Report

Financial Fraud Detection Using Batch Processing: Project Documentation

Overview

This document provides a detailed guide for **Project 2: Financial Fraud Detection Using Batch Processing**. The project involves developing a batch processing system to detect fraudulent transactions in a financial dataset. The process encompasses data ingestion from a relational database into Hadoop, data transformation using Spark, querying with Hive, applying machine learning for fraud detection, and visualizing the results with a business intelligence (BI) tool.

Key Tasks

1. Data Ingestion: Import financial transactions data into Hadoop HDFS using Sqoop from a MySQL database.
2. Data Processing & Transformation: Use Apache Spark to process and transform the ingested data for further analysis.
3. Querying: Utilize Hive to query the data and store processed results.
4. Machine Learning: Implement machine learning algorithms, such as decision trees or support vector machines (SVM), to detect fraudulent activities.
5. Visualization & Reporting: Visualize the detection results using a BI tool, generating insightful reports for decision-making.

Technologies

- Sqoop: For data ingestion from MySQL into Hadoop.
- Hadoop: To store and manage large volumes of historical transaction data.
- Apache Spark: To process and transform the ingested data.
- Hive: For querying and analyzing the transformed data.
- BI Tool: To visualize and report on fraud detection insights.
- ML Algorithms: Decision trees, SVM, and other algorithms for detecting fraudulent patterns.

Data Ingestion

1. Data Preparation

1.1 Analyzing Column Types

The initial step involves analyzing the CSV file to determine appropriate MySQL data types for each column. This is achieved using Python and the `pandas` library.

```
import pandas as pd

# Load CSV file
df = pd.read_csv('fraudTrain.csv')

# Analyze columns and suggest MySQL data types
def suggest_mysql_dtype(df):
    type_mapping = {
        'int64': 'INT',
        'float64': 'FLOAT',
        'object': 'VARCHAR(255)',
        'bool': 'TINYINT(1)',
        'datetime64[ns]': 'DATETIME'
    }

    for column, dtype in df.dtypes.items():
        mysql_dtype = type_mapping.get(str(dtype), 'VARCHAR(255)')
        print(f"Column: {column}, Pandas Type: {dtype}, Suggested MySQL
Type: {mysql_dtype}")

suggest_mysql_dtype(df)
```

Resulting Data Types:

Column	Pandas Type	Suggested MySQL Type
Unnamed: 0	int64	INT
trans_date_trans_time	object	VARCHAR(255)
cc_num	int64	INT
merchant	object	VARCHAR(255)
category	object	VARCHAR(255)
amt	float64	FLOAT
first	object	VARCHAR(255)
last	object	VARCHAR(255)
gender	object	VARCHAR(255)
street	object	VARCHAR(255)
city	object	VARCHAR(255)
state	object	VARCHAR(255)
zip	int64	INT
lat	float64	FLOAT
long	float64	FLOAT
city_pop	int64	INT
job	object	VARCHAR(255)
dob	object	VARCHAR(255)
trans_num	object	VARCHAR(255)
unix_time	int64	INT
merch_lat	float64	FLOAT
merch_long	float64	FLOAT
is_fraud	int64	INT

Skip to 2. MySQL Database Setup if you downloaded the modified file :

SIC_fraudTrain_modified (kaggle.com)

If you wish to modify it yourself follow along:-

1.2 Modifying the CSV File

To ensure consistency with the MySQL table schema, rename the first column `Unnamed: 0` to `id` and save the modified CSV file.

```
# Rename the first column
df.rename(columns={'Unnamed: 0': 'id'}, inplace=True)

# Save the modified CSV file
df.to_csv('fraudTrain_modified.csv', index=False)
```

“ on the VM use this command to make the required directories”

```
mkdir -p /home/student/fraudDetection/Dataset
```

Then drag and drop the file from local machine to this path

2. MySQL Database Setup

2.1 Log in to MySQL

```
mysql --user=student --password=student
```

2.2 Create Database and Table

Execute the following SQL commands to create a database and a table with the appropriate schema.

```
CREATE DATABASE financial_fraud;
USE financial_fraud;

CREATE TABLE transactions (
    id INT PRIMARY KEY ~~AUTO_INCREMENT~~,
    trans_date_trans_time VARCHAR(255),
    cc_num BIGINT,
    merchant VARCHAR(255),
    category VARCHAR(255),
    amt FLOAT,
    first VARCHAR(255),
    last VARCHAR(255),
    gender VARCHAR(255),
    street VARCHAR(255),
    city VARCHAR(255),
    state VARCHAR(255),
    zip INT,
    lat FLOAT,
    lon FLOAT,
    city_pop INT,
    job VARCHAR(255),
    dob DATE,
    trans_num VARCHAR(255),
    unix_time INT,
    merch_lat FLOAT,
    merch_long FLOAT,
    is_fraud TINYINT(1)
);
```

2.3 Load Data into the Table

Use the following command to load data from the modified CSV file into the `transactions` table.

```
LOAD DATA LOCAL INFILE
'/home/student/fraudDetection/Dataset/fraudTrain_modified.csv'
INTO TABLE transactions
FIELDS TERMINATED BY ','
OPTIONALLY ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS
(id, trans_date_trans_time, cc_num, merchant, category, amt, first, last,
gender, street, city, state, zip, lat, lon, city_pop, job, dob, trans_num,
unix_time, merch_lat, merch_long, is_fraud);
```

3. Import Data from MySQL to Hadoop

3.1 Sqoop Import Command

Use Sqoop to import data from the MySQL `transactions` table into Hadoop. Note that no `target-dir` is specified, so data will be imported to the default directory `/user/student/transactions/`.

```
sqoop import \
    --connect jdbc:mysql://localhost/financial_fraud \
    --username student \
    --password student \
    --table transactions \
    --split-by id \
    --incremental append \
    --check-column id \
    --last-value 0 \
    --as-parquetfile \
    --verbose
```

Data Processing & Transformation

1. Importing libraries & necessary steps

1.1 Importing libraries

```
# Install necessary packages
!pip install seaborn
!pip install sklearn

# Import libraries for data manipulation and visualization
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Import Spark SQL functions
from pyspark.sql.functions import col, from_unixtime, to_date, when, sum, count, log

# Import Spark MLlib features and models
from pyspark.ml.feature import VectorAssembler, StringIndexer, OneHotEncoder
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
```

1.2 Reading parquet file

```
df = spark.read.parquet("/user/student/transactions")
df.show(5, truncate=False)
```

1.3 Schema

Found there are some wrong data type

```
df.printSchema()
```

```
root
 |-- id: integer (nullable = true)
 |-- trans_date_trans_time: string (nullable = true)
 |-- cc_num: long (nullable = true)
 |-- merchant: string (nullable = true)
 |-- category: string (nullable = true)
 |-- amt: float (nullable = true)
 |-- first: string (nullable = true)
 |-- last: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- street: string (nullable = true)
 |-- city: string (nullable = true)
 |-- state: string (nullable = true)
 |-- zip: integer (nullable = true)
 |-- lat: float (nullable = true)
 |-- lon: float (nullable = true)
 |-- city_pop: integer (nullable = true)
 |-- job: string (nullable = true)
 |-- dob: long (nullable = true)
 |-- trans_num: string (nullable = true)
 |-- unix_time: integer (nullable = true)
 |-- merch_lat: float (nullable = true)
 |-- merch_long: float (nullable = true)
 |-- is_fraud: boolean (nullable = true)
```

Conclusion:

Data is saved in HDFS as multiple parquet file so it was imported into one dataframe finding that there was some mistakes in data types

2. Transformations

2.1 Change some column's data type

2.1.1 Change dob column

```
: # found dob column is long instead of date type
df.select("dob").show(5, truncate=False)

# Convert 'dob' from long to date and overwrite the 'dob' column
df = df.withColumn("dob", to_date(from_unixtime(col("dob") / 1000)))

# Show the updated DataFrame with the 'dob' column replaced
df.select("dob").show(5, truncate=False)
```

2.1.2 Change unix_time column

```
# found unix_time column is integer instead of timestamp type
df.select("unix_time").show(5, truncate=False)
# Convert 'unix_time' from integer (seconds) to timestamp
df = df.withColumn("unix_time", from_unixtime(col("unix_time")).cast("timestamp"))

# Show the updated DataFrame with the 'unix_time' column converted
df.select("unix_time").show(5, truncate=False)
```

2.1.3 Change is_fraud column

```
: # found is_fraud column is boolean instead of integer type containing 0s and 1s
df.select("is_fraud").show(5, truncate=False)

# Convert 'is_fraud' from boolean to integer (0 and 1)
df = df.withColumn("is_fraud", when(col("is_fraud") == True, 1).otherwise(0))

# Show the updated DataFrame with the 'is_fraud' column replaced
df.select("is_fraud").show(5, truncate=False)
```

2.2 From sparkDataFrame to pandas DataFrame for better understanding

```
n [7]: # Convert to Pandas DataFrame
pandas_df = df.limit(5).toPandas()

# Display as a table
print(pandas_df)
```

Conclusion:

1) *dob type was changed from long to date type*
2) *unix_time type was changed from integer to timestamp type*
3) *is_fraud type was changed from boolean to integer containing binary values*
4) *turn spark dataframe into pandas for better visualization*

3. Data Cleaning:

3.1 Check for Nulls:

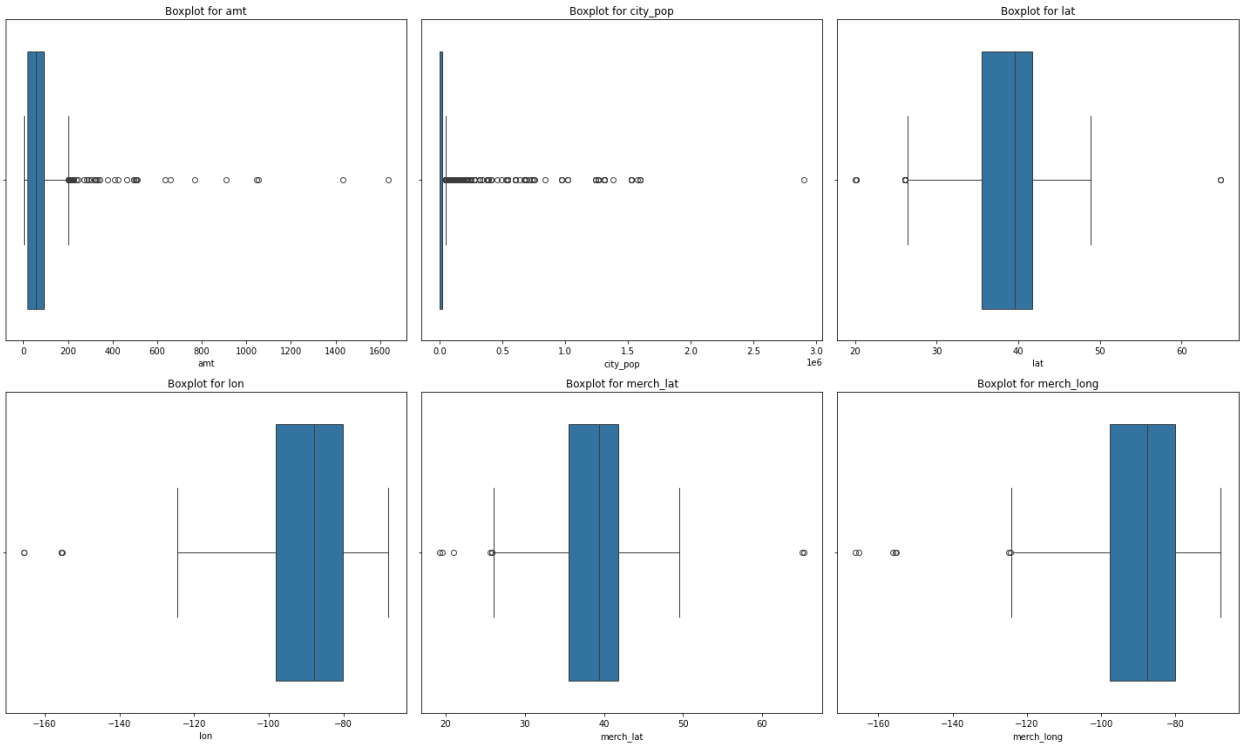
```
# Calculate null counts for each column
null_counts = df.select([sum(col(column).isNull().cast("int")).alias(column) for column in df.columns])

# Collect the result as a dictionary
null_counts_dict = null_counts.collect()[0].asDict()

# Print column names and their null counts
for column, count in null_counts_dict.items():
    print(f"Column '{column}' has {count} null values.")
```

3.2 Check for Duplicates:

```
# Create a DataFrame with a column for duplicate counts
duplicates_df = df.groupBy(df.columns).count().filter(col("count") > 1)

# Calculate the total number of duplicates for each column
duplicate_counts = {}
for column in df.columns:
    # Count duplicates by grouping on the column and counting occurrences
    count_duplicates = df.groupBy(column).count().filter(col("count") > 1).count()
    duplicate_counts[column] = count_duplicates

# Print the column names and their duplicate counts
for column, count in duplicate_counts.items():
    print(f"Column '{column}' has {count} duplicate values.")
```

3.3 Exploring Numeric Data with Box Plots:

```
# Limit the data to a manageable size
pandas_df = df.limit(1000).toPandas()

# List of numeric columns to visualize
numeric_columns = ["amt", "city_pop", "lat", "lon", "merch_lat", "merch_long"]

# Set up a figure to hold multiple plots
plt.figure(figsize=(20, 12))

# Create a box plot for each numeric column to visualize outliers
for i, column in enumerate(numeric_columns, 1):
    plt.subplot(2, 3, i)  # Adjust the grid size according to the number of columns
    sns.boxplot(data=pandas_df, x=column)
    plt.title(f'Boxplot for {column}')

# Display the plots
plt.tight_layout()
plt.show()
```



Conclusion:

1) *This data doesn't contain any nulls*
2) *Also duplicates existence make sense*
3) *There is no unusual outliers*

4. Querying

4.1 Total Fraudulent Transactions by Merchant

```
#Total Fraudulent Transactions by Merchant
df.filter(df.is_fraud == 1) \
  .groupBy("merchant") \
  .count() \
  .orderBy("count", ascending=False) \
  .show()
```

4.2 Top Cities by Number of Fraudulent Transactions

```
#Top Cities by Number of Fraudulent Transactions
df.filter(df.is_fraud == 1) \
  .groupBy("city") \
  .count() \
  .orderBy("count", ascending=False) \
  .limit(10) \
  .show()
```

4.3 Fraudulent Transactions by Time of Day

```
#Fraudulent Transactions by Time of Day
from pyspark.sql.functions import hour

df.filter(df.is_fraud == True) \
  .withColumn("transaction_hour", hour("trans_date_trans_time")) \
  .groupBy("transaction_hour") \
  .count() \
  .orderBy("count", ascending=False) \
  .show()
```

4.4 Average Transaction Amount for Fraudulent vs

```
#Average Transaction Amount for Fraudulent vs. Non-Fraudulent Transactions
df.groupBy("is_fraud") \
  .agg({"amt": "avg"}) \
  .withColumnRenamed("avg(amt)", "avg_transaction_amount") \
  .show()
```

Non-Fraudulent Transactions

4.5 Fraud Rate by Category

```
#Fraud Rate by Category
from pyspark.sql.functions import sum as _sum

df.groupBy("category") \
  .agg((_sum(df.is_fraud.cast("int")) * 100 / _sum(df.is_fraud.cast("int") + (1 - df.is_fraud.cast("int")))).alias(
  .orderBy("fraud_rate", ascending=False) \
  .show()
```

4.6 Find the Most Common Job Among Fraudsters

```
#Find the Most Common Job Among Fraudsters
df.filter(df.is_fraud == 1) \
  .groupBy("job") \
  .count() \
  .orderBy("count", ascending=False) \
  .limit(10) \
  .show()
```

4.7 find if fraud is affected by gender

```
#find if fraud is affected by gender
df.filter(df.is_fraud == 1) \
  .groupBy("gender") \
  .count() \
  .orderBy("count", ascending=False) \
  .limit(10) \
  .show()
```

Conclusion:

1) *Houston is number one in Fraudulent Transactions while Allentown take the last place*
2) *Most Fraudulent Transactions happen between 22:00 and 1:00 Am*
3) *Most Fraudulent Transactions happen in categories like shopping_net,misc_net, and grocery_pos*
4) *There are Common jobs among Fraudsters like Materials engineering, Trading standards.,Naval architect and Exhibition designer*
5) *Fraud does not affected by gender as their proportions are almost equal*

5. Preparations for machine learning

(Note: this part was done with the great help of Eng. Ossama Taha):

5.1 Indexing and Encoding Categorical Data for Machine Learning

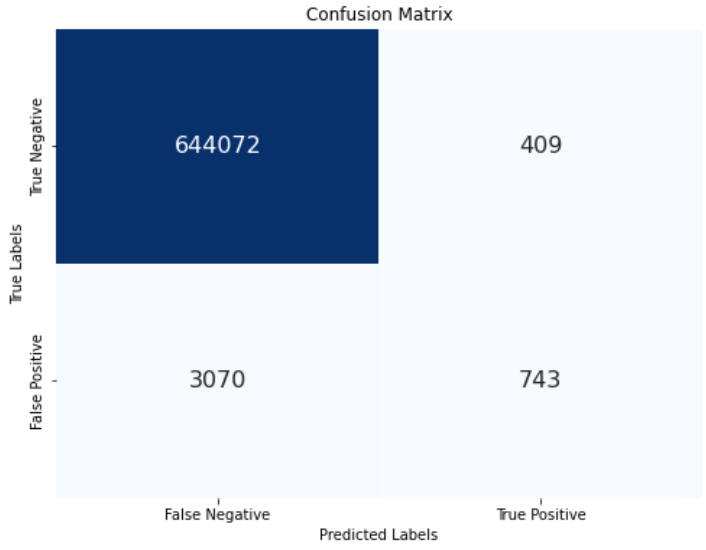
```
14]: # Added a new column to calculate the transaction hour
transformed_df = df.withColumn("trans_hour", col("trans_date_trans_time").substr(12, 2).cast('int'))


# List of categorical columns
categorical_columns = ['merchant', 'category', 'gender', 'city', 'state', 'job']

# Index and encode categorical columns
indexers = []
encoders = []

for col_name in categorical_columns:
    indexer = StringIndexer(inputCol=col_name, outputCol=col_name + "_index")
    encoder = OneHotEncoder(inputCols=[col_name + "_index"], outputCols=[col_name + "_encoded"])

    indexers.append(indexer)
    encoders.append(encoder)

    transformed_df = indexer.fit(transformed_df).transform(transformed_df)
    transformed_df = encoder.fit(transformed_df).transform(transformed_df)

# Select feature columns, including encoded categorical features
feature_columns = ['amt', 'trans_hour', 'city_pop', 'lat', 'lon', 'merch_lat', 'merch_long'] \
    + [col + "_encoded" for col in categorical_columns]

# Apply VectorAssembler
assembler = VectorAssembler(inputCols=feature_columns, outputCol="features")
ml_data = assembler.transform(transformed_df)

# Show the resulting DataFrame with the "features" vector column
```

5.2 Splitting Data into Training and Testing Sets for Model Evaluation

```
target_column = 'is_fraud'
train_data, test_data = ml_data.randomSplit([0.5, 0.5])
```

5.3 Training and Evaluating the Logistic Regression Model

```
# Initialize and train the Logistic Regression model
lr = LogisticRegression(featuresCol='features', labelCol=target_column)
lr_model = lr.fit(train_data)

# Make predictions
lr_predictions = lr_model.transform(test_data)

# Initialize evaluators
accuracy_evaluator = MulticlassClassificationEvaluator(labelCol=target_column, predictionCol="prediction", metricNa
precision_evaluator = MulticlassClassificationEvaluator(labelCol=target_column, predictionCol="prediction", metricN
recall_evaluator = MulticlassClassificationEvaluator(labelCol=target_column, predictionCol="prediction", metricName
f1_evaluator = MulticlassClassificationEvaluator(labelCol=target_column, predictionCol="prediction", metricName="f1

# Calculate metrics
accuracy = accuracy_evaluator.evaluate(lr_predictions)
precision = precision_evaluator.evaluate(lr_predictions)
recall = recall_evaluator.evaluate(lr_predictions)
f1_score = f1_evaluator.evaluate(lr_predictions)

# Print metrics
print(f"Logistic Regression Accuracy: {accuracy}")
print(f"Logistic Regression Precision: {precision}")
print(f"Logistic Regression Recall: {recall}")
print(f"Logistic Regression F1 Score: {f1_score}")

# Show a sample of predictions
lr_predictions.select("prediction", target_column, "features").show(10, truncate=False)
```



To view the full code and outputs you will find the notebook in the Kaggle link below :
https://www.kaggle.com/code/lobnasalah/data-processing-using-pyspark

Hive

First start hue service

```
sudo systemctl start hue
```

Then open it at localhost: http://localhost:8888

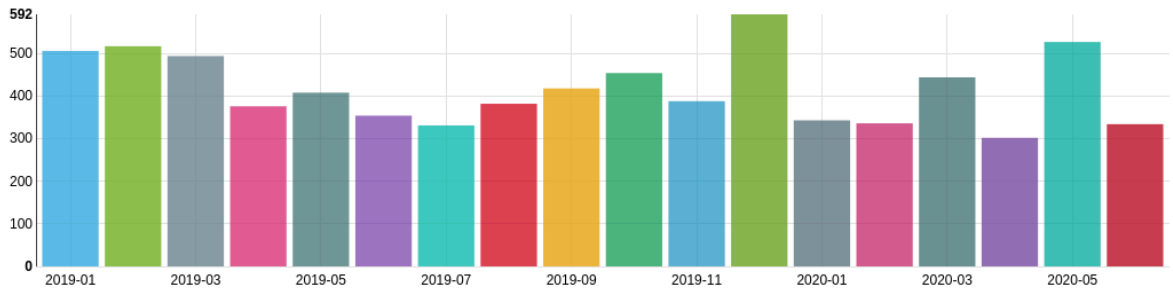
```
CREATE EXTERNAL TABLE transactions (
    id INT,
    trans_date_trans_time STRING,
    cc_num BIGINT,
    merchant STRING,
    category STRING,
    amt FLOAT,
    first STRING,
    last STRING,
    gender STRING,
    street STRING,
    city STRING,
    state STRING,
    zip INT,
    lat FLOAT,
    lon FLOAT,
    city_pop INT,
    job STRING,
    dob DATE,
    trans_num STRING,
    unix_time INT,
    merch_lat FLOAT,
    merch_long FLOAT,
    is_fraud TINYINT
)
STORED AS PARQUET
LOCATION '/user/student/transactions_new/updated_fraud_data_parquet';
```

After this you will have all the data loaded and ready.

Queries:

1. Monthly Trend of Fraudulent Transactions:

```sql
SELECT
    DATE_FORMAT(trans_date_trans_time, 'yyyy-MM') AS month,
    COUNT(*) AS fraudulent_transactions_count
FROM
    transactions
WHERE
    is_fraud = 1
GROUP BY
    DATE_FORMAT(trans_date_trans_time, 'yyyy-MM')
```

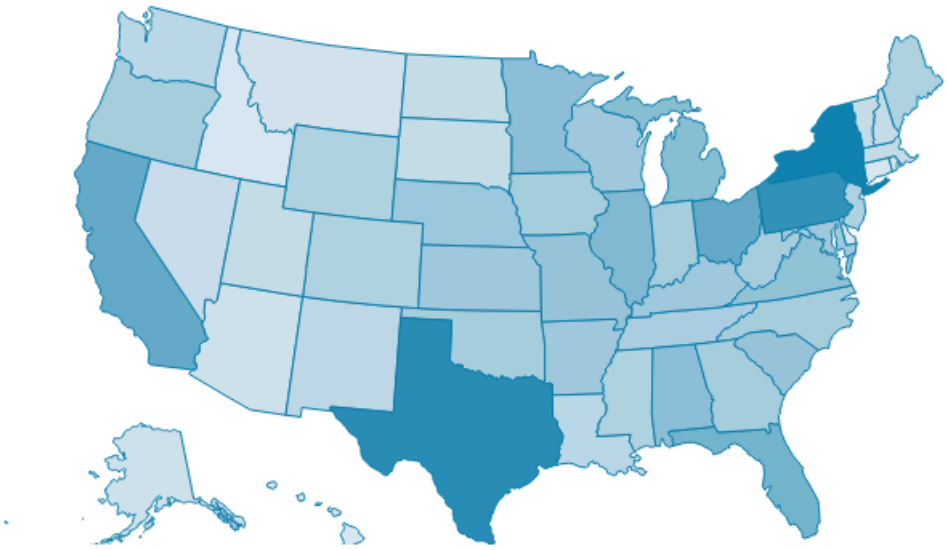


Conclusions:

- **Seasonality**: The spike in fraudulent transactions in **December 2019** may suggest seasonality, where fraud peaks during periods of higher transaction volumes (such as the holiday season).
- **Pandemic Impact**: The dip in fraudulent transactions in early 2020, especially in **April**, might be linked to the economic downturn and limited transaction activity during the onset of the COVID-19 pandemic.
- **Cyclic Nature of Fraud**: Fraudulent transactions do not follow a steady pattern but tend to fluctuate monthly, indicating external influences such as economic conditions or seasonal trends.

2. Distribution of Fraudulent Transactions by State:

```sql
SELECT
    state,
    COUNT(*) AS fraudulent_transactions_count
FROM
    transactions
WHERE
    is_fraud = 1
GROUP BY
    state
ORDER BY
    fraudulent_transactions_count DESC;
```

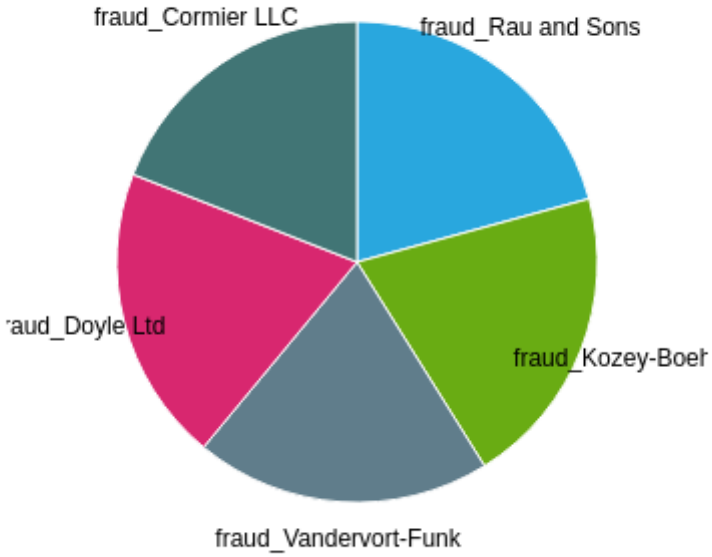


Conclusions:

- **Fraud Hotspots**: The data shows clear fraud hotspots in **New York**, **Texas**, and **Pennsylvania**, likely due to higher population density and transaction volume in these areas.
- **Regional Differences**: Southern states like **Florida**, **Alabama**, and **South Carolina** have moderate levels of fraud, while the **Midwest** and **Northeast** also see considerable cases. States in the **West** and **Mountain regions** generally show fewer fraudulent transactions.
- **Population Impact**: The results suggest a strong correlation between population size and the number of fraudulent transactions. States with larger populations naturally see more fraud. However, this is not an absolute rule, as **California**, despite its population size, ranks lower than expected.

3. Fraudulent Transactions by Merchant and Category

```sql
SELECT
    merchant,
    category,
    COUNT(*) AS fraudulent_transactions_count
FROM
    transactions
WHERE
    is_fraud = 1
GROUP BY
    merchant,
    category
ORDER BY
    fraudulent_transactions_count DESC;
```



Conclusions:

1. **Top Merchants**:
   - **fraud_Rau and Sons** (grocery_pos) has the highest count of fraudulent transactions with 49 cases.
   - **fraud_Kozey-Boehm** (shopping_net) and **fraud_Vandervort-Funk** (grocery_pos) are close behind with 48 and 47 cases, respectively.
   - The top merchants are predominantly involved in **grocery_pos** and **shopping_net** categories.
2. **Merchant Categories**:
   - The two most common categories for fraudulent transactions are **grocery_pos** and **shopping_net**.
   - Merchants in the **grocery_pos** category appear frequently in the top results, indicating a potentially higher incidence of fraud in this category compared to others.
   - **shopping_net** also has a high number of fraudulent transactions but generally appears slightly less frequently than **grocery_pos** in the top results.

Machine Learning (ML)

Code Breakdown : Notebook

1. Import Libraries

```
from pyspark.ml.feature import StringIndexer, OneHotEncoder,
VectorAssembler
from pyspark.sql.functions import col, when
from pyspark.sql.types import StructType, ArrayType
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import MulticlassClassificationEvaluator,
BinaryClassificationEvaluator
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix as sk_confusion_matrix
import pandas as pd
```

2. Model Training

a. Load Data

```
# Define the path to the CSV file in HDFS
csv_path = "hdfs:///user/student/transactions_new/updated_fraud_data_csv"

# Read the CSV file into a DataFrame
df = spark.read.csv(csv_path, header=True, inferSchema=True)
```

b. Add Transaction Hour Column

```
# Added a new column to calculate the transaction hour
transformed_df = df.withColumn("trans_hour",
col("trans_date_trans_time").substr(12, 2).cast('int'))
```

c. Categorical Feature Encoding

```
# List of categorical columns
categorical_columns = ['merchant', 'category', 'gender', 'city', 'state',
'job']

# Index and encode categorical columns
indexers = []
encoders = []

for col_name in categorical_columns:
    indexer = StringIndexer(inputCol=col_name, outputCol=col_name +
"_index")
    encoder = OneHotEncoder(inputCols=[col_name + "_index"],
outputCols=[col_name + "_encoded"])

    indexers.append(indexer)
    encoders.append(encoder)

    transformed_df = indexer.fit(transformed_df).transform(transformed_df)
    transformed_df = encoder.fit(transformed_df).transform(transformed_df)
```

d. Feature Engineering

```
# Select feature columns, including encoded categorical features
feature_columns = ['amt', 'trans_hour', 'city_pop', 'lat', 'lon',
'merch_lat', 'merch_long'] \
    + [col + "_encoded" for col in categorical_columns]

# Apply VectorAssembler
assembler = VectorAssembler(inputCols=feature_columns,
outputCol="features")
ml_data = assembler.transform(transformed_df)
```

e. Handling Class Imbalance

```
# Handling class imbalance by adding class weights
fraud_ratio = ml_data.filter(col('is_fraud') == 1).count() /
ml_data.count()
ml_data = ml_data.withColumn("class_weight", when(col("is_fraud") == 1, 1 /
fraud_ratio).otherwise(1.0))
```

f. Data Splitting

```
# Split the data into 50% train and 50% test
train_data, test_data = ml_data.randomSplit([0.5, 0.5])
```

g. Train Logistic Regression Model

```
# Initialize and train the Logistic Regression model with regularization
lr = LogisticRegression(featuresCol='features', labelCol='is_fraud',
weightCol="class_weight", regParam=0.01, elasticNetParam=0.8)
lr_model = lr.fit(train_data)
```

h. Make Predictions

```
# Make predictions on the test data
lr_predictions = lr_model.transform(test_data)
```

3. Model Evaluation

a. Metrics Calculation

```
# Initialize evaluators
accuracy_evaluator = MulticlassClassificationEvaluator(labelCol="is_fraud",
predictionCol="prediction", metricName="accuracy")
precision_evaluator =
MulticlassClassificationEvaluator(labelCol="is_fraud",
predictionCol="prediction", metricName="precisionByLabel")
recall_evaluator = MulticlassClassificationEvaluator(labelCol="is_fraud",
predictionCol="prediction", metricName="recallByLabel")
f1_evaluator = MulticlassClassificationEvaluator(labelCol="is_fraud",
predictionCol="prediction", metricName="f1")
roc_evaluator = BinaryClassificationEvaluator(labelCol="is_fraud",
rawPredictionCol="rawPrediction", metricName="areaUnderROC")

# Calculate metrics
accuracy = accuracy_evaluator.evaluate(lr_predictions)
precision = precision_evaluator.evaluate(lr_predictions,
{precision_evaluator.metricLabel: 1.0})  # for class 1 (fraud)
recall = recall_evaluator.evaluate(lr_predictions,
{recall_evaluator.metricLabel: 1.0})  # for class 1 (fraud)
f1_score = f1_evaluator.evaluate(lr_predictions)
roc_auc = roc_evaluator.evaluate(lr_predictions)
```

```
# Print metrics
print(f"Logistic Regression Accuracy: {accuracy}")
print(f"Logistic Regression Precision (Fraud Class): {precision}")
print(f"Logistic Regression Recall (Fraud Class): {recall}")
print(f"Logistic Regression F1 Score: {f1_score}")
print(f"Logistic Regression ROC-AUC: {roc_auc}")
```

b. Confusion Matrix

```
# Create and show the confusion matrix
confusion_matrix_df = lr_predictions.groupBy("is_fraud",
"prediction").count()

# Convert Spark DataFrame to Pandas DataFrame for confusion matrix
predictions_and_labels = lr_predictions.select(col("prediction"),
col("is_fraud")).rdd
prediction_df = predictions_and_labels.toDF(["prediction",
"label"]).toPandas()

# Compute confusion matrix
conf_matrix = sk_confusion_matrix(prediction_df["label"],
prediction_df["prediction"])

# Create a DataFrame for visualization
conf_matrix_df = pd.DataFrame(conf_matrix, index=["True Negative", "False
Positive"], columns=["False Negative", "True Positive"])

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix_df, annot=True, fmt='d', cmap='Blues', cbar=False,
annot_kws={"size": 16})
plt.title('Confusion Matrix')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.savefig("confusion_matrix_plot.png")  # Save the plot as an image file
plt.show()
```

4. Save Results

a. Save Confusion Matrix to HDFS

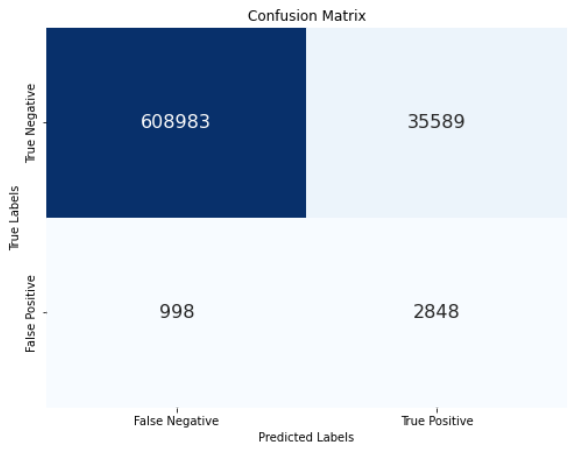
```
# Convert confusion matrix DataFrame to a Spark DataFrame
conf_matrix_spark_df = spark.createDataFrame(conf_matrix_df.reset_index())
# Save confusion matrix to CSV in a single partition
conf_matrix_spark_df.repartition(1).write.option("header",
"true").mode("overwrite").csv("hdfs:///user/student/transactions_new/confus
ion_matrix.csv")
```

b. Save Evaluation Metrics to HDFS

```
# Save evaluation metrics to CSV in a single partition (HDFS)
metrics_df = pd.DataFrame({
    "Metric": ["Accuracy", "Precision (Fraud Class)", "Recall (Fraud
Class)", "F1 Score", "ROC-AUC"],
    "Value": [accuracy, precision, recall, f1_score, roc_auc]
})

# Convert metrics DataFrame to Spark DataFrame
metrics_spark_df = spark.createDataFrame(metrics_df)

# Save evaluation metrics to CSV in a single partition
metrics_spark_df.repartition(1).write.option("header",
"true").mode("overwrite").csv("hdfs:///user/student/transactions_new/metric
s.csv")
```

Conclusion

This code provides a comprehensive pipeline for building, evaluating, and saving the results of a Logistic Regression model for fraud detection. It includes steps for data preprocessing, feature engineering, model training, evaluation, and visualization. Ensure to adjust paths and configurations as needed for your environment.



Business Intelligence (BI) Tools

1. Data Preparation

Prior to visualization, several data preparation steps were implemented to enhance the analysis and reporting:

- **Amount Bins**: Transaction amounts were categorized into bins of $1,000 to simplify analysis.
- **Age Calculation**: Age was calculated from the date of birth (`dob`) using DAX:
  `Age = YEAR(TODAY()) - YEAR([dob])`
- **Age Bins**: Age was further categorized into bins of 10 years to analyze fraud trends across different age groups
- **Job Field Categorization**: Jobs were categorized into more generic fields using DAX for slicing:

```
GenericJob =
SWITCH(
    TRUE(),
    // Engineering roles
    CONTAINSSTRING([Job], "Engineer"), "Engineering",

    // Science and Research roles
    CONTAINSSTRING([Job], "Scientist") || CONTAINSSTRING([Job],
"Research") || CONTAINSSTRING([Job], "Physicist") ||
CONTAINSSTRING([Job], "Geologist") || CONTAINSSTRING([Job],
"Biochemist"), "Science",

    // Management and Leadership roles
    CONTAINSSTRING([Job], "Manager") || CONTAINSSTRING([Job],
"Officer") || CONTAINSSTRING([Job], "Consultant") ||
CONTAINSSTRING([Job], "Executive") || CONTAINSSTRING([Job], "Chief")
|| CONTAINSSTRING([Job], "Comptroller") || CONTAINSSTRING([Job],
"Administrator"), "Management",

    // Technology and IT-related roles
    CONTAINSSTRING([Job], "Developer") || CONTAINSSTRING([Job],
"Programmer") || CONTAINSSTRING([Job], "Technologist") ||
CONTAINSSTRING([Job], "IT") || CONTAINSSTRING([Job], "Systems") ||
CONTAINSSTRING([Job], "Technology"), "Technology",
```

```
    // Psychology and Counseling roles
    CONTAINSSTRING([Job], "Psychologist") || CONTAINSSTRING([Job],
"Psychotherapist") || CONTAINSSTRING([Job], "Therapist") ||
CONTAINSSTRING([Job], "Counsellor") || CONTAINSSTRING([Job],
"Forensic"), "Psychology",

    // Healthcare and Medical roles
    CONTAINSSTRING([Job], "Doctor") || CONTAINSSTRING([Job], "Nurse")
|| CONTAINSSTRING([Job], "Medical") || CONTAINSSTRING([Job],
"Physicist") || CONTAINSSTRING([Job], "Radiographer") ||
CONTAINSSTRING([Job], "Podiatrist") || CONTAINSSTRING([Job],
"Health"), "Healthcare",

    // Education-related roles
    CONTAINSSTRING([Job], "Teacher") || CONTAINSSTRING([Job],
"Lecturer") || CONTAINSSTRING([Job], "Education") ||
CONTAINSSTRING([Job], "Trainer"), "Education",

    // Design, Arts, and Creative roles
    CONTAINSSTRING([Job], "Designer") || CONTAINSSTRING([Job],
"Artist") || CONTAINSSTRING([Job], "Architect") ||
CONTAINSSTRING([Job], "Exhibition") || CONTAINSSTRING([Job],
"Curator") || CONTAINSSTRING([Job], "Fine artist"), "Design and Arts",

    // Legal and Financial roles
    CONTAINSSTRING([Job], "Barrister") || CONTAINSSTRING([Job],
"Legal") || CONTAINSSTRING([Job], "Tax") || CONTAINSSTRING([Job],
"Finance") || CONTAINSSTRING([Job], "Accountant") ||
CONTAINSSTRING([Job], "Adviser"), "Legal and Finance",

    // Administration roles
    CONTAINSSTRING([Job], "Officer") || CONTAINSSTRING([Job],
"Inspector") || CONTAINSSTRING([Job], "Advisor") ||
CONTAINSSTRING([Job], "Surveyor"), "Administration",

    // Writing, Editing, and Communication roles
```

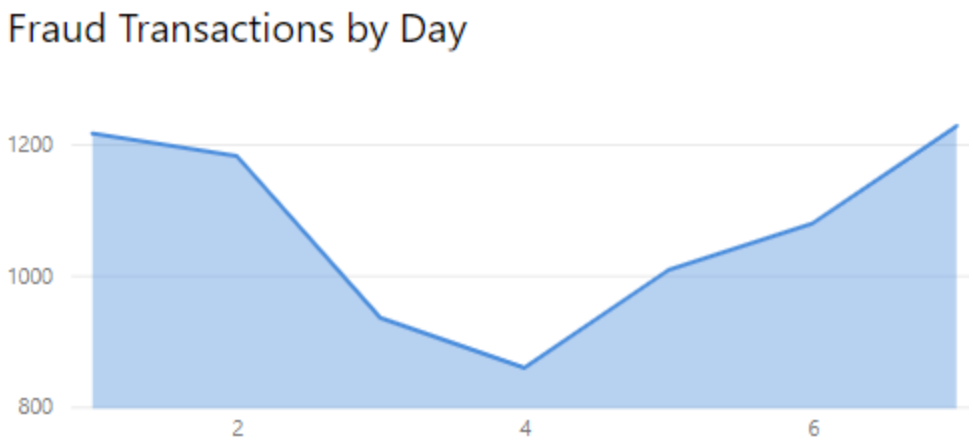
```
    CONTAINSSTRING([Job], "Writer") || CONTAINSSTRING([Job], "Editor")
|| CONTAINSSTRING([Job], "Communications") || CONTAINSSTRING([Job],
"Broadcast") || CONTAINSSTRING([Job], "Presenter") ||
CONTAINSSTRING([Job], "Copywriter") || CONTAINSSTRING([Job], "Public
relations"), "Writing and Communication",

    // Logistics and Distribution roles
    CONTAINSSTRING([Job], "Logistics") || CONTAINSSTRING([Job],
"Distribution") || CONTAINSSTRING([Job], "Freight"), "Logistics",

    // Other
    "Other"
)
```

- **Generic Categories**: Consolidated transaction categories into broader groups for more insightful analysis.
- **Hour Extraction**: Extracted the hour from the `trans_date_trans_time` for time-based analysis:
  `Hour = HOUR([trans_date_trans_time])`

2. Data Visualization

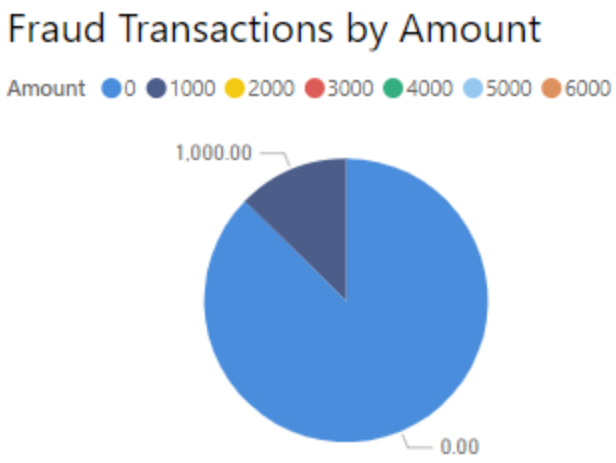
To effectively convey insights from the fraud detection data, I created a range of visualizations, each designed to highlight specific patterns and trends within the dataset:

- **Stacked Area Chart of Fraud by Day**: This visualization was essential in revealing temporal patterns in fraud transactions. It demonstrated that fraudulent activities spike toward the start and end of the week, with significantly fewer fraud transactions occurring midweek. Additionally, a drill-down feature was added to break down fraud occurrences by hour, highlighting peaks at hours 22 and 23.

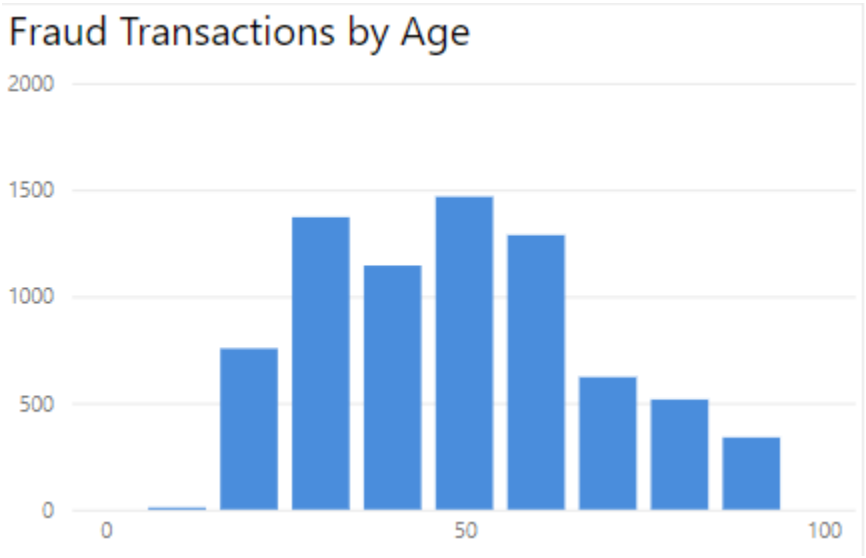


- **Line Chart of Fraud by Month**: The line chart provided a broader temporal view, showing a distinct pattern where fraud occurrences were significantly higher from January through May. A marked decrease was observed from July to December, with June acting as a transitional phase between the two periods.

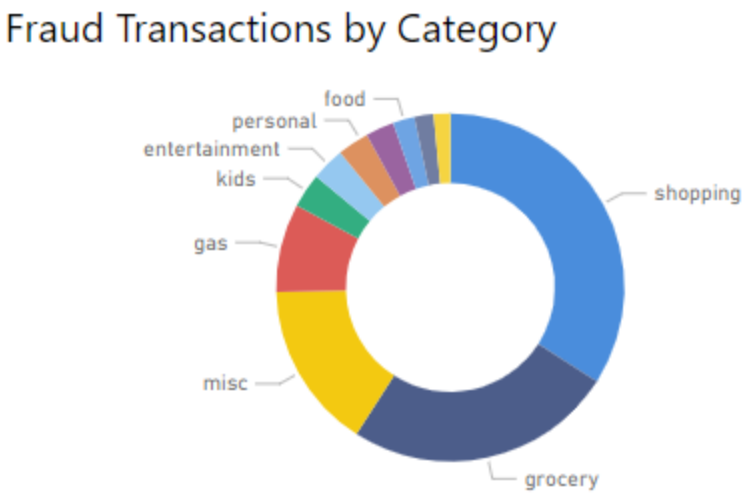
- **Pie Chart of Transaction Amount and Fraud**: This chart revealed that fraud is most prevalent in transactions ranging from $0 to $1,000. There was a considerable drop in fraud cases for amounts between $1,000 and $2,000, and no fraud was detected for transactions exceeding $2,000.



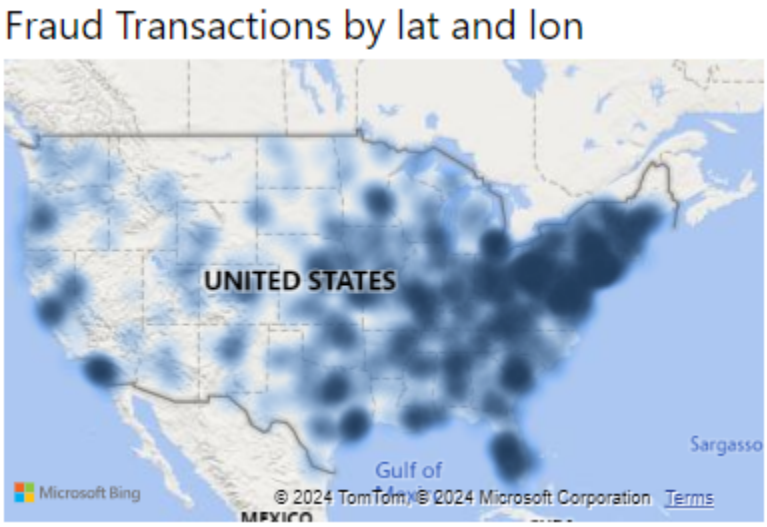
- **Bar Chart of Age and Fraud**: Here, a normal distribution emerged, showing that individuals between the ages of 30 and 60 were most likely to be involved in fraudulent transactions, with fewer cases outside this age range.



- **Donut Chart of Categories and Fraud**: The donut chart provided categorical insights, revealing that fraud is most common in the shopping sector, followed by grocery transactions.



- **Map Chart of Fraud by Location (Latitude and Longitude)**: This geographic representation of fraud indicated a higher concentration of fraudulent transactions on the east coast of the USA.

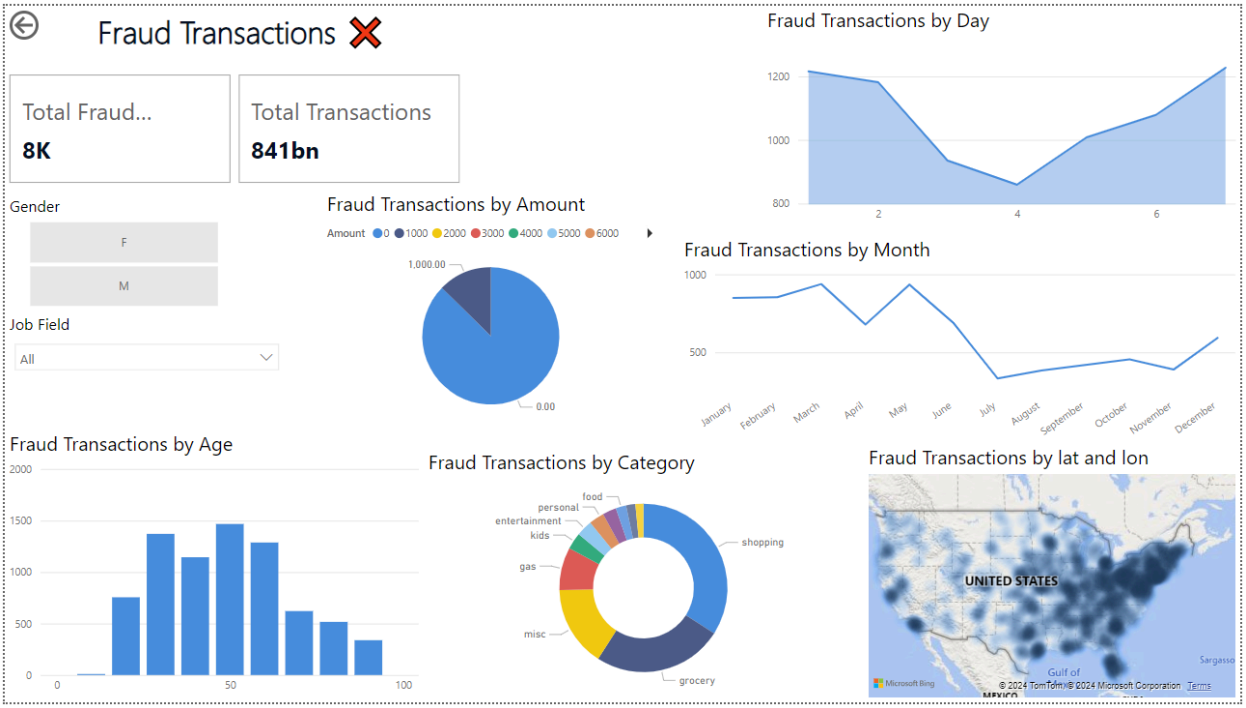


3. Dashboard Creation

A comprehensive dashboard was built to monitor and analyze these key metrics in real-time, offering stakeholders a dynamic way to track fraud trends:

- **Transaction Overview**: The dashboard begins with a high-level summary, displaying the total number of transactions alongside the number of fraudulent transactions. This provides immediate insight into the scale of fraud within the overall transaction volume.

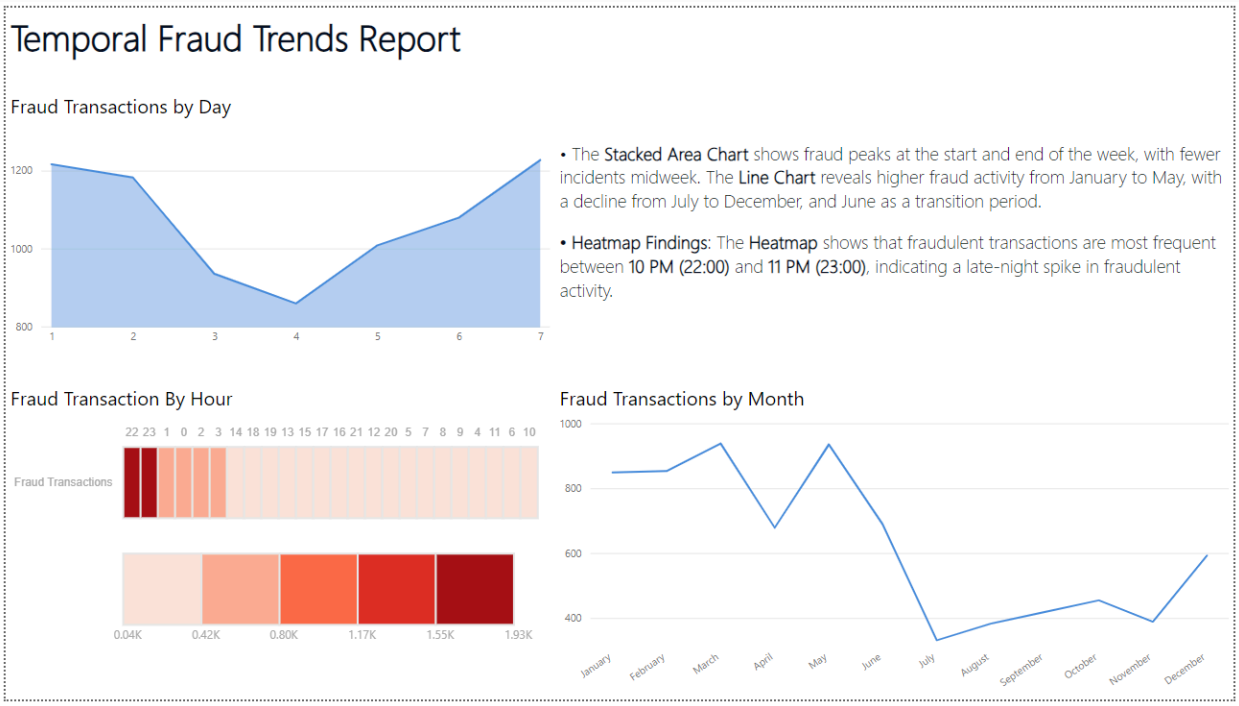
- **Slicing by Demographics**: The dashboard also includes slicing capabilities by gender and job field, allowing users to filter the data and identify demographic trends related to fraud occurrences.
- **Multi-dimensional Analysis**: Through interactive elements, users can drill down into specific time periods, transaction amounts, and categories, fostering a more granular understanding of fraud trends.



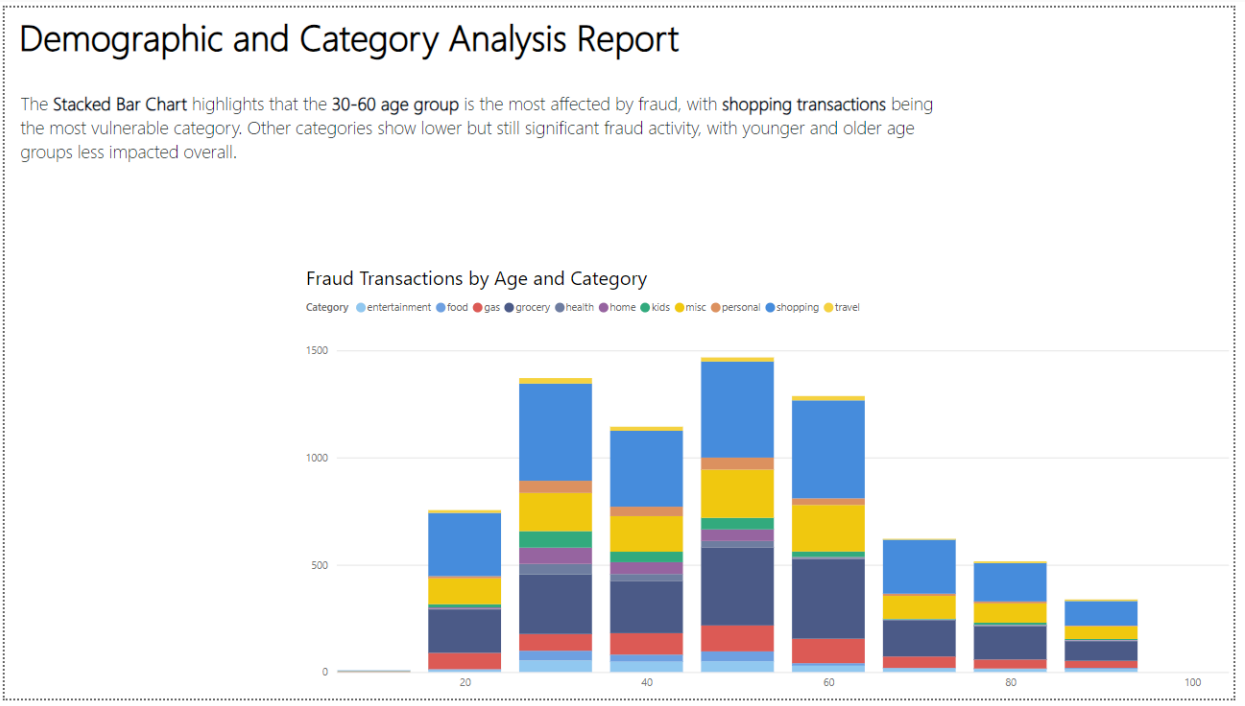
4. Reporting

The dashboard insights were further distilled into actionable reports, summarizing the key findings for presentation to stakeholders:

- **Temporal Fraud Trends**: Reports highlighted that fraudulent activities peak at the beginning and end of the week and are more prevalent in the first half of the year. A heatmap was added to show fraud occurrences by hour, revealing that the highest fraud rates occur at hours 22 and 23.



- **Demographic and Category Analysis**: The report now includes a stacked chart illustrating age and category, with the ability to drill down into specific categories. It pointed to the age group 30-60 as the most affected by fraud and identified shopping transactions as particularly vulnerable to fraudulent activity.



- **Geospatial Fraud Distribution**: The geospatial analysis was updated with a heatmap to show the states with the highest fraud occurrences. It was found that New York (NY), Texas (TX), and Pennsylvania (PA) have the highest concentrations of fraud.