Module:

BDM 1024 - Data Technology Solutions 02

Project Title:

Fraud Detection in the Banking Sector (Final Submission)

Intake:

Spring 2023 | Term One

Submitted By: **Group A**

Koshish Aryal

Punam Bhattarai

Saaz Neupane

Bipin Pandey

Shweta Laljibhai Thummar

Manan Tushar Kapadia

Robert Thapa

Dipti Baral

Introduction

This project aims to provide a reliable fraud detection system for the banking industry. Fraudulent transactions have the potential to cause large financial losses and harm a bank's standing. To identify and stop fraudulent activity in real time, an efficient data model in conjunction with data analytics is essential.

Problem Definition

A GlobalData survey conducted in September 2022 stated that 25% of Canadians had experienced fraud in the past three years. Among them, 35% were concerned about identity theft where as 26% worried fraudsters would steal their card details online. According to the new survey by the TD Bank Group, around 62% of Canadians feel more targeted now than ever by the financial fraud. Along with that according to TD, nearly 72% of Canadians said they were targeted by email/text message fraud, while 66% reported being targeted over the phone. In order to study the claim and further looking into the problem, we decided to looking into the banking data and possible fraud transactions that has been occurring.

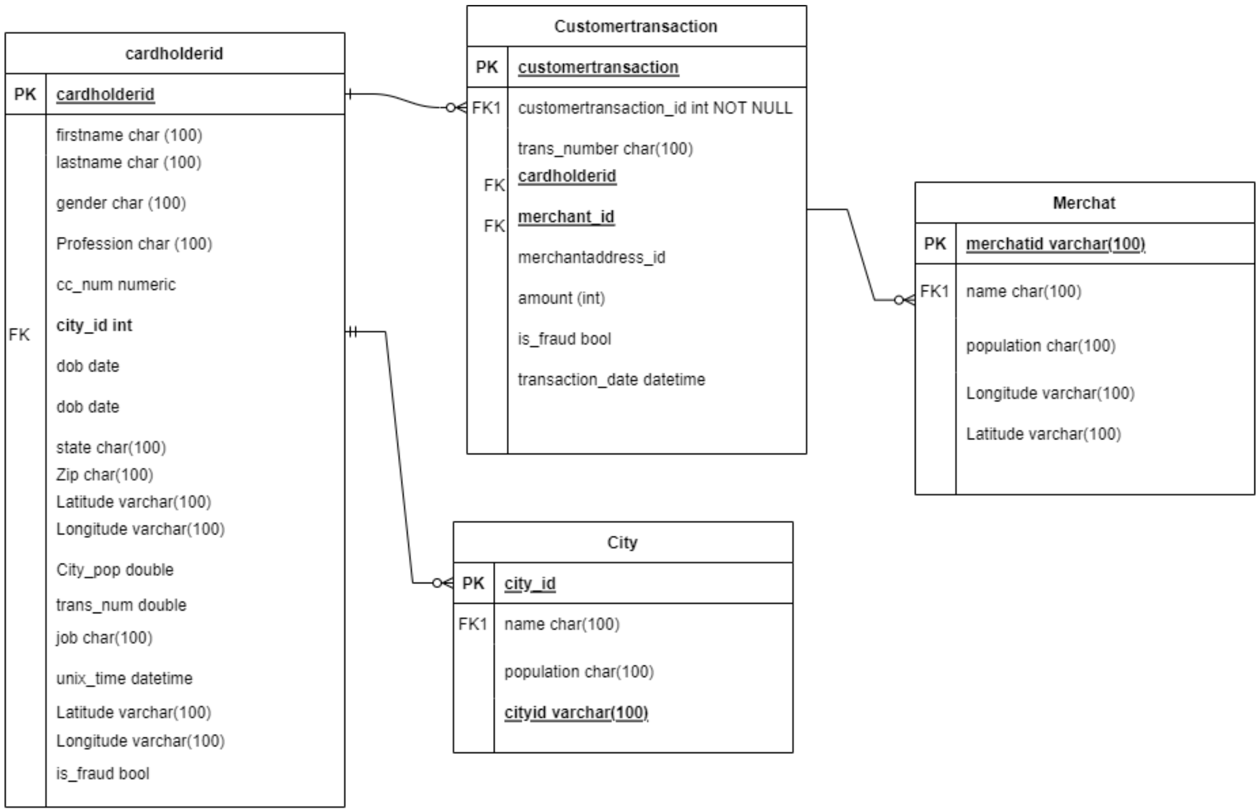
Data Design Pattern

The system analyzes a fraud dataset using two patterns: Role (Cardholder and Merchant) and Context (transaction details, cities, and merchant addresses). Role Pattern captures specific attributes for each role, while Context Pattern focuses on spatial and temporal aspects of transactions, helping detect anomalies based on locations, dates, and amounts for better fraud detection.

Data Model Design

The data model is designed to capture essential information about cardholders, merchants, and customer transactions.

Entity-Relationship Diagram (ERD):



Entity Descriptions:

- **Cardholder:** Represents banking customers, with attributes like CardholderID, FirstName, LastName, Gender, Profession, Cc_num, Dob, CityID, Street, State, Zip, Lat, and Long.
- **City:** Stores information about cities, including CityID, Name, and Population.
- **Merchant:** Holds data about merchants, with attributes like MerchantID, Name, and Category.
- **Customer Transaction:** Represents individual transactions made by cardholders. It includes CustomerTransactionID, Trans_num, CardholderID, MerchantID, Amount, Is_fraud, and Transaction_date.

SQL Queries and Insights

SQL (Structured Query Language) is a programming language used for managing and manipulating relational databases. It is widely used in the field of data management and is considered the standard language for interacting with databases. We utilized SQL queries to extract valuable insights from the dataset and analyze transaction patterns.

```sql
SELECT
  c.merchantid,
  COUNT(*) AS transaction_count,
  SUM(amount) AS total_amount
FROM
  customertransaction c join merchant m on c.merchantid = m.merchantid
GROUP BY
  c.merchantid
HAVING
  SUM(amount) > (SELECT AVG(amount) * 2 FROM customertransaction)
ORDER BY
  total_amount DESC;
```

Insight: This query calculates the total number of transactions and the total transaction amount for each merchant. It then filters the results to include only those merchants whose total transaction amount is greater than twice the average transaction amount across all transactions. The final result set is sorted in descending order based on the total transaction amount, displaying merchants with the highest total amounts first.

```sql
SELECT m.Category, AVG(ct.Amount) as AvgTransactionAmount
FROM CustomerTransaction ct
JOIN Merchant m ON ct.MerchantID = m.MerchantID
GROUP BY m.Category
ORDER BY AvgTransactionAmount DESC;
```

Insight: This query calculates the average transaction amount for each merchant category and presents the results in descending order based on the average transaction amount. The result set will show the merchant categories with the highest average transaction amounts first, providing insights into the most profitable or popular categories based on transaction data.

Query Query History

```sql
WITH suspicious_transactions AS (
  SELECT
    custtransactionid, amount, cardholderid, transaction_date,
    COUNT(*) OVER (PARTITION BY cardholderid) AS num_transactions,
    AVG(amount) OVER (PARTITION BY cardholderid) AS avg_transaction_amount,
    STDDEV(amount) OVER (PARTITION BY cardholderid) AS std_dev_transaction_amount,
    LAG(amount) OVER (PARTITION BY cardholderid ORDER BY transaction_date) AS prev_transaction_amount
  FROM
    customertransaction
),
potential_fraud AS (
  SELECT
    custtransactionid, amount, cardholderid, transaction_date, num_transactions, avg_transaction_amount,
    std_dev_transaction_amount, prev_transaction_amount,
    CASE
      WHEN amount > (avg_transaction_amount + (3 * std_dev_transaction_amount)) THEN 'High Amount'
      WHEN amount > (1.5 * prev_transaction_amount) THEN 'Significant Increase'
      WHEN num_transactions > 10 THEN 'High Frequency'
      ELSE 'None'
    END AS fraud_type
  FROM
    suspicious_transactions
)
SELECT *
FROM potential_fraud
WHERE fraud_type <> 'None';
```

Insight: This query uses CTEs and window functions to analyze transaction data, calculate transaction statistics per cardholder, and identify potential fraudulent transactions based on specific criteria. The result is a list of suspicious transactions with their respective fraud types.

Query Query History

```sql
SELECT
  custtransactionid,
  amount
FROM
  customertransaction
WHERE
  ABS(
      amount - (SELECT AVG(amount) FROM customertransaction))
      > 3 * (SELECT STDDEV(amount) FROM customertransaction);
```

Insight: This query identifies transactions that significantly deviate from the average transaction amount. It does so by comparing the absolute difference between each transaction amount and the average transaction amount to three times the standard deviation. The result set contains the customertransactionid and amount of the transactions that meet the specified condition. These transactions may be considered as outliers or potential anomalies in the dataset.

Query Query History

```sql
SELECT
  ch.cc_num,concat(ch.firstname,' ',ch.lastname) as name,
  COUNT(*) AS transaction_count,
  MIN(transaction_date) AS first_transaction_date,
  MAX(transaction_date) AS last_transaction_date
FROM
  customertransaction ct join cardholder ch on ct.cardholderid = ch.cardholderid
GROUP BY
  ch.cc_num,name
HAVING
  COUNT(*) > 100
  AND (EXTRACT(DAY FROM MAX(transaction_date)) - EXTRACT(DAY FROM MIN(transaction_date))) < 30;
```

Insight: This query identifies cardholders who have made more than 100 transactions within a 30day period. It retrieves their credit card numbers, full names, total transaction counts, first transaction dates, and last transaction dates. The result set contains the information of cardholders who meet the specified conditions. These cardholders may be considered as active users or potential targets for further analysis based on their transaction activity.

Query Query History

```sql
SELECT
    ch.Gender,
    COUNT(*) AS FraudulentTransactions
FROM
    CustomerTransaction ct
JOIN
    Cardholder ch ON ct.CardholderID = ch.CardholderID
WHERE
    ct.is_fraud = true
GROUP BY
    ch.Gender;
```

Insight: This query groups fraudulent transactions by gender of the cardholder. It can help identify if there are any gender-specific patterns or vulnerabilities related to fraud, aiding in fraud prevention strategies.

Query Query History

```sql
SELECT
    m.Category,
    AVG(ct.Amount) AS AvgTransactionAmount
FROM
    CustomerTransaction ct
JOIN
    Merchant m ON ct.MerchantID = m.MerchantID
GROUP BY
    m.Category
ORDER BY
    AvgTransactionAmount DESC;
```

Insight: This query calculates the average transaction amount for each merchant category. It helps identify which categories generate the highest transaction values, guiding marketing strategies and business focus.

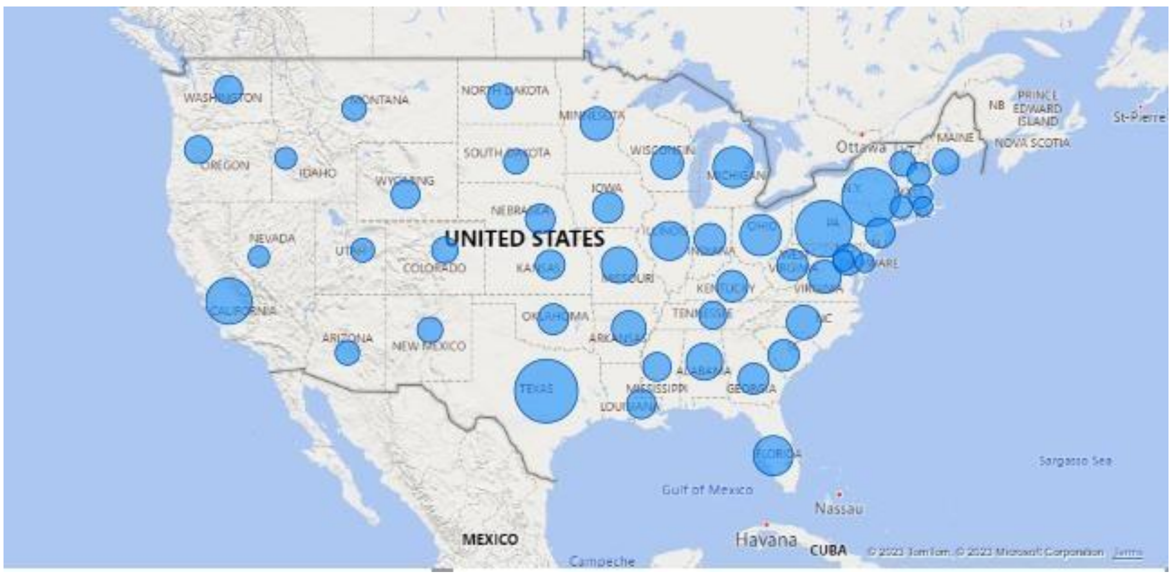
Query Query History

```sql
SELECT
    CASE
        WHEN DATE_PART('YEAR', NOW()) - DATE_PART('YEAR', ch.Dob) < 18 THEN 'Under 18'
        WHEN DATE_PART('YEAR', NOW()) - DATE_PART('YEAR', ch.Dob) BETWEEN 18 AND 30 THEN '18-30'
        WHEN DATE_PART('YEAR', NOW()) - DATE_PART('YEAR', ch.Dob) BETWEEN 31 AND 50 THEN '31-50'
        ELSE 'Over 50'
    END AS AgeGroup,
    COUNT(*) AS TotalTransactions
FROM
    CustomerTransaction ct
JOIN
    Cardholder ch ON ct.CardholderID = ch.CardholderID
GROUP BY
    AgeGroup
ORDER BY
    AgeGroup;
```

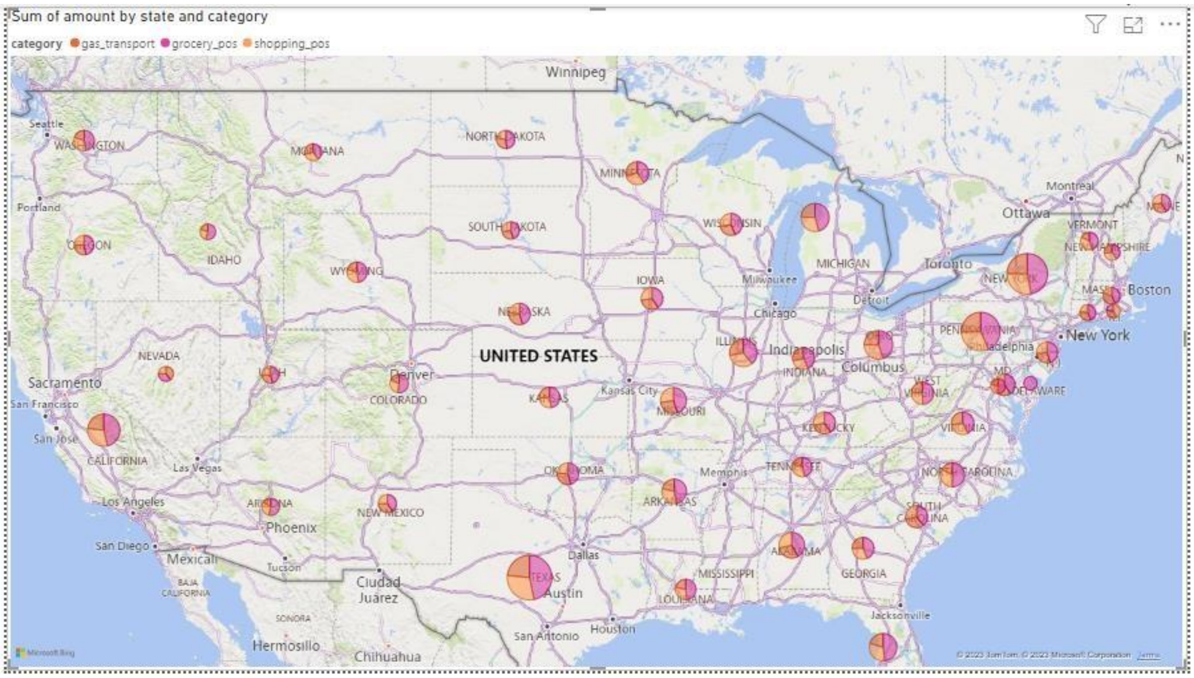
Insight: This query groups transactions into different age groups based on the cardholder's date of birth. It can reveal the transaction behaviors of different age segments, aiding in targeted marketing and tailored product offerings.

Data Visualization

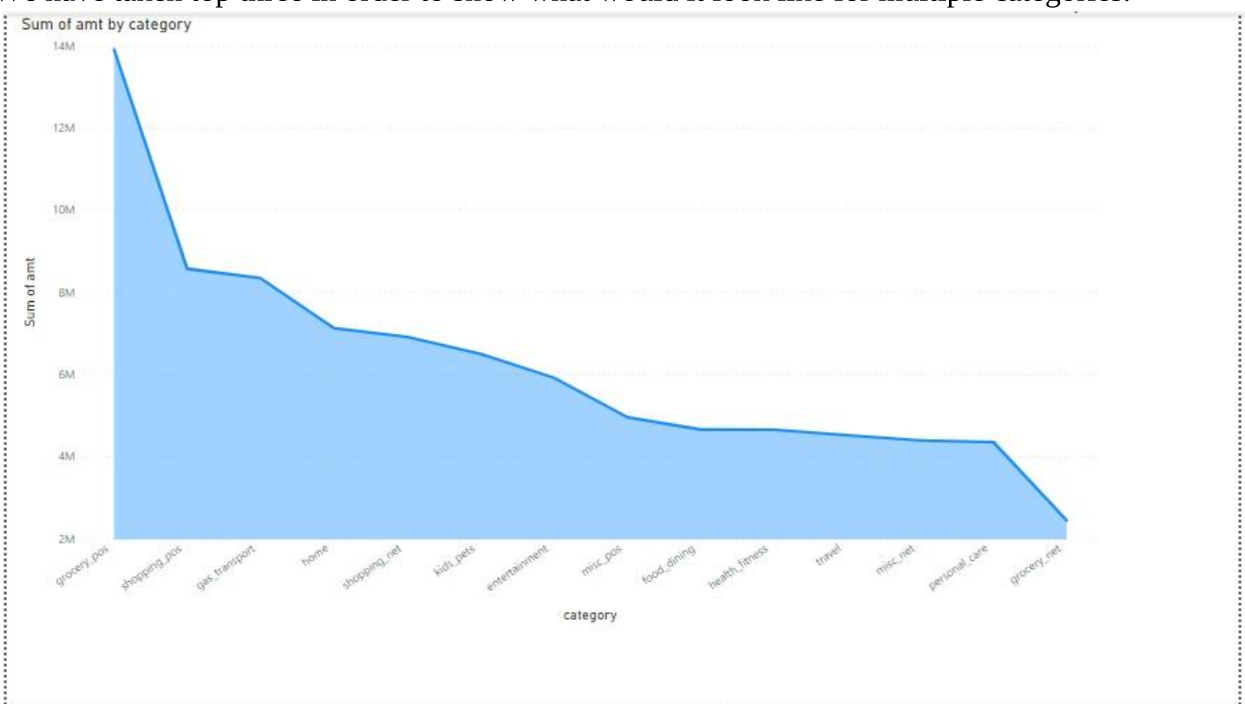
Data visualization is all about transforming data and information into visual representations that are easy for humans to understand. It includes creating visual stories using charts, graphs, maps, or infographics by using colors, shapes, and patterns, data visualization simplifies complex data sets, making them more accessible and digestible for people. It helps us to see patterns, trends, and insights that might otherwise be hidden in rows and columns of numbers.



Here, the Sum of the transaction amount that has happen is in the different states is calculate and show in the figure. Here, size of the bubble gets bigger and bigger with the increase in the amount of transaction that has been occurred. By showing this, we have tried to show the regional trend where the most transaction could occur.

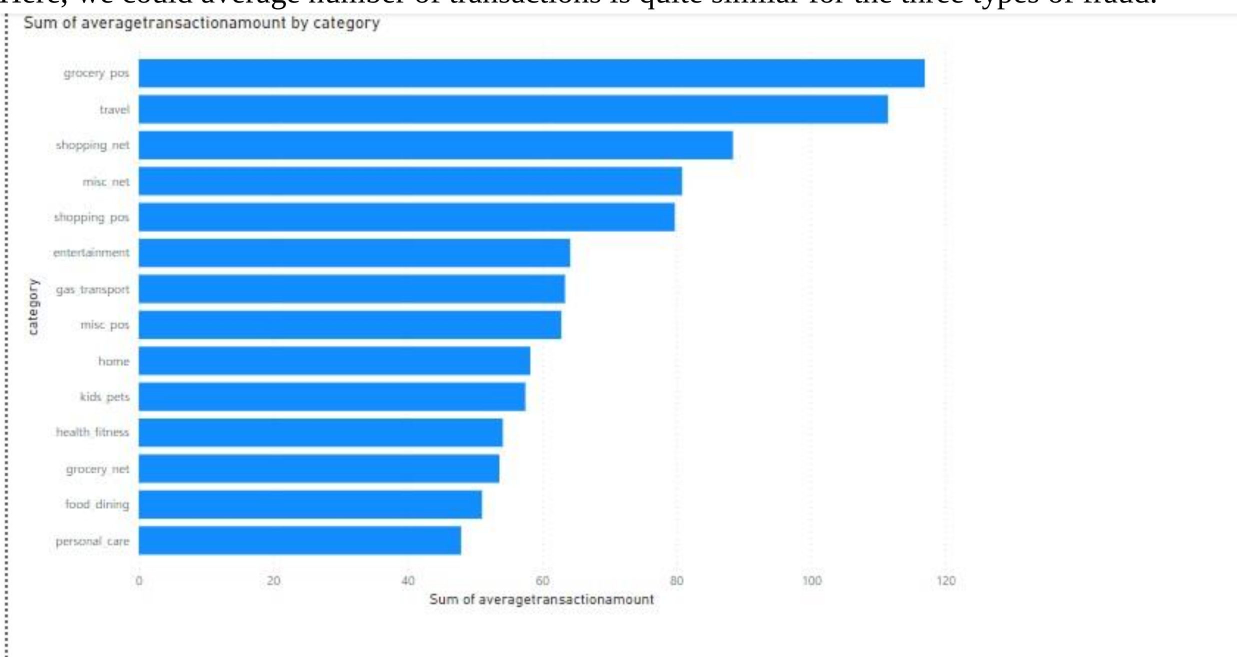


Here again we have used the total sum based on top categories where the fraud could take place. We have taken top three in order to show what would it look like for multiple categories.

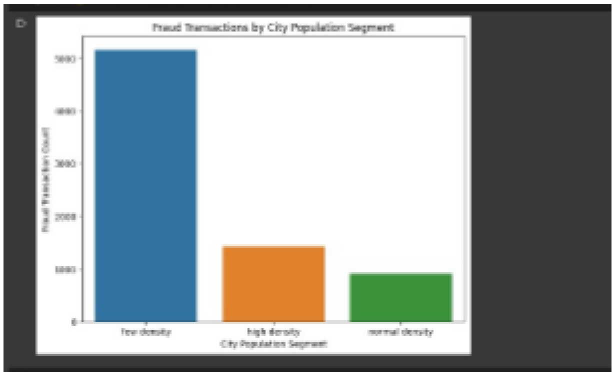


In this graph we tend to show the trend of the possible fraud transaction of that could occur in different categories. Here, we could see the in groceries highest chances of occurring fraud transaction.

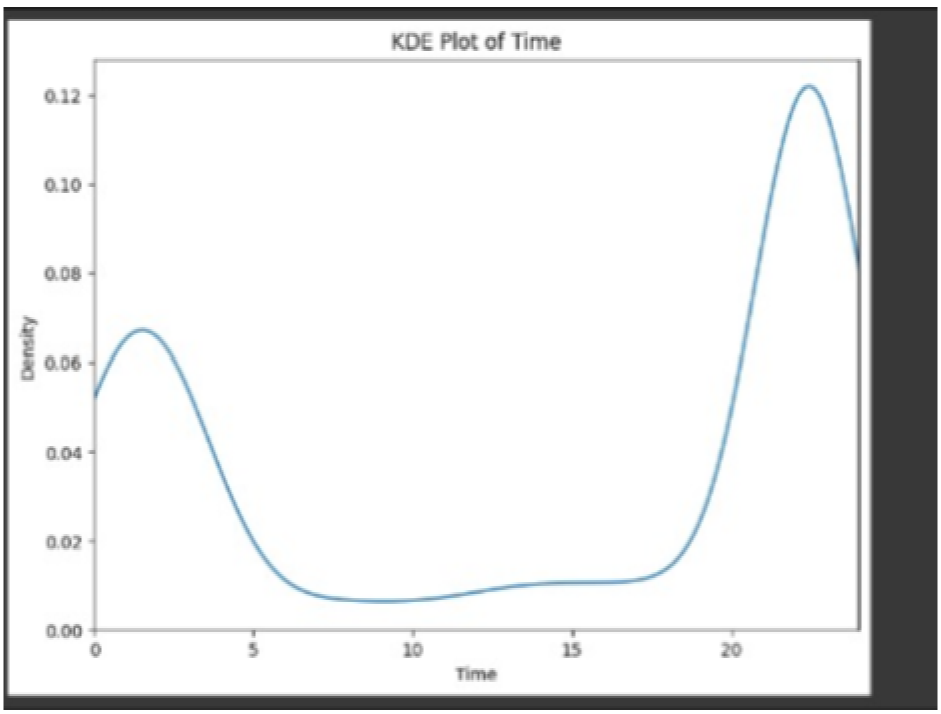
It is used to represent data in a visually appealing and concise manner. Here, the donut chart is used to visualize the average number of transactions that has occurred in the specific fraud type. Here, we could average number of transactions is quite similar for the three types of fraud.



Bar graph is used to visually display and compare the values of different categories or groups. Here, the graph shows that average amount of transaction that has been occurred in different categories.



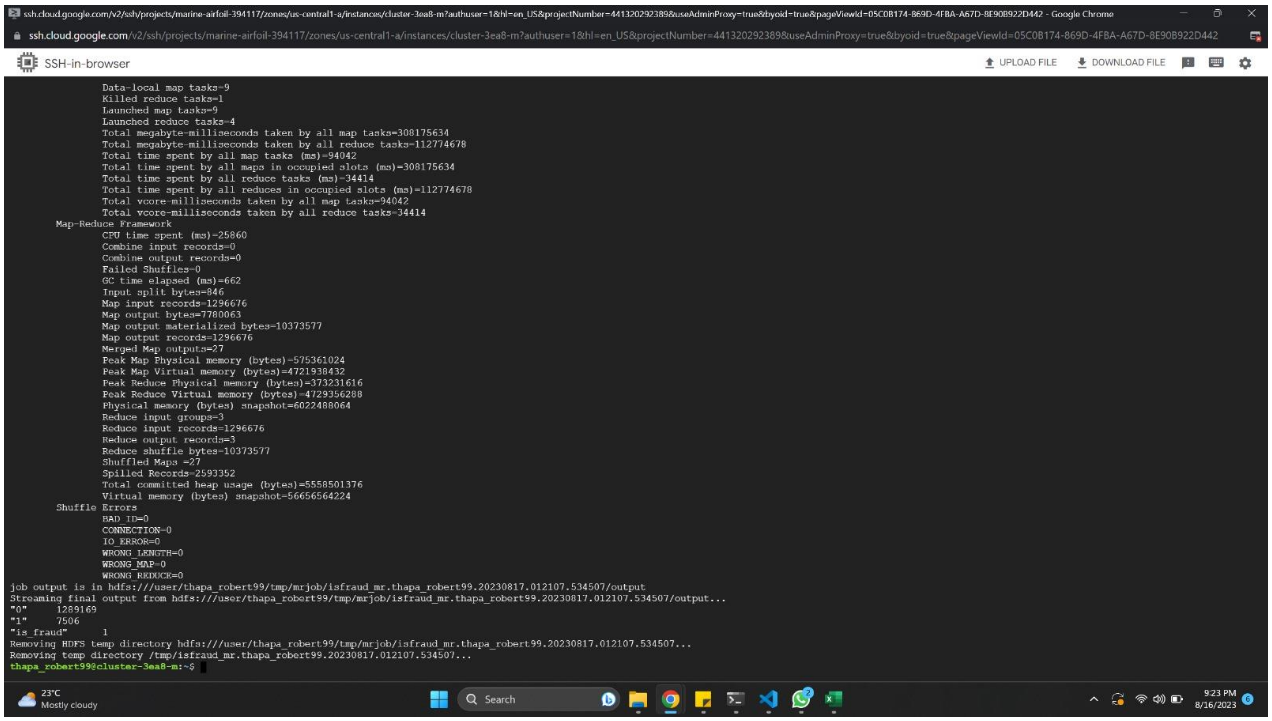
Here, fraud transaction based on the population density is represented using seaborn and matplotlib libraries.



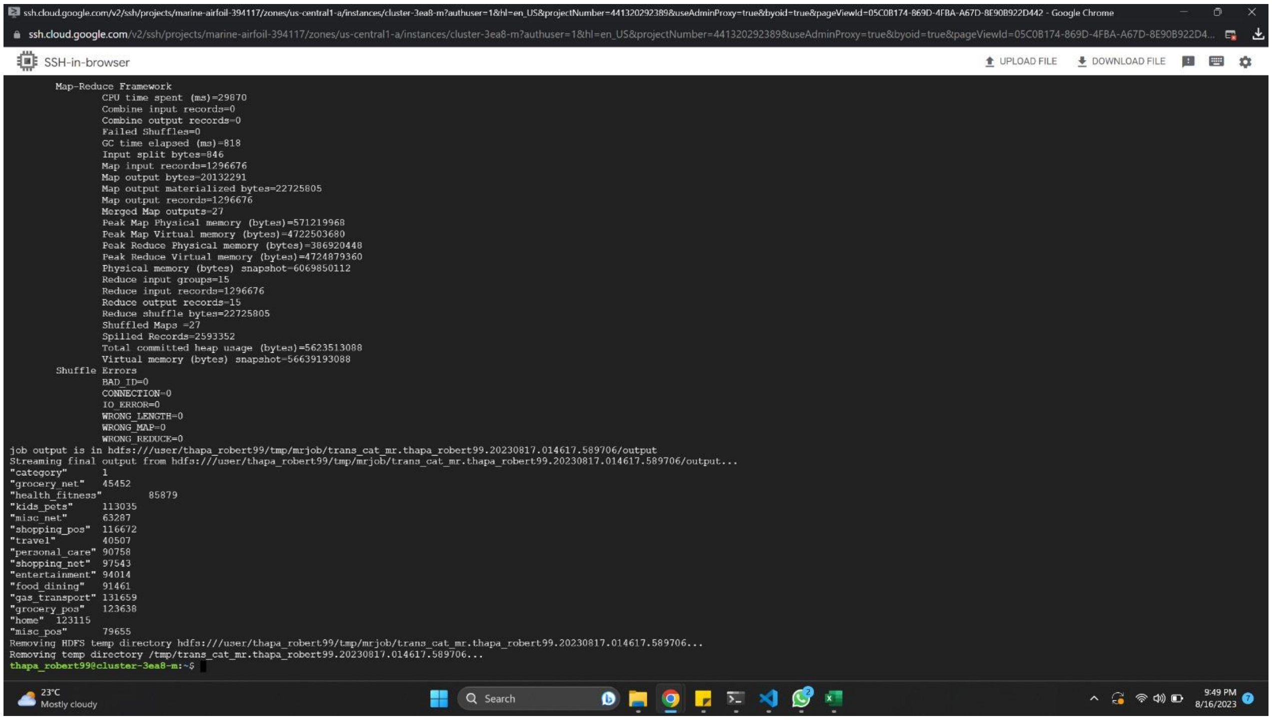
Here, KED plot of time is used in order to visualize the hour of days to find the possible fraud transactions.

Hadoop

As data grow more and more sql takes more time to complete the queries. We need new way to make things easier and Hadoop comes in play here. Hadoop is a tool that helps process and store large amounts of data across multiple computers. It is great for handling big data because it splits the work among different machines. Hadoop has a special file system called HDFS that stores data across many computers, so even if one computer fails, the data is still safe. It also has a processing framework called MapReduce that breaks down tasks into smaller parts and runs them in parallel on different computers. This makes data processing faster and more efficient. Here, we have used to hadoop cluster to analyze the data in order to process and run mapreduce job for it.



Here, Hadoop cluster is introduced in order to find out the performed transaction is fraud or not using mapreduce in Hadoop cluster.



Here, Hadoop cluster is run in order to differentiate the categories using mapreduce. The results is presented in ascending order based on transaction count.

PySpark Jobs

PySpark is a Python library that provides an interface for programming Apache Spark, a fast and distributed data processing engine. It integrates well with popular Python libraries such as NumPy, Pandas, and scikit-learn. As it has distributed computing capabilities.

pySparkJob.py > ...

```python
import pyspark
import sys
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, current_date, datediff, when

#SparkContext
sc = pyspark.SparkContext()
sqlContext = pyspark.sql.SQLContext(sc)

inputdir = sys.argv[1]

# Load the data into a DataFrame
df = sqlContext.read.format('com.databricks.spark.csv').options(header='true', inferschema='true').load(inputdir+'transaction.csv')

# Step 3: Calculate the age of each cardholder and categorize them into age groups
df = df.withColumn("dob", df["dob"].cast("date"))
df = df.withColumn("age", datediff(current_date(), df["dob"]) / 365)
df = df.withColumn(
    "AgeGroup",
    when(col("age") < 18, "Under 18")
    .when((col("age") >= 18) & (col("age") <= 30), "18-30")
    .when((col("age") >= 31) & (col("age") <= 50), "31-50")
    .otherwise("Over 50")
)

# Step 4: Group by AgeGroup and calculate total transactions
result_df = df.groupBy("AgeGroup").agg({"age": "count"}).withColumnRenamed("count(age)", "TotalTransactions")

# Display the final result
print("AgeGroup with total transactions : ")
result_df.show()
```

```
+--------+-----------------+
|AgeGroup|TotalTransactions|
+--------+-----------------+
|   31-50|           543268|
| Over 50|           601171|
|   18-30|           152236|
+--------+-----------------+
```

This job categorizes customers into age groups, calculates total transactions per group, and displays the analysis results - demonstrating common PySpark DataFrame operations like casting, date functions, conditional logic, grouping, aggregating and renaming columns.

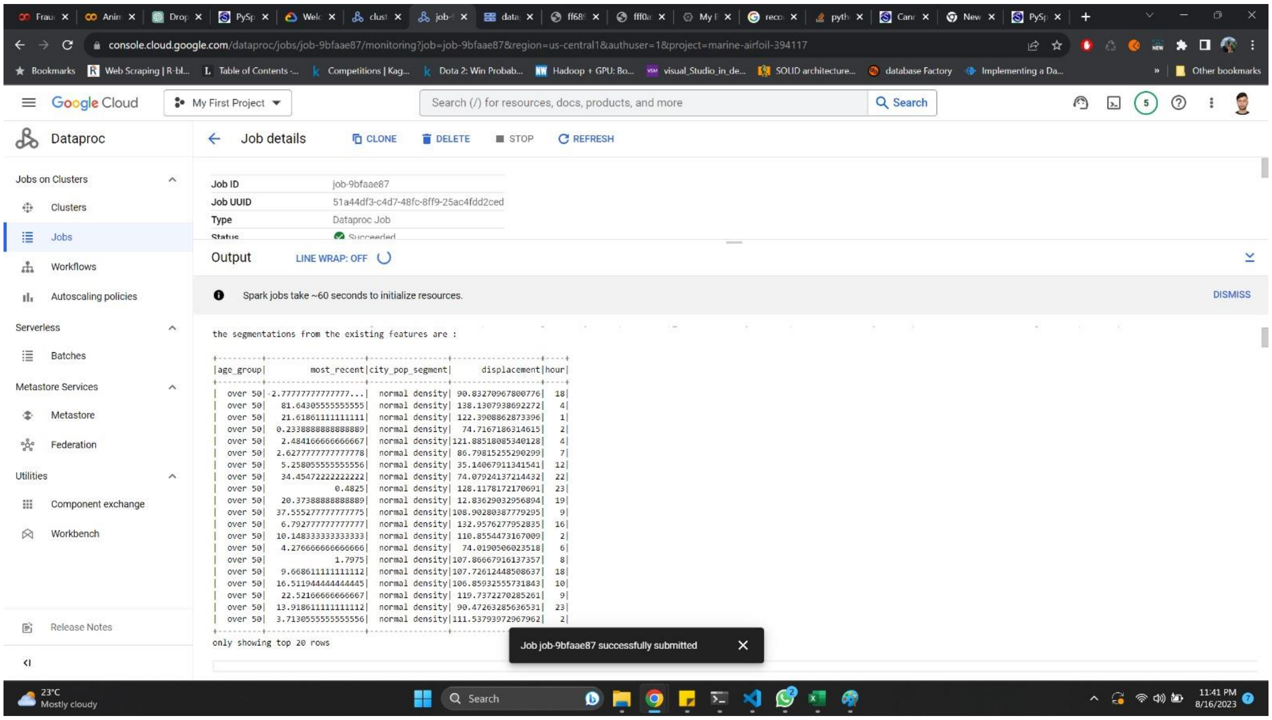
pySparkJob3.py > ...

```python
import pyspark
import sys
from pyspark.sql import SparkSession
from pyspark.sql.window import Window
from pyspark.sql import functions as F
from pyspark.sql.functions import col, lag, udf
from pyspark.sql.types import StringType

sc = pyspark.SparkContext()
sqlContext = pyspark.sql.SQLContext(sc)

inputdir = sys.argv[1]

df = sqlContext.read.format('com.databricks.spark.csv').options(header='true', inferschema='true').load(inputdir+'transaction.csv')

windowSpec = Window.partitionBy("cc_num").orderBy("trans_date_trans_time")

df = df.withColumn("num_transactions", F.count("trans_num").over(windowSpec))

df = df.withColumn("avg_transaction_amount", F.avg("amt").over(windowSpec))

df = df.withColumn("std_dev_transaction_amount", F.stddev("amt").over(windowSpec))

df = df.withColumn("prev_transaction_amount", F.lag("amt", 1).over(windowSpec))
```

```python
def detect_fraud(amt, avg_transaction_amount, std_dev_transaction_amount, prev_transaction_amount, num_transactions):
    if amt is None or avg_transaction_amount is None or std_dev_transaction_amount is None or prev_transaction_amount is None:
        return 'None'

    if amt > (avg_transaction_amount + (3 * std_dev_transaction_amount)):
        return 'High Amount'
    elif amt > (1.5 * prev_transaction_amount):
        return 'Significant Increase'
    elif num_transactions > 10:
        return 'High Frequency'
    else:
        return 'None'

detect_fraud_udf = udf(detect_fraud, StringType())
df = df.withColumn("fraud_type", detect_fraud_udf("amt", "avg_transaction_amount", "std_dev_transaction_amount", "prev_transaction_amount", "num_tr

potential_fraud_df = df.filter(df['fraud_type'] != 'None')

selected_columns = ['trans_num', 'amt', 'cc_num', 'trans_date_trans_time', 'fraud_type']
selected_potential_fraud_df = potential_fraud_df.select(selected_columns)

selected_potential_fraud_df.show()
```

trans_num	amt	cc_num	trans_date_trans_time	fraud_type
12554920d4920895a...	52.67	501828204849	2019-01-02 12:48:10	Significant Increase
4a52d2e61cb687153...	31.79	501828204849	2019-01-15 17:10:52	Significant Increase
e3d88c2eedd635a49...	61.56	501828204849	2019-01-22 11:38:05	Significant Increase
e75b1bb8e5ba6823e...	50.91	501828204849	2019-01-22 18:35:13	High Frequency
1ed15a8619b0ceba8...	64.03	501828204849	2019-01-23 05:43:39	High Frequency
367bc6d8b47753138...	118.58	501828204849	2019-01-23 17:18:28	Significant Increase

This PySpark code detects fraudulent transactions by calculating statistics like average, standard deviation and previous amount over each card using window functions. It then flags transactions based on rules like high amount, significant increase, or high frequency by applying a UDF. Finally, it filters and displays transactions with potential fraud.



Feature engineering is the process of transforming raw data into a format that is suitable for machine learning algorithms. It involves creating new features or modifying existing ones to improve the performance and effectiveness of machine learning models.

Here, we have implemented feature engineering and analysis in spark which give error on which hour of time and which age group is more targeted. The segmentation is done based on time where we also calculated displacement between customer and merchant.

Conclusion

This project successfully developed a data model leveraging design patterns and integrated data analytics decision trees to build an efficient fraud detection system for the banking sector. The SQL queries provided valuable insights into transaction patterns. The implementation of this fraud detection system can significantly reduce financial losses due to fraudulent activities, while enhancing customer trust and security within the banking domain.

Work division

Group Member	Task Performed

Robert Thapa	- Develop Hadoop cluster and run mapreduce job - Develop database design and implementation - Feature engineering and segmentation in pyspark - Review the visualization performed by **Koshish**
Koshish Aryal	- Create visualizations using BI tool along with **Punam** - Design conceptual and logical data model diagrams - Review and help **Robert** feature engineering and segmentation
Saaz neupane	- Import and preprocess dataset in Python - Write 2 complex SQL queries to analyze data - Help and review **Shweta** to visualization and run Spark Jobs
Shweta Laljibhai Thummar	- Statistical analysis of data in SQL - Create visualizations using Python matplotlib and seaborn - Review and help **Saaz** for SQL queries and preprocessing
Punam Bhattarai	- Create Power BI reports and visualizations along with **Koshish** - Research on pyspark jobs - Review and help **Dipti** to prepare reports
Dipti Baral	- Statistical analysis of data in SQL along with **Shweta** - Create visualizations using the on seaborn and matplotlib after spark job is run - Review the task performed by **Punam** - Prepare comprehensive project report and slides
Bipin Pandey	- Create presentation highlighting project architecture and key insights

	- Develop Spark job for business insights - Review the final reports and help **Manan** with spark jobs
Manan Tushar Kapadia	- Verify the final presentations - Develop Spark job for business insights along with **Bipin** - Review the final reports done by **Dipti**