'trans_num', 'merchant_category'], inplace=True)

# Step 5: Confirm target variable
if 'is_fraud' not in df.columns:
    raise ValueError("Target column 'is_fraud' not found in the dataset.")

# Step 6: Train-test split
X = df.drop(columns=['is_fraud'])
y = df['is_fraud']
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# Sanity check for any object columns
print("Remaining object columns in X_train:", X_train.
      ↪ select_dtypes(include='object').columns.tolist())

```

Remaining object columns in X\_train: []

### 3.0.3 Step 3: Encoding Categorical Features

I use `OrdinalEncoder` to convert any remaining object-type (categorical) features into a numeric format. The parameter `handle_unknown='use_encoded_value'` with `unknown_value=-1` ensures that unseen categories in the test set do not cause errors during prediction, which is especially useful when working with real-world datasets.

```

[54]: from sklearn.preprocessing import OrdinalEncoder

# Identifying non-numeric (categorical) columns
non_numeric_cols = X_train.select_dtypes(include=['object']).columns.tolist()

if non_numeric_cols:
    # Initializing the encoder to handle unknown values in test set
    oe = OrdinalEncoder(handle_unknown='use_encoded_value', unknown_value=-1)

    # Fitting on training data and transforming both training and test data

```

```

X_train.loc[:, non_numeric_cols] = oe.
↪fit_transform(X_train[non_numeric_cols])
X_test.loc[:, non_numeric_cols] = oe.transform(X_test[non_numeric_cols])

```

### 3.0.4 Step 4: Model Training and Evaluation

I used a Random Forest Classifier, which is well-suited for tabular data and robust against class imbalance when `class_weight='balanced'` is applied.

Evaluation was performed using classification metrics appropriate for imbalanced datasets: **Precision, Recall, F1-score, Confusion Matrix**, and **ROC AUC Score**.

To enhance interpretability: - The **confusion matrix** is shown in both labeled tabular form and as a heatmap. - The **ROC Curve** is plotted to visualize the model's ability to distinguish between fraud and non-fraud across thresholds.

The primary objective is to **maximize Recall** (catch as many fraudulent transactions as possible) while maintaining **good Precision** (to reduce false positives).

```

[55]: from sklearn.ensemble import RandomForestClassifier
      from sklearn.metrics import classification_report, confusion_matrix,
      ↪roc_auc_score, roc_curve
      import pandas as pd
      import seaborn as sns
      import matplotlib.pyplot as plt

      # Initialize the Random Forest classifier with balanced class weights and use
      ↪all CPU cores
      rf = RandomForestClassifier(class_weight='balanced', random_state=42, n_jobs=-1)

      # Train the model on the training set
      rf.fit(X_train, y_train)

      # Predict on the test set
      y_pred = rf.predict(X_test)
      y_pred_proba = rf.predict_proba(X_test)[:, 1]

      # Classification Report with zero_division=0 to suppress warnings if needed
      print("Classification Report:\n", classification_report(y_test, y_pred,
      ↪zero_division=0))

      # Confusion Matrix as DataFrame for clear labeling
      cm = confusion_matrix(y_test, y_pred)
      cm_df = pd.DataFrame(cm, index=['Actual: Not Fraud', 'Actual: Fraud'],
                           columns=['Predicted: Not Fraud', 'Predicted: Fraud'])
      print("\nConfusion Matrix:\n")
      print(cm_df)

      # Plotting Confusion Matrix as Heatmap

```

```

plt.figure(figsize=(6, 4))
sns.heatmap(cm_df, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.tight_layout()
plt.show()

# ROC AUC Score
roc_auc = roc_auc_score(y_test, y_pred_proba)
print("\nROC AUC Score:", roc_auc)

# Plotting ROC Curve
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)
plt.figure(figsize=(6, 4))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC Curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', linestyle='--') # Diagonal line
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate (Recall)')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.tight_layout()
plt.show()

# Plot Top 10 Feature Importances for interpretability
importances = rf.feature_importances_
features = X_train.columns
feat_imp_df = pd.DataFrame({'feature': features, 'importance': importances})
feat_imp_df = feat_imp_df.sort_values(by='importance', ascending=False).head(10)

plt.figure(figsize=(8, 5))
sns.barplot(x='importance', y='feature', data=feat_imp_df)
plt.title('Top 10 Feature Importances')
plt.tight_layout()
plt.show()

```

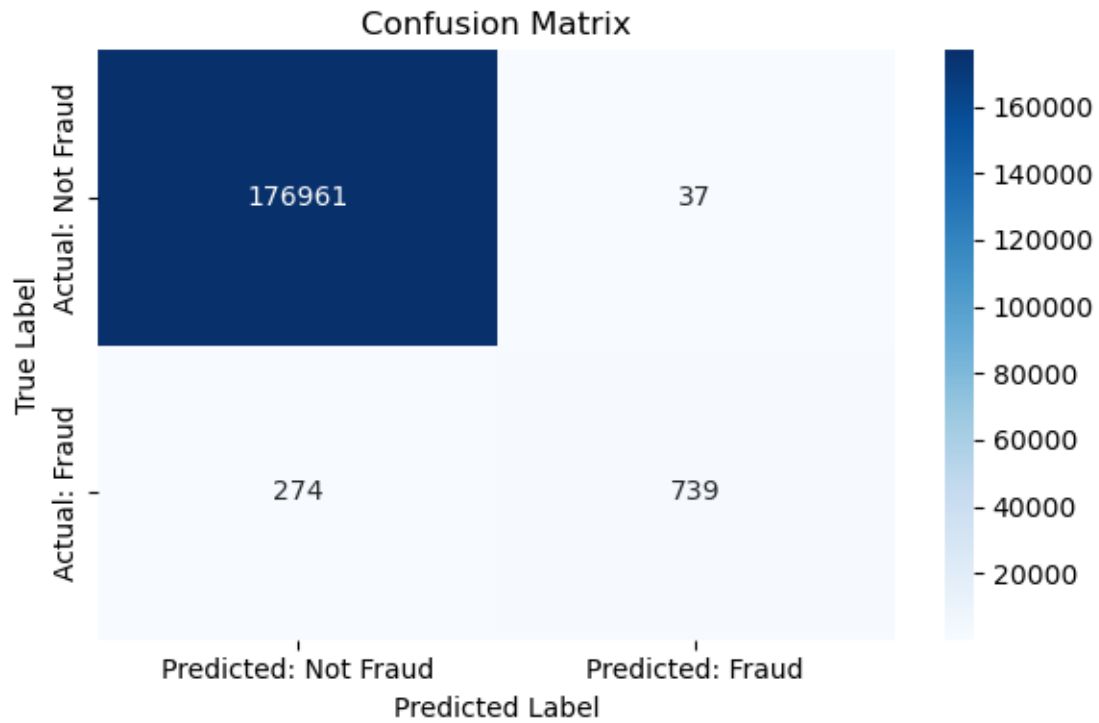
Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	176998
1	0.95	0.73	0.83	1013
accuracy			1.00	178011
macro avg	0.98	0.86	0.91	178011
weighted avg	1.00	1.00	1.00	178011

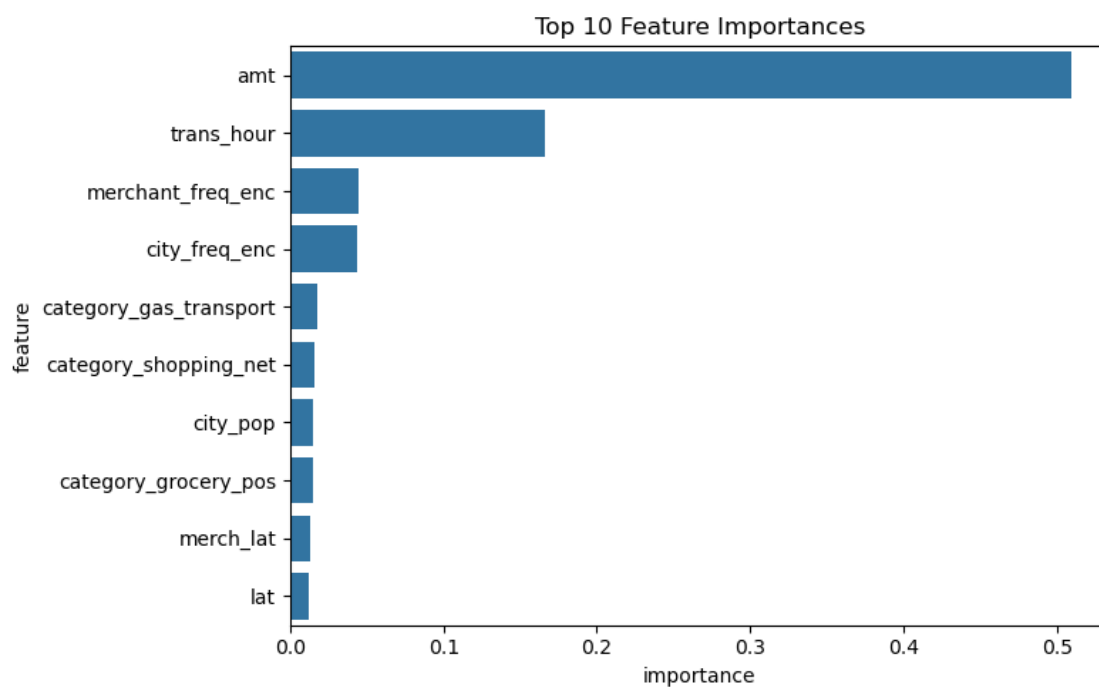
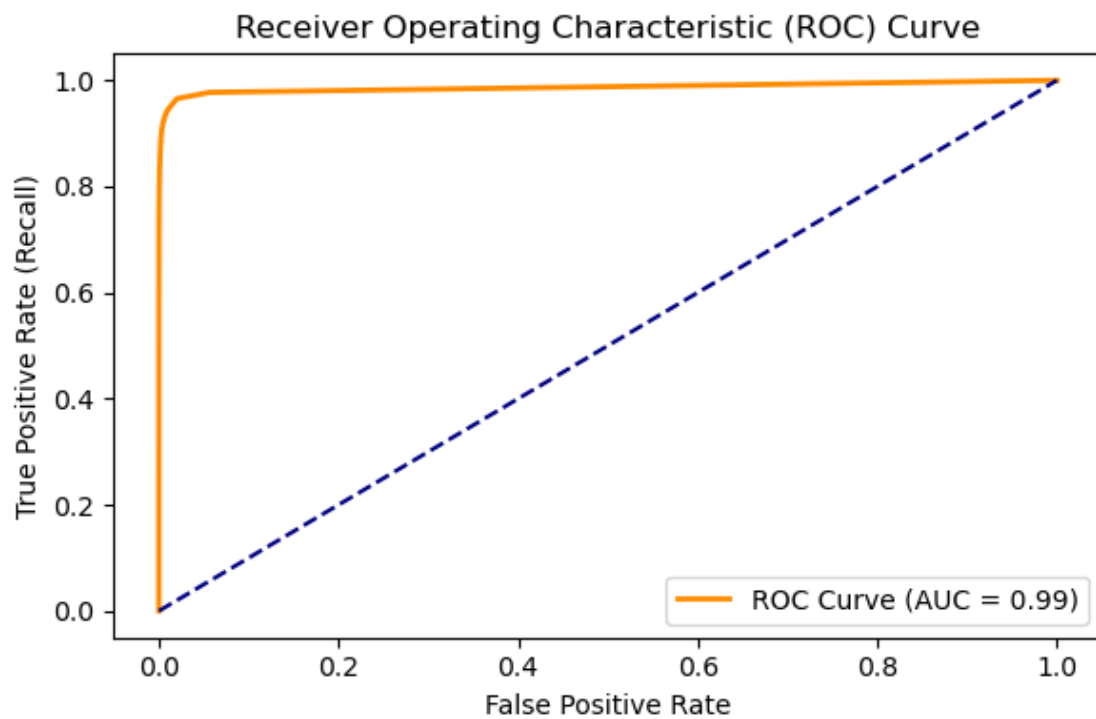


Confusion Matrix:

	Predicted: Not Fraud	Predicted: Fraud
Actual: Not Fraud	176961	37
Actual: Fraud	274	739



ROC AUC Score: 0.9868173952852625



Based on the model results, the classifier performed very well overall. The precision for fraud (class

1) is high at 0.95 (95%), meaning that when the model flags a transaction as fraudulent, it is correct 95% of the time. The recall for fraud is moderate at 0.73 (73%), indicating the model successfully identifies nearly three-quarters of all actual fraud cases. This suggests that while the model is quite accurate when predicting fraud, it still misses about one-quarter of fraudulent transactions.

The F1-score for fraud is 0.83, showing a good balance between precision and recall. The ROC AUC score of 0.987 further confirms that the model does an excellent job distinguishing between fraud and non-fraud cases across different classification thresholds. The ROC curve, which rises sharply toward the top-left corner, visually supports this strong discriminative performance.

Given the dataset's high class imbalance, additional metrics like the Precision-Recall curve and Average Precision Score (discussed earlier) provide a more focused evaluation of the minority class performance. These combined evaluations suggest the model is robust, but there remains room for improvement, especially in recall. Approaches such as threshold tuning, resampling techniques, or using more advanced algorithms could help enhance the model's ability to detect more fraudulent transactions without significantly sacrificing precision.

### 3.0.5 Precision-Recall Curve

Given the significant class imbalance in the dataset—where the majority of transactions are non-fraudulent—metrics like accuracy and ROC AUC can sometimes be misleading. The Precision-Recall curve provides a more informative evaluation by focusing specifically on the model's performance with respect to the minority class (fraudulent transactions).

This curve highlights the crucial trade-off between precision (the accuracy of fraud predictions) and recall (the ability to detect all fraud cases). In fraud detection, maximizing recall is important to catch as many fraudulent transactions as possible, while maintaining high precision helps minimize false alarms.

The curve was generated using scikit-learn's `precision_recall_curve` function to visualize this balance and guide further model tuning.

```
[56]: from sklearn.metrics import precision_recall_curve, average_precision_score
import matplotlib.pyplot as plt

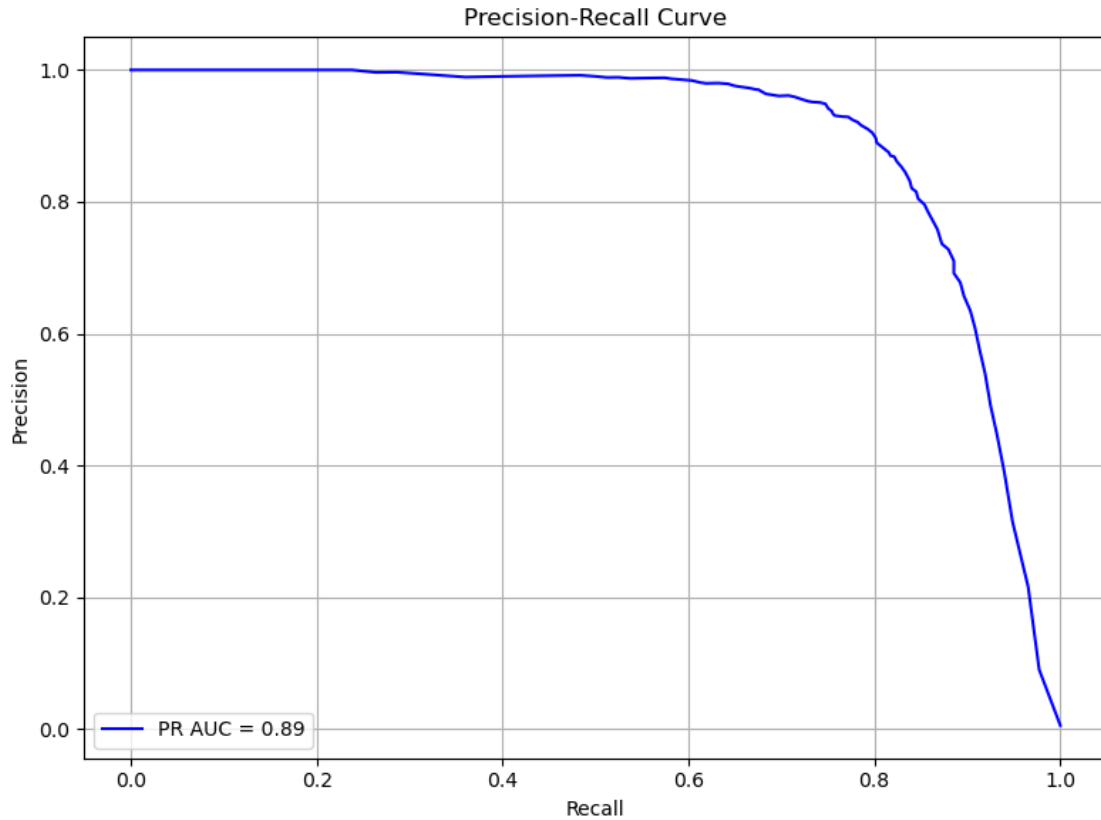
# Calculate the Average Precision Score (area under the Precision-Recall curve)
avg_precision = average_precision_score(y_test, y_pred_proba)
print(f"Precision-Recall AUC (Average Precision): {avg_precision:.4f}")

# Generate precision and recall values for different thresholds
precision, recall, _ = precision_recall_curve(y_test, y_pred_proba)

# Plot Precision-Recall curve
plt.figure(figsize=(8,6))
plt.plot(recall, precision, label=f'PR AUC = {avg_precision:.2f}', color='blue')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.legend(loc='lower left')
plt.grid(True)
```

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

Precision-Recall AUC (Average Precision): 0.8943



### 3.0.6 Summary of the Precision-Recall Curve Result

The Precision-Recall AUC (also known as Average Precision) is approximately 0.89 (89%), indicating the model effectively balances precision and recall across various decision thresholds. This metric is particularly valuable for detecting credit card fraud in a highly imbalanced dataset, where traditional metrics like accuracy or ROC AUC can be misleading.

Examining the curve, precision remains very high — around 0.98 to 0.99 — through recall levels up to about 0.3. Beyond this point, the precision curve experiences a slight decline but stays near these high levels until recall reaches approximately 0.6. After recall passes 0.6, precision begins to decrease more noticeably, with an accelerated drop starting around 0.7 recall. By the time recall reaches 1.0, precision falls sharply toward zero, reflecting the increasing number of false positives as the model tries to identify every fraud case.

This trade-off between precision and recall is common in imbalanced classification problems. The Precision-Recall curve thus offers a focused evaluation of the model's performance on the minority (fraud) class, providing more insight than ROC AUC in this context.

## 3.1 DSC 550 Term Project Milestone 4: Final Submission Additions

### 3.1.1 Updates Based on Feedback from Milestone 3

- Added an additional model: XGBoostClassifier
- Performed hyperparameter tuning using GridSearchCV on RandomForestClassifier
- Evaluated new model(s) and compared performance metrics to previous results

### 3.1.2 Step 1: Train and Evaluate XGBoost Model

```
[57]: import xgboost as xgb
from sklearn.metrics import classification_report, confusion_matrix, \
    roc_auc_score, precision_recall_curve, average_precision_score
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

# Calculating scale_pos_weight to handle class imbalance
neg, pos = y_train.value_counts()
scale_pos_weight = neg / pos

# Converting to DMatrix format for xgb.train
dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)

# XGBoost parameters
params = {
    'objective': 'binary:logistic',
    'eval_metric': 'logloss',
    'scale_pos_weight': scale_pos_weight,
    'max_depth': 4,
    'eta': 0.1,
    'seed': 42
}

# List of evaluation sets for early stopping
evals = [(dtrain, 'train'), (dtest, 'eval')]

# Train the model with early stopping (no verbose logs)
bst = xgb.train(
    params=params,
    dtrain=dtrain,
    num_boost_round=100,
    evals=evals,
    early_stopping_rounds=10,
    verbose_eval=False
)
```

```

# Predict on test set
y_pred_proba = bst.predict(dtest)
y_pred = (y_pred_proba >= 0.5).astype(int)

# Classification report
print("XGBoost Classification Report:\n", classification_report(y_test, y_pred,
    ↪zero_division=0))

# Confusion matrix (numeric for heatmap)
cm = confusion_matrix(y_test, y_pred)

# Formatted annotation (with commas)
annot = [[f"{val:,"} for val in row] for row in cm]

# Heatmap
plt.figure(figsize=(6, 4))
sns.heatmap(cm, annot=annot, fmt='', cmap='Blues',
            xticklabels=['Predicted: Not Fraud', 'Predicted: Fraud'],
            yticklabels=['Actual: Not Fraud', 'Actual: Fraud'])
plt.title('XGBoost Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.tight_layout()
plt.show()

# ROC AUC
roc_auc = roc_auc_score(y_test, y_pred_proba)
print(f"XGBoost ROC AUC Score: {roc_auc:.4f}")

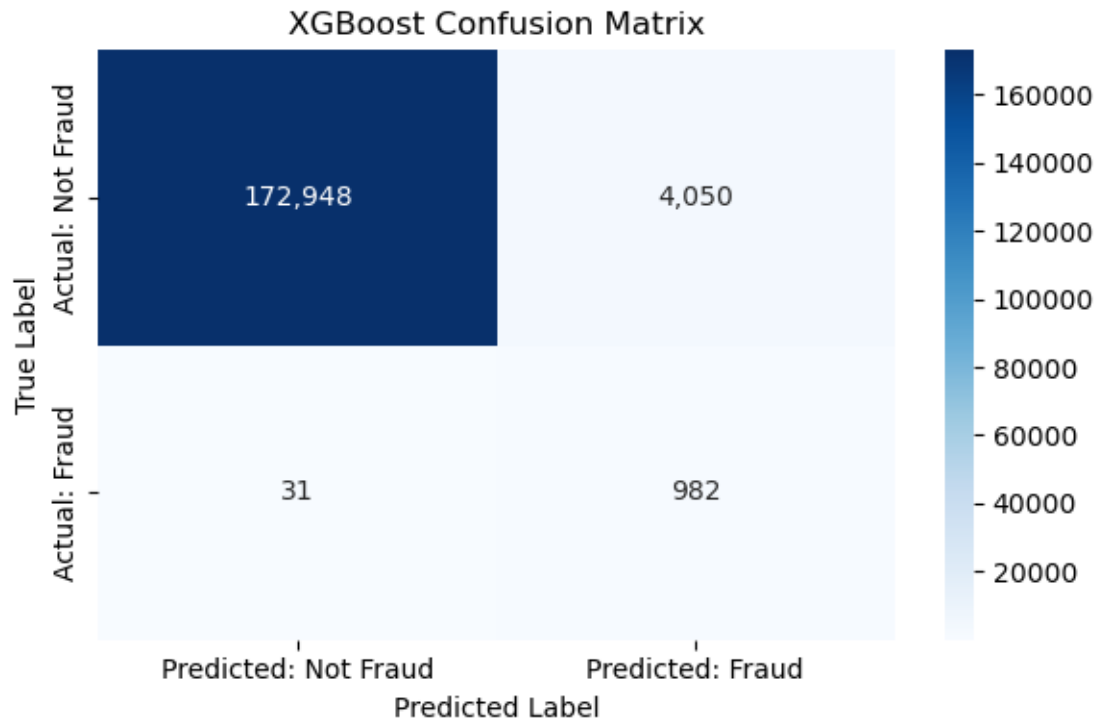
# Precision-Recall AUC
avg_precision = average_precision_score(y_test, y_pred_proba)
print(f"XGBoost Precision-Recall AUC (Average Precision): {avg_precision:.4f}")

# Precision-Recall curve plot
precision, recall, _ = precision_recall_curve(y_test, y_pred_proba)
plt.figure(figsize=(8, 6))
plt.plot(recall, precision, label=f'XGBoost PR AUC = {avg_precision:.2f}',
    ↪color='green')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('XGBoost Precision-Recall Curve')
plt.legend(loc='lower left')
plt.grid(True)
plt.tight_layout()
plt.show()

```

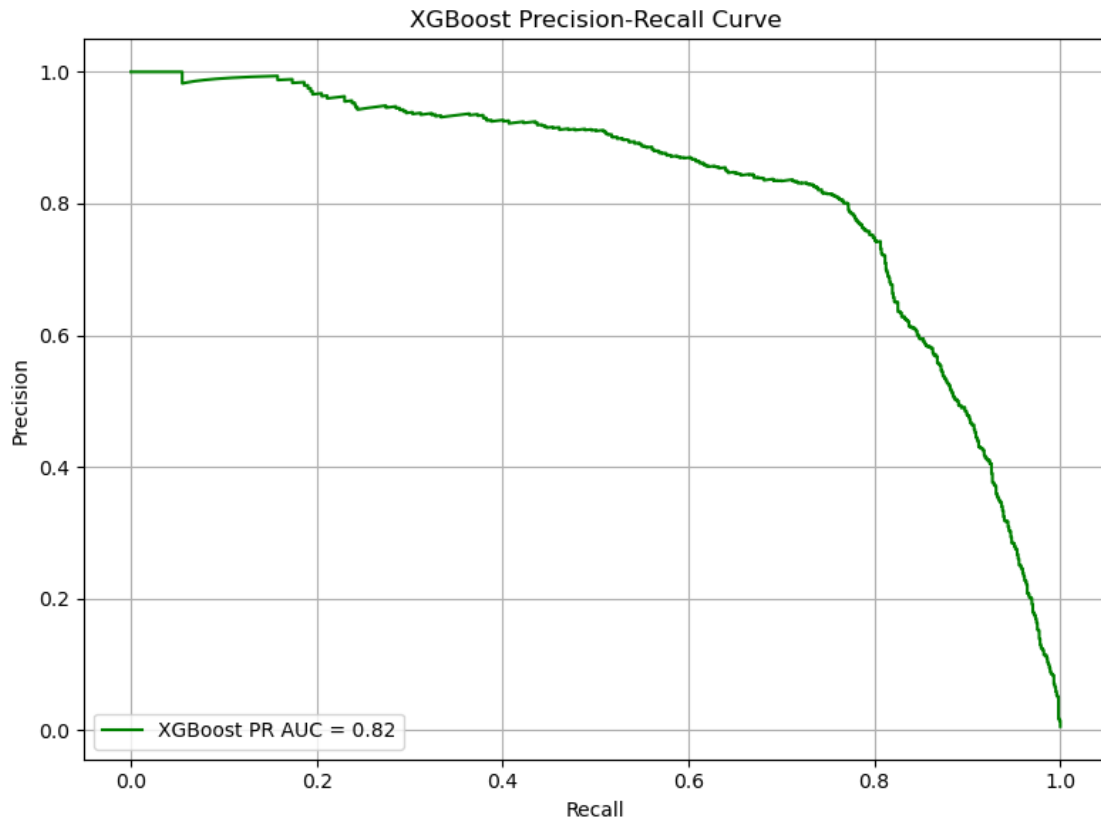
XGBoost Classification Report:

	precision	recall	f1-score	support
0	1.00	0.98	0.99	176998
1	0.20	0.97	0.32	1013
accuracy			0.98	178011
macro avg	0.60	0.97	0.66	178011
weighted avg	1.00	0.98	0.98	178011



XGBoost ROC AUC Score: 0.9965

XGBoost Precision-Recall AUC (Average Precision): 0.8203



### 3.1.3 Step 2: Hyperparameter Tuning: Random Forest Example

```
[58]: from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report, roc_auc_score

# Parameter grid
param_grid = {
    'n_estimators': [100],
    'max_depth': [10, 20],
    'min_samples_split': [2],
    'min_samples_leaf': [1]
}

# Grid search with class_weight='balanced' and average_precision scoring for
↳ imbalanced data
rf = RandomForestClassifier(random_state=42, class_weight='balanced')

grid_search = GridSearchCV(estimator=rf, param_grid=param_grid,
```



```

scoring='average_precision', cv=2, n_jobs=-1,
↳verbose=1)
grid_search.fit(X_train, y_train)

# Best model
best_rf = grid_search.best_estimator_
rf_preds = best_rf.predict(X_test)
rf_probs = best_rf.predict_proba(X_test)[:, 1]

# Evaluation
print("Classification Report for Tuned Random Forest:")
print(classification_report(y_test, rf_preds, zero_division=0))
print(f"Random Forest ROC AUC Score: {roc_auc_score(y_test, rf_probs):.4f}")
print(f"Random Forest Precision-Recall AUC (Average Precision): {grid_search.
↳best_score_:.4f}")

```

Fitting 2 folds for each of 2 candidates, totalling 4 fits

Classification Report for Tuned Random Forest:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	176998
1	0.78	0.82	0.80	1013
accuracy			1.00	178011
macro avg	0.89	0.91	0.90	178011
weighted avg	1.00	1.00	1.00	178011

Random Forest ROC AUC Score: 0.9954

Random Forest Precision-Recall AUC (Average Precision): 0.8347

### 3.1.4 Step 3: Optimize Classification Threshold for Random Forest

```

[59]: import numpy as np
from sklearn.metrics import precision_recall_curve, classification_report

precision, recall, thresholds = precision_recall_curve(y_test, rf_probs)

# Finding threshold that maximizes F1 score
f1_scores = 2 * (precision * recall) / (precision + recall + 1e-8)
best_idx = np.argmax(f1_scores)
best_threshold = thresholds[best_idx]

print(f"Best threshold for RF by F1: {best_threshold:.3f}")

# Recalculating predictions with best threshold
y_pred_best = (rf_probs >= best_threshold).astype(int)

```

```
print("Classification report with best threshold:\n",
      ↪classification_report(y_test, y_pred_best, zero_division=0))
```

Best threshold for RF by F1: 0.609

Classification report with best threshold:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	176998
1	0.88	0.77	0.82	1013
accuracy			1.00	178011
macro avg	0.94	0.88	0.91	178011
weighted avg	1.00	1.00	1.00	178011

### 3.1.5 Step 4. Model Performance Comparison Table

Model	Precision (Fraud)	Recall (Fraud)	F1-Score (Fraud)	ROC AUC	PR AUC	Notes
Milestone 3: Baseline RF	0.95	0.73	0.83	-	0.8943	Strong baseline, slight recall drop
Milestone 4: Tuned RF	0.78	0.82	0.80	0.9954	0.8347	Improved recall, slightly lower F1
Milestone 4: Tuned RF + Thresh	0.88	0.77	0.82	-	-	Balanced trade-off via thresholding
Milestone 4: XGBoost	0.20	0.97	0.32	0.9965	0.8203	Very high recall, very low precision

### 3.1.6 Summary of Model Evaluation and Findings

In this project, I evaluated multiple approaches for fraud detection using a highly imbalanced credit card transactions dataset. The baseline Random Forest model from Milestone 3 demonstrated strong performance, achieving an F1-score of 0.83 (83%) for the minority class and a Precision-Recall AUC of 0.8943 (89.43%).

In Milestone 4, hyperparameter tuning further optimized the model, yielding a slightly lower F1-score of 0.80 (80%), but improving recall to 0.82 (82%), indicating better fraud capture rates. To enhance performance further, threshold adjustment was applied to the tuned model, producing a more balanced precision of 0.88 (88%) and recall of 0.77 (77%), resulting in an F1-score of 0.82 (82%).

In contrast, the XGBoost model prioritized recall (0.97 or 97%) but significantly compromised on precision (0.20 or 20%), achieving a low F1-score of 0.32 (32%) despite a high ROC AUC of 0.9965 (99.65%). While XGBoost is aggressive in identifying potential frauds, it may be unsuitable for production deployment due to its high rate of false positives.

Overall, the tuned Random Forest with threshold optimization offered the best trade-off between fraud detection and minimizing false alarms, making it the most balanced and practical model for

deployment.