Big Data Analytics

Spark, Hadoop Map-Reduce, and more

Saman Teymouri

February 2025

CONTENTS

ABSTRACT

This study uses big data analysis to analyze fraud detection in banking transactions. First, the importance of fraud detection and the role of big data analytics in that area is discussed. The dataset used for this study contains more than 7 million records and is well-designed to reflect fraudulent and genuine transactions. Afterward, this study uses Map Reduce and Spark models to prepare the dataset for analysis. It shows that Spark is much more efficient than the Hadoop Map Reduce programming model. In the next step, this paper applies a classification model to extract insight from the dataset and predict future fraud. The results show 93 percent accuracy in predictions. Moreover, visualizations help to explore data and gain useful information about it. Finally, feature selection using correlation shows that building a model based on a few features also gives the same result with lower resource and time consumption.

1. INTRODUCTION

Nowadays, analyzing data and extracting insights from it has become one of the most important parts of every business. Along with the growth of businesses and enlarging the amount of data, it is necessary to have appropriate tools to analyze vast datasets. Big data analytics is about gathering, processing, and studying large amounts of data to gain knowledge about a specific business or problem (Shabbir and Gardezi, 2020).

There are sort of tools to address big data analytics problems. Apache Hadoop and Apache Spark are two of the most reputable and robust tools that can be used in this field. Apache Hadoop is well-known for its distributed storage and Map Reduce programming model while Apache Spark offers faster in-memory analysis using its RDD (Resilient Distributed Dataset) (Zaharia et al., 2012). Spark is a more advanced tool that provides specific modules to work with datasets same as a table via SQL syntax which is simpler and more efficient than the Map Reduce model.

The other important ability of Spark is its MLlib and ML libraries which are responsible for machine learning operations. There are different modules that are suitable for regression, classification, or clustering as well as deep learning. These abilities make Spark one of the most promising tools in the world of big data analytics.

When we are dealing with big data analytics, we also need to think about storage capacities and computation capabilities. Cloud-based platforms are very useful and practical solutions for this matter. AWS (Amazon Web Services), GCP (Google Cloud Platform), and Microsoft Azure are the most promising platforms to handle big data.

2. Problem Definition and Business Context

2.1. Fraud in Banking System

There are several forms of fraud including Healthcare and medical fraud, financial fraud, real-state fraud, contractual fraud, etc. After medical fraud which deals with human lives, financial fraud, specifically banking transaction fraud, is a very crucial matter since it influences the economic system and personal financial security. The Association of Certified Fraud Examiners (ACFE, 2022) reported that the average organization loses about 5% of its annual revenue to fraud (Median loss of about $117,000). Fraud in banking transactions can cause not only financial loss but also trust loss and social problems. Before the prominence of online shopping, some traditional approaches might be enough to stop fraud. But after that, the number of transactions also increased drastically so the urge raised to predict and prevent fraudulent activities before happening. Moreover, cryptocurrency emergence makes tracking transactions much more difficult than before. As a result, nowadays fraud detection in banking transactions should be managed automatically, predictive, and effectively.

2.2. Big Data in Fraud Detection

Banks process millions of transactions every day which makes it impossible to check them manually or with traditional methods. It reveals the need for more advanced technologies such as big data analytics to analyze large amounts of data in order to find fraudulent transactions. Conventional systems for example rule-based systems are not able to work efficiently since fraud techniques change rapidly and can bypass static rules. On the other hand, data-driven approaches extract patterns and update them regularly which can be very effective in identifying fraudulent behaviors (Ngai et al., 2011).

Big data refers to data with large volume, complex structure that can be generated with high velocity. Banking institutions create different types of data such as transaction records and customer profiles which can meet all requirements to be considers as big data. Big data analytics enables banking system to enhance accuracy, real-time monitoring, and predictive detection. There are different machine learning models, specifically binary classification models, such as logistic regression, support vector machine, and random forest to analyze previous labeled data and build a predictive model to detect future fraudulent transactions (Sulaiman et al., 2019).

Integration of fraud detection with big data analytics is useful for the banking system in different ways (Udeh et al., 2024):

1. **Better Customer Experience:** Analyzing large amounts of data and building a suitable model based on it provides higher accuracy and precision for both classes. It means more true positives (fraud correctly detected) and fewer false positives (genuine mistakenly distinguished as fraud). The first one helps customers to have secure accounts and the second one improves their experience by not bothering them with excessive sensitivity (Ahmadi, 2024).
2. **Cost Effectiveness:** Using big data analytics helps banks reduce manual efforts and lowers the costs. Additionally, early fraud detection prevents legal claims (Ahmadi, 2024).
3. **Compliance:** Big data analytics helps banking institutions with regulations such as Anti Money Laundry (AML) via real-time monitoring to detect anomalies.
4. **Scalability:** Big data analytics maintains stable performance in presence of high transaction amount. It is crucial for fraud detection to be scalable and remain effective even if the number of transactions get higher or new fraud strategies emerge.

There are also some challenges in implementing big data analytics for fraud detection in bank transactions:

1. **Data Privacy and Security**: Dealing with sensitive customer data needs strict regulations to prevent data breaches. It consumes a significant amount of cost and effort to create these infrastructures. Other issue is data privacy regulations, such as the General Data Protection Regulation (GDPR) in the European Union and the California Consumer Privacy Act (CCPA) in the United States that force organizations to obey specific rules regarding the collection, processing, and storage of personal data (Udeh et al., 2024).
2. **Integration Complexity**: Integration of the current traditional system with a new big data analytics system can be costly and complex.
3. **Skill Shortages**: According to the skill set of current employees, banks need to employ new staff to manage big data systems.

To conclude, it is obvious that implementing big data analytics in banking systems to prevent fraud in transactions can be a very challenging task but it is worth it to protect customers' assets and maintain trust. It brings more accuracy for fraud detection, real-time monitoring, and cost-efficiency in the long term.

2.3. Dataset for Fraud Detection using Big Data Analytics

This study uses a dataset that contains a combination of fraudulent and genuine transactions (Samadov, 2024). It has 24 columns and 7,477,306 rows which can easily mark it as a big dataset.

The schema of data (Figure 1) and a few rows of it (Figure 2) are shown below.

```
1 The schema of data before preparation is
root
 |-- transaction_id: string (nullable = true)
 |-- customer_id: string (nullable = true)
 |-- card_number: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- merchant_category: string (nullable = true)
 |-- merchant_type: string (nullable = true)
 |-- merchant: string (nullable = true)
 |-- amount: double (nullable = true)
 |-- currency: string (nullable = true)
 |-- country: string (nullable = true)
 |-- city: string (nullable = true)
 |-- city_size: string (nullable = true)
 |-- card_type: string (nullable = true)
 |-- card_present: boolean (nullable = true)
 |-- device: string (nullable = true)
 |-- channel: string (nullable = true)
 |-- device_fingerprint: string (nullable = true)
 |-- ip_address: string (nullable = true)
 |-- distance_from_home: integer (nullable = true)
 |-- high_risk_merchant: boolean (nullable = true)
 |-- transaction_hour: integer (nullable = true)
 |-- weekend_transaction: boolean (nullable = true)
 |-- velocity_last_hour: string (nullable = true)
 |-- is_fraud: boolean (nullable = true)
```

Figure 1: Dataset Schema

```
2 all columns for 10 first rows ==> record count: 10
+--------------+-----------+----------------+--------------------------+-----------------+-------------+--------------+---------+--------+---------+------------+---------+---------------+
|transaction_id|customer_id|card_number     |timestamp                 |merchant_category|merchant_type|merchant      |amount   |currency|country  |city        |city_size|card_type      |
+--------------+-----------+----------------+--------------------------+-----------------+-------------+--------------+---------+--------+---------+------------+---------+---------------+
|TX_a0ad2a2a   |CUST_72886 |6646734767813109|2024-09-30 02:00:01.03482 |Restaurant       |fast_food    |Taco Bell     |294.87   |GBP     |UK       |Unknown City|medium   |Platinum Credit|
|TX_3599c101   |CUST_70474 |376800864692727 |2024-09-30 02:00:01.764464|Entertainment    |gaming       |Steam         |3368.97  |BRL     |Brazil   |Unknown City|medium   |Platinum Credit|
|TX_a9461c6d   |CUST_10715 |5251909460951913|2024-09-30 02:00:02.273762|Grocery          |physical     |Whole Foods   |102582.38|JPY     |Japan    |Unknown City|medium   |Platinum Credit|
|TX_7be21fc4   |CUST_16193 |376079286931183 |2024-09-30 02:00:02.297466|Gas              |major        |Exxon         |630.6    |AUD     |Australia|Unknown City|medium   |Premium Debit  |
|TX_150f490b   |CUST_87572 |6172948052178810|2024-09-30 02:00:02.544063|Healthcare       |medical      |Medical Center|724949.27|NGN     |Nigeria  |Unknown City|medium   |Basic Debit    |
|TX_7fb62ea6   |CUST_55630 |6771346275824473|2024-09-30 02:00:03.050458|Education        |online       |Coursera      |11.76    |BRL     |Brazil   |Unknown City|medium   |Platinum Credit|
|TX_e0d7eb37   |CUST_89147 |371305533183152 |2024-09-30 02:00:03.14944 |Grocery          |online       |Instacart     |2606.19  |BRL     |Brazil   |Unknown City|medium   |Platinum Credit|
|TX_eb55c2be   |CUST_10150 |5927166525373625|2024-09-30 02:00:06.295911|Travel           |hotels       |Westin        |828.33   |EUR     |Germany  |Unknown City|medium   |Platinum Credit|
|TX_eb63010a   |CUST_83143 |5029335992770061|2024-09-30 02:00:06.813965|Healthcare       |medical      |Medical Center|104921.0 |NGN     |Nigeria  |Unknown City|medium   |Premium Debit  |
|TX_b0cd71e2   |CUST_35022 |4412911822339760|2024-09-30 02:00:07.105592|Retail           |online       |eBay          |51521.84 |MXN     |Mexico   |Unknown City|medium   |Premium Debit  |
+--------------+-----------+----------------+--------------------------+-----------------+-------------+--------------+---------+--------+---------+------------+---------+---------------+

+------------+-----------+-------+--------------------------------+---------------+------------------+------------------+---------------+-------------------+------------------+--------+
|card_present|device     |channel|device_fingerprint              |ip_address     |distance_from_home|high_risk_merchant|transaction_hour|weekend_transaction|velocity_last_hour|is_fraud|
+------------+-----------+-------+--------------------------------+---------------+------------------+------------------+---------------+-------------------+------------------+--------+
|false       |iOS App    |mobile |e8e6160445c935fd0001501e4cbac8bc|197.153.60.199 |0                 |false             |0              |false              |{'num_transaction |false   |
|false       |Edge       |web    |a73043a57091e775af37f252b3a32af9|208.123.221.203|1                 |true              |0              |false              |{'num_transaction |true    |
|false       |Firefox    |web    |218864e94ceaa41577d216b149722261|10.194.159.204 |0                 |false             |0              |false              |{'num_transaction |false   |
|false       |iOS App    |mobile |70423fa3a1e74d01203cf93b51b9631d|17.230.177.225 |0                 |false             |0              |false              |{'num_transaction |false   |
|false       |Chrome     |web    |9880776c7b6038f2af86bd4e18a1b1a4|136.241.219.151|1                 |false             |0              |false              |{'num_transaction |true    |
|false       |Chrome     |web    |f79b73f19703d83fbfc2736f24cb108c|184.56.130.84  |1                 |false             |0              |false              |{'num_transaction |true    |
|false       |Android App|mobile |20464622be96fda2c75ee7d3698c004d|163.189.239.227|0                 |false             |0              |false              |{'num_transaction |false   |
|false       |Edge       |web    |170a1d12ba71772366076fd302fe29ca|122.115.119.11 |0                 |true              |0              |false              |{'num_transaction |false   |
|false       |Edge       |web    |44a2dde7b410ab3f71344022d1940d4b|103.121.115.91 |0                 |false             |0              |false              |{'num_transaction |false   |
|true        |NFC Payment|pos    |fbff6151bf7ab2d768a4646aad2cc5b2|3.35.80.156    |1                 |false             |0              |false              |{'num_transaction |true    |
+------------+-----------+-------+--------------------------------+---------------+------------------+------------------+---------------+-------------------+------------------+--------+
```

Figure 2: Dataset First 10 Rows

The dataset has 15 string, 4 boolean, 4 numeric, and 1 timestamp columns. It contains 4,869 unique customers who committed transaction through different merchants. There are some other information about dataset which are stated later.

3. Cloud Environment Setup and Data Ingestion

3.1. Cloud Platform Selection

There are different cloud platforms for big data analytics such as Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure. The first option is AWS EMR which is used for some analysis. Regardless of Amazon's claim based on the existence of a free tier, it is not really free and starts charging even in very early usage. The second option is GCP Dataproc which offers a 90-day free trial with 300 euros credit that is very helpful for students although the credit consumption model is not very clear. It charges the user for significant computational costs even if the cluster is stopped.

3.2. Setup Google Cloud Platform

After activating free trial google cloud account, the first step is to create a new project (Figure 3).

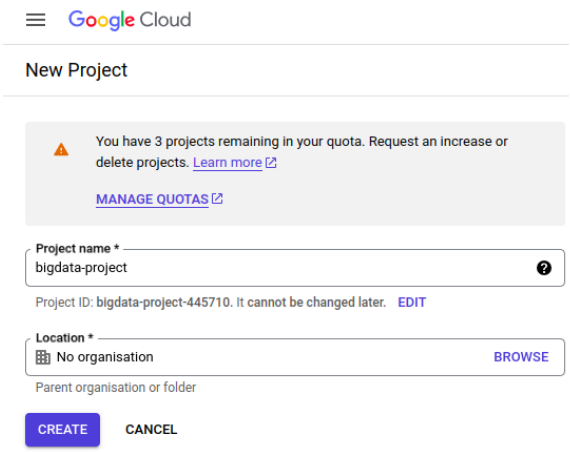

Figure 3: Create a New Project

The second step is linking the project to the billing account (Figure 4).

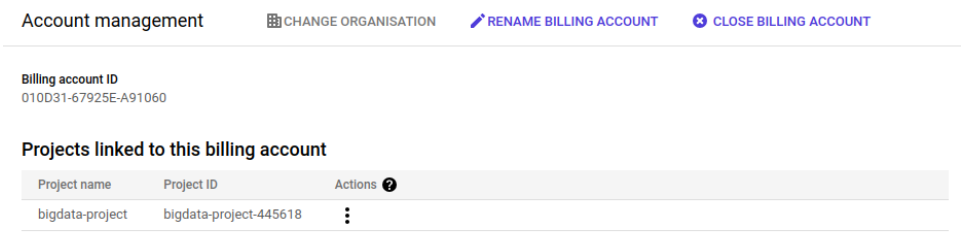

Figure 4: Link The Project to The Billing Account

The next step is to create a bucket. Buckets are fundamental storage units in Google Cloud. It is a container for different types of files and serves independently from other units in the platform. Every bucket should have a globally unique name to ensure no two buckets in the world have the

same name. It is essential to select a suitable region for our bucket and it should be as close as possible to our clusters. This study selects European regions for the bucket. Figures 5-9 show the steps to create a bucket.

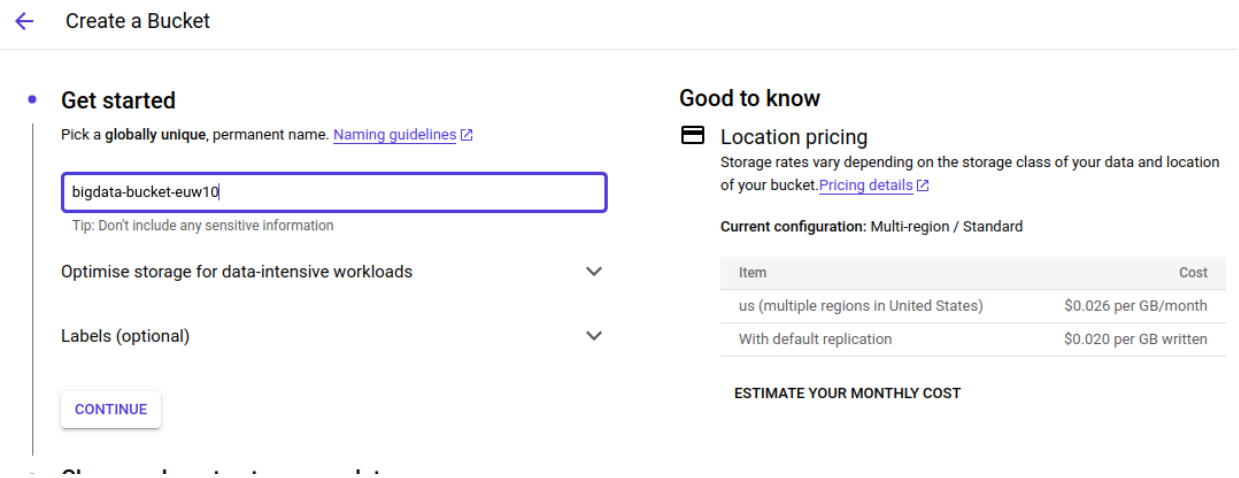

Figure 5: Specify The Bucket Unique Name

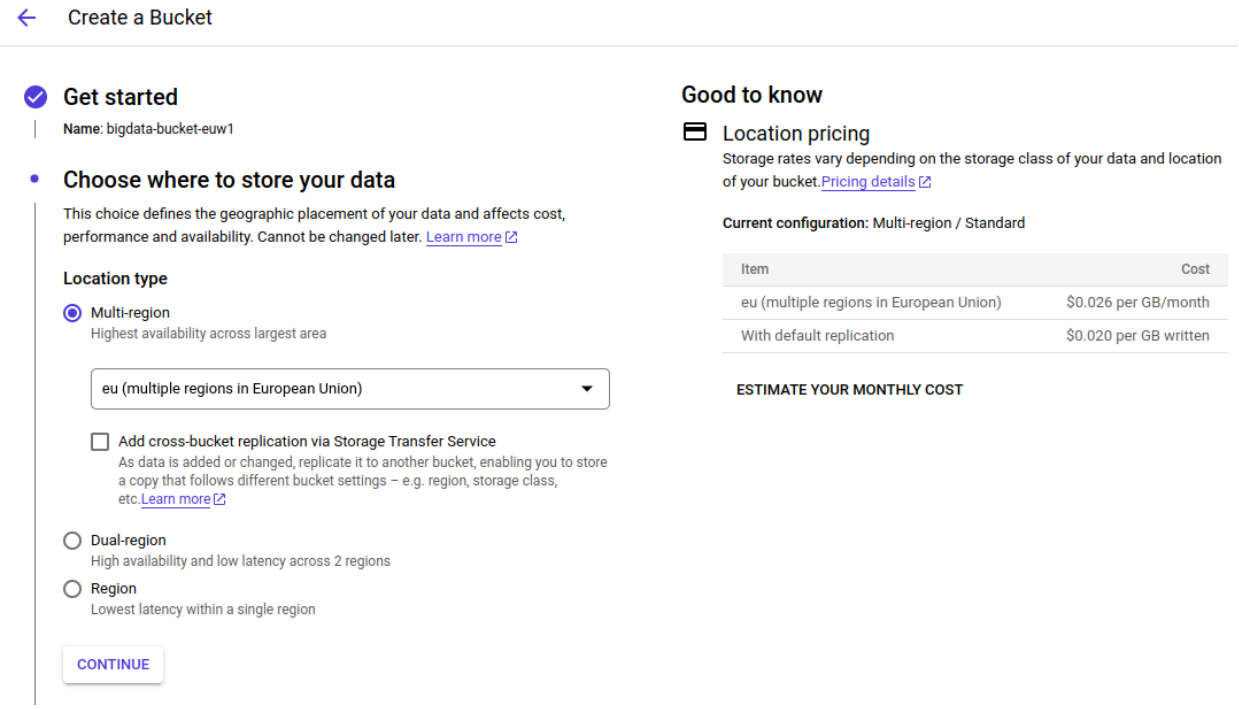

Figure 6: Select Where to Create The Bucket (Region)

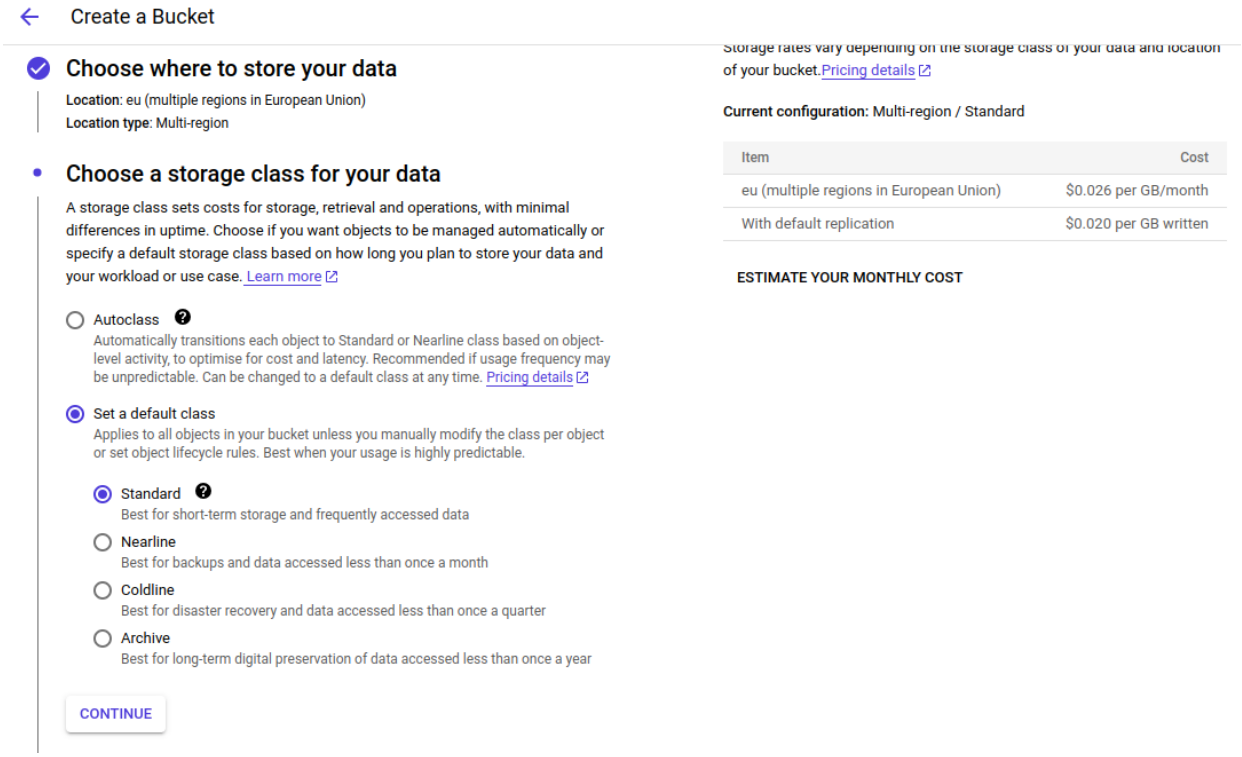

Figure 7: Define how to store Data in The Bucket

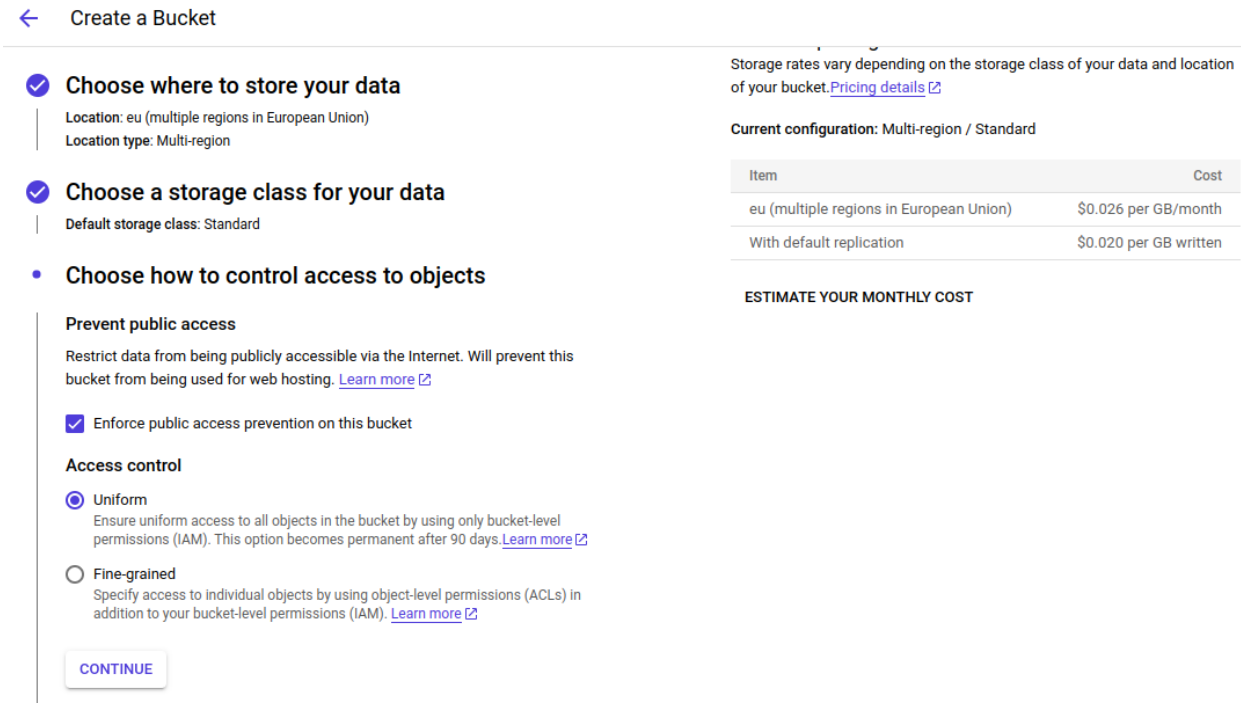

Figure 8: Manage Access Control

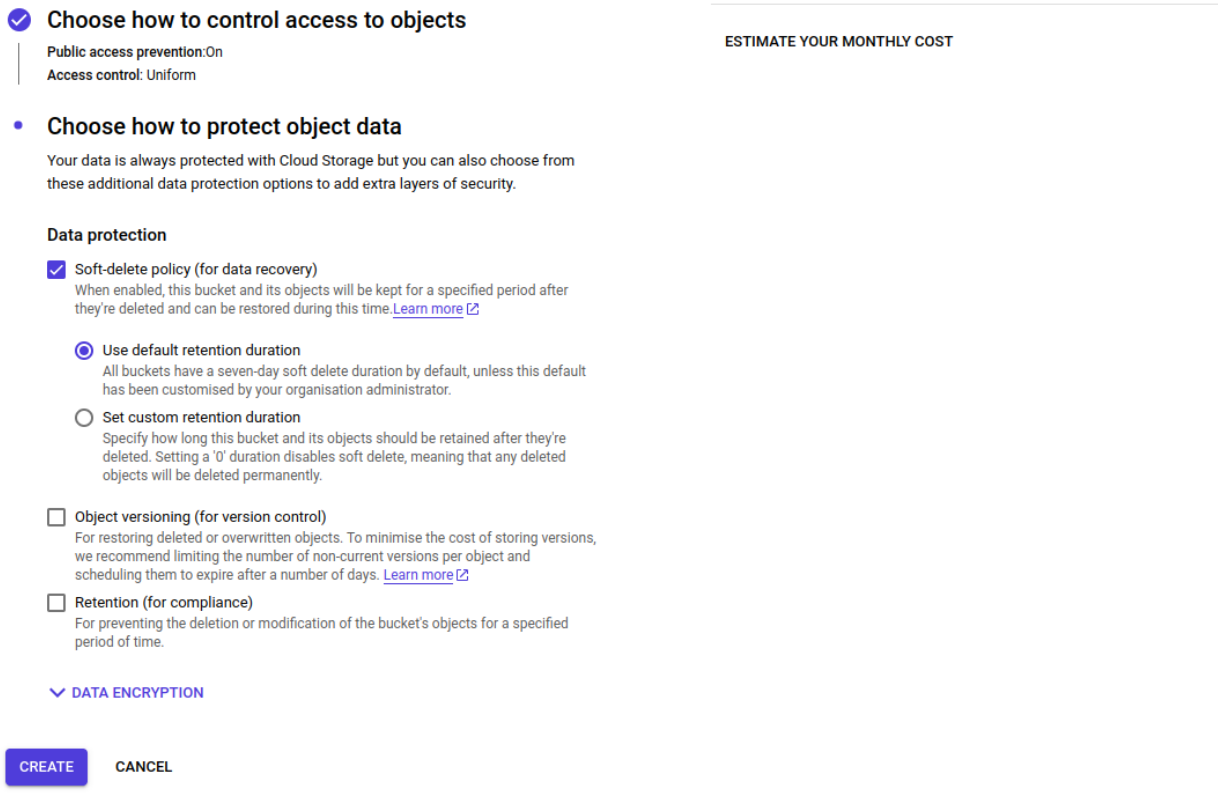

Figure 9: Select How to Protect Data

The next step is to enable some APIs (Application Programming Interface) which are responsible for the management of Dataproc clusters and their security issues. The following APIs need to be enabled:

- Cloud Dataproc API
- Dataproc Resource Manager API
- Compute Engine API
- Cloud Resource Manager API
- IAM Service Account Credentials API

To find an API and enable it, we should search for it in the APIs and Services section (Figure 10).

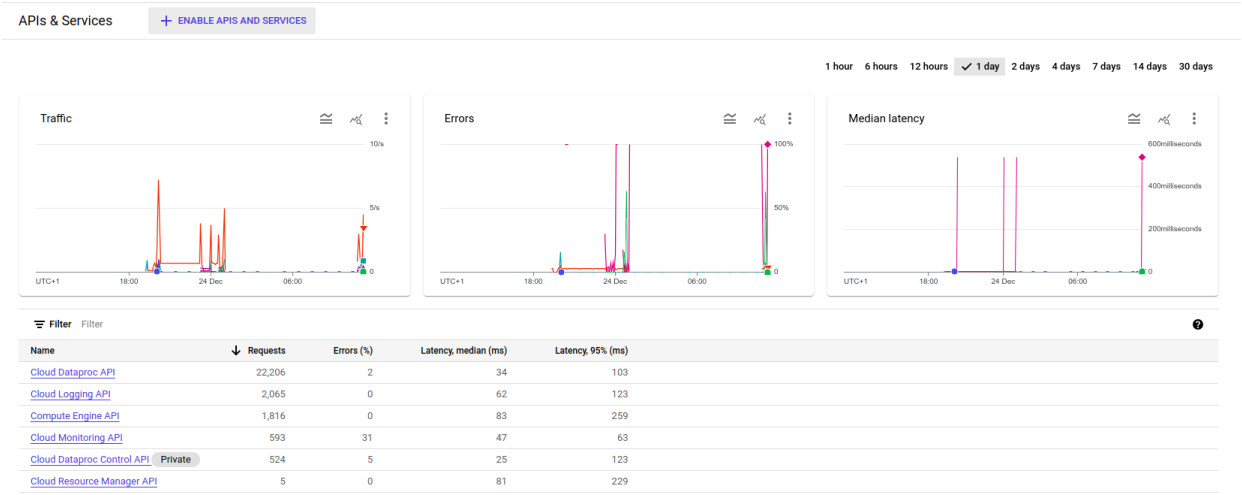

Figure 10: APIs and Services Section

After finding the API we should enable it like below (Figure 11).

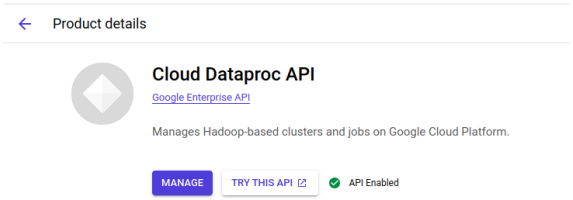

Figure 11: Enable an API

The last step before creating a Dataproc cluster is to manage access permissions for our project through IAM (Identity and Access Management). They should be done like below (Figure 12).

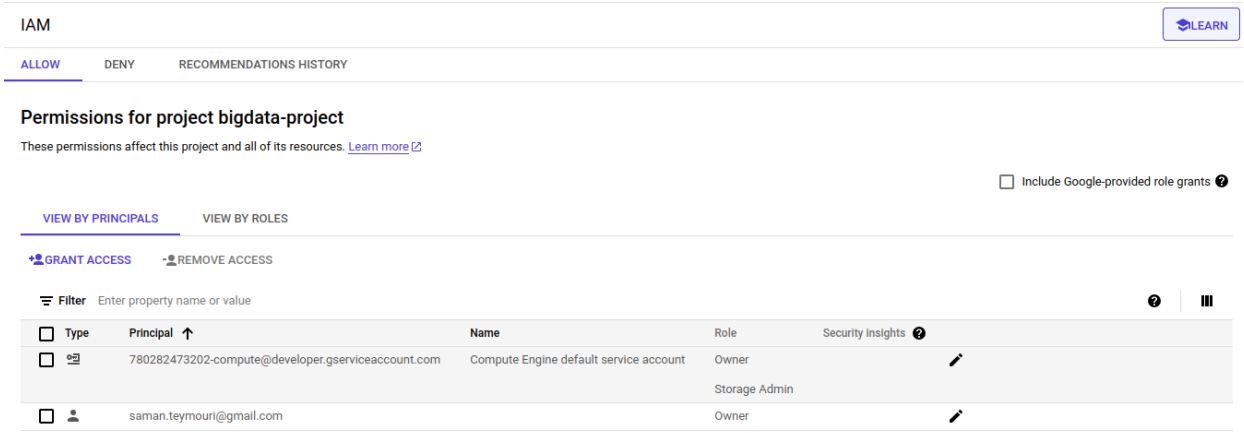

Figure 12: Manage Access Permission via IAM

Finally, we can create a Dataproc cluster to perform our jobs. At first, it is necessary to assign a proper name and region to the cluster. This study creates a cluster in europe-west10 region because it is placed in Berlin (Figure 13). The other options on the setup cluster page need to remain unchanged.

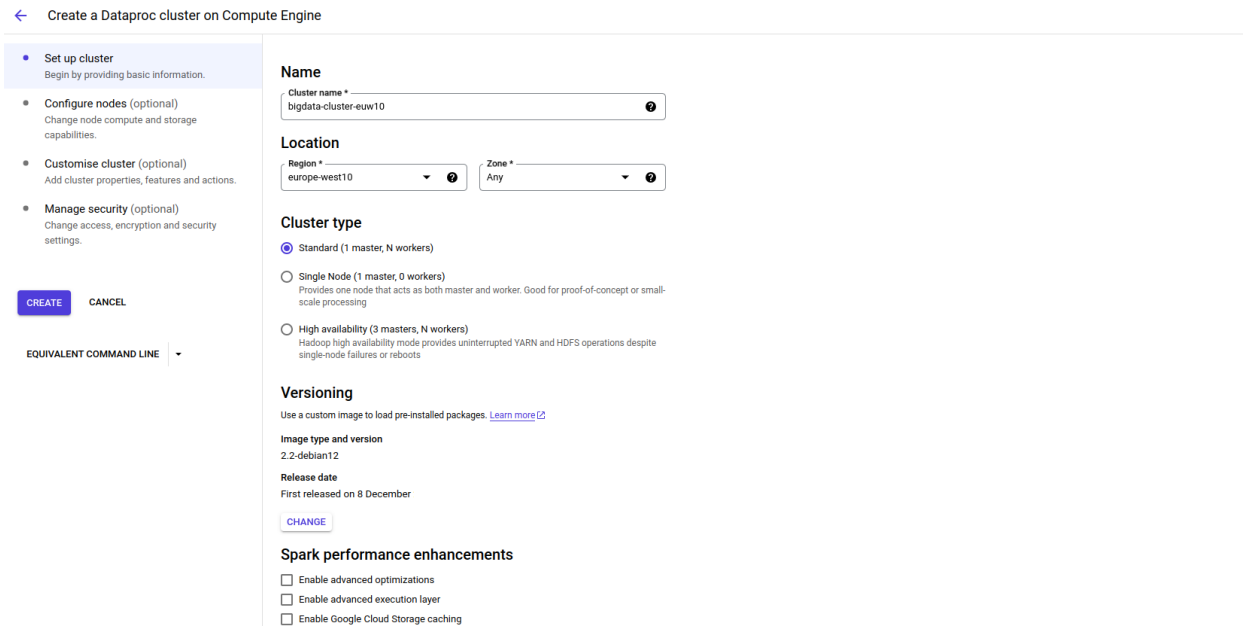

Figure 13: Dataproc Cluster Name and Region

On the next page (Configure Nodes), CPU Platform is set to E2 which means CPU platform selection is based on availability. This assures high availability for our cluster. The machine type is set to e2-standard2 (lowest configuration) with 2 vCPUs and 8GB of memory. For the primary disk type, we choose Standard Persistent Disk since it is much less expensive in comparison with

other options. We did the same configuration for both the Manager and working nodes (Figure 14).

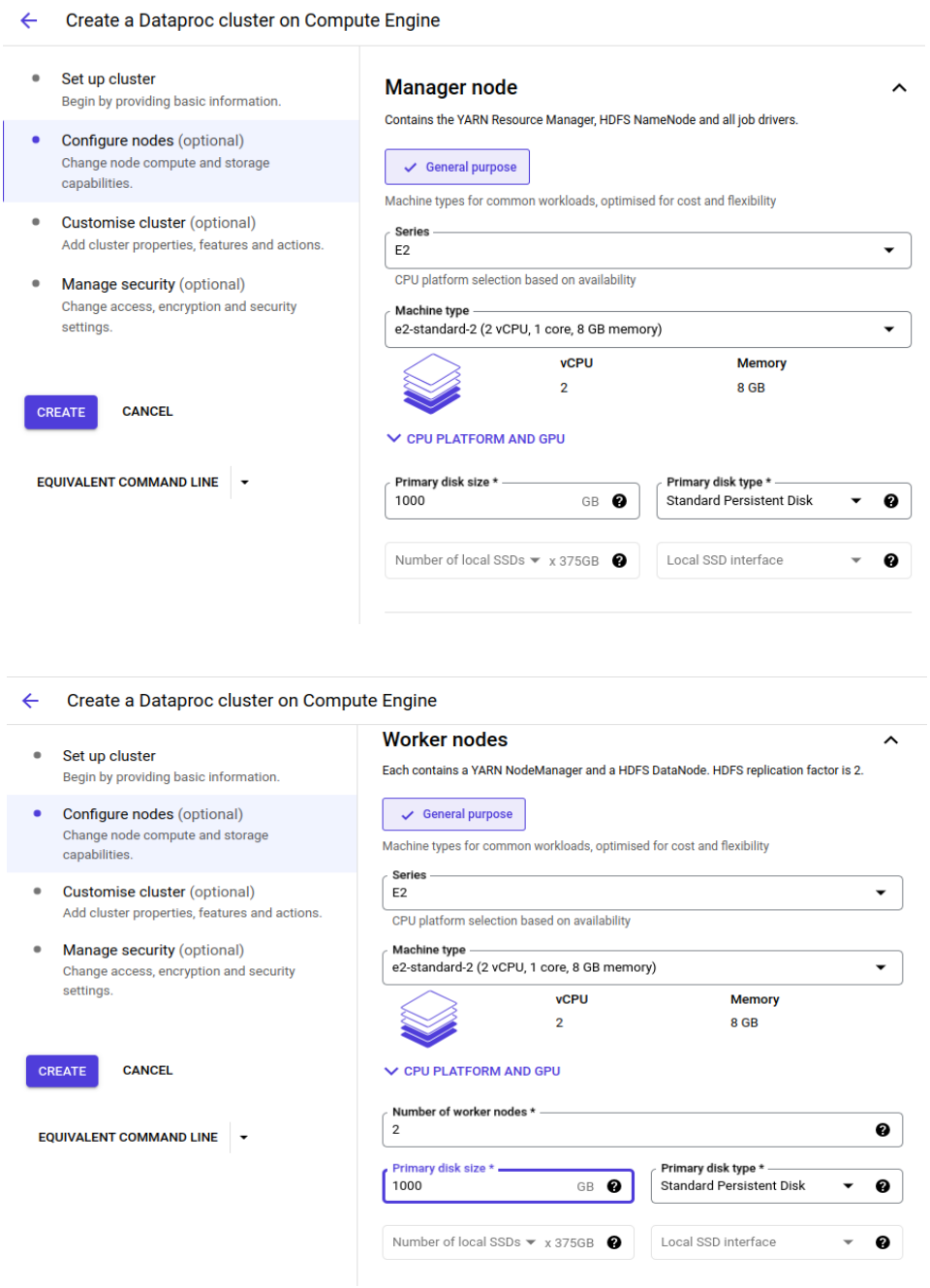

Figure 14: Configure Nodes of The Cluster

On the third page, we should uncheck the Internal IP Only options as it is not necessary (Figure 15). The rest of the options on this page and the forth page should remain unchanged.

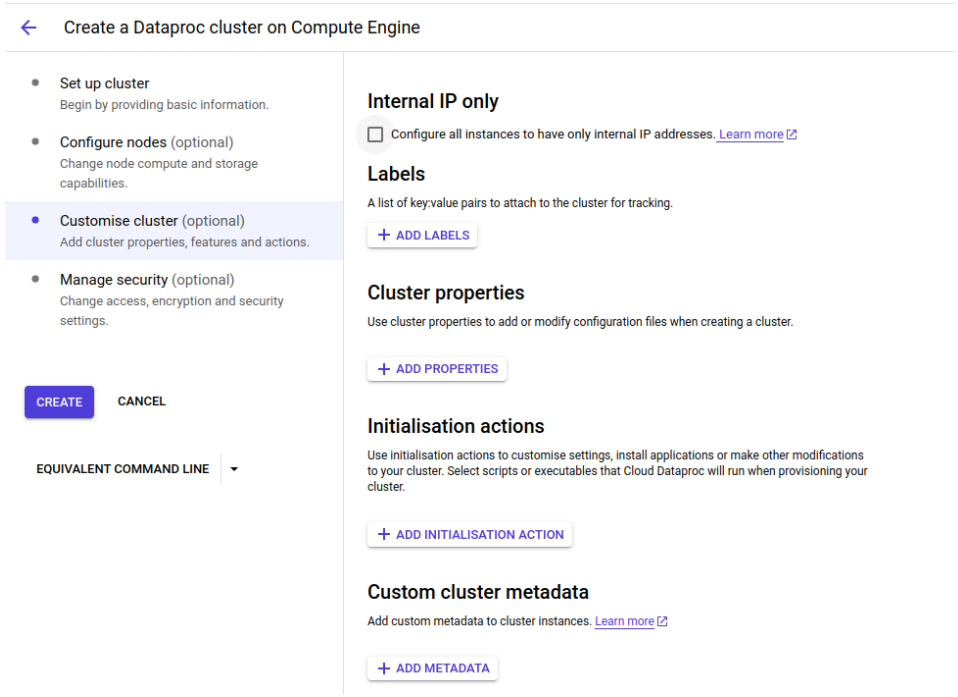

Figure 15: Customize The Cluster

3.3. Dataset uploading process

The dataset is uploaded via GCP gsutil interface (Figure 16)

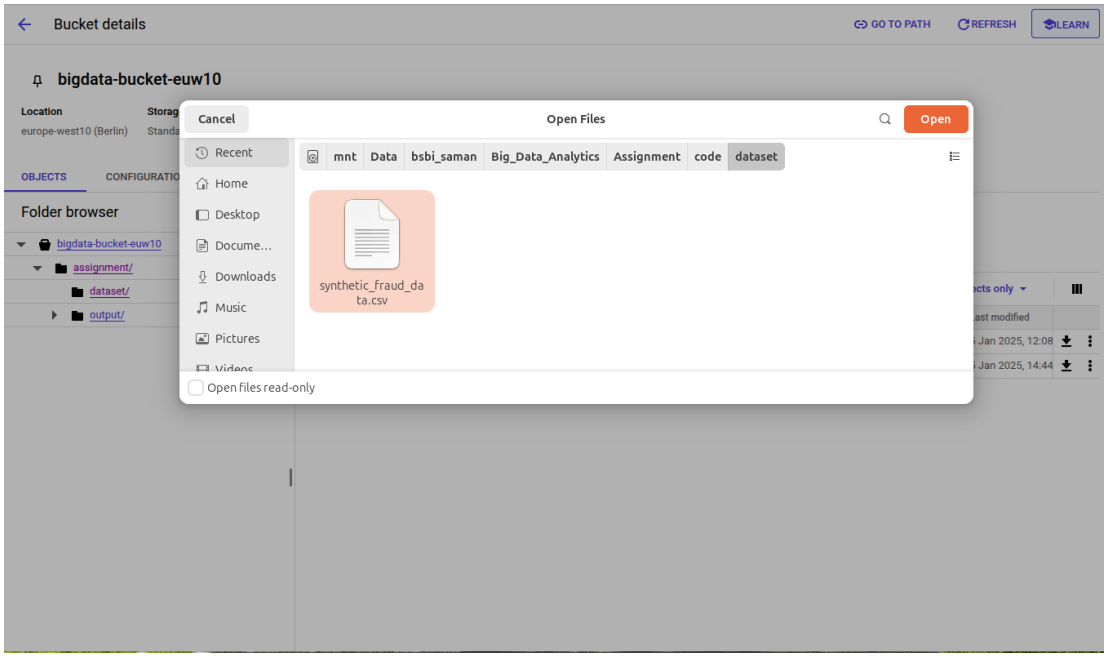

Figure 16: Upload The Dataset

There are some facilities to prevent data loss. We can create a replica bucket to replicate some objects to another bucket which is designed to store them for a long time (Nearline Storage Class) (Figure 17)

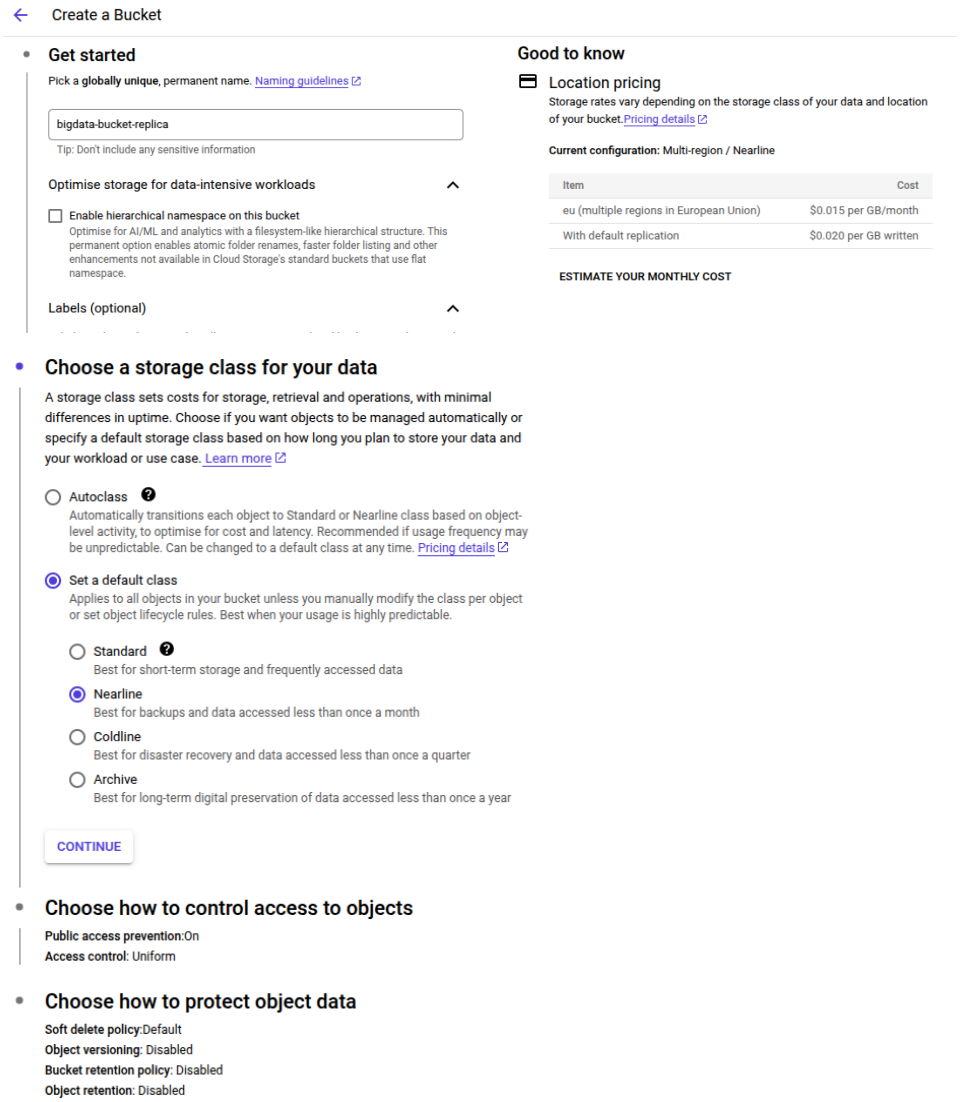

Figure 17: Create The Replica Bucket

At this stage, we should assign the replica bucket to the main bucket to perform replication (Figure 18). In the include objects with prefixes, we can specify which folders are needed to be backed up.

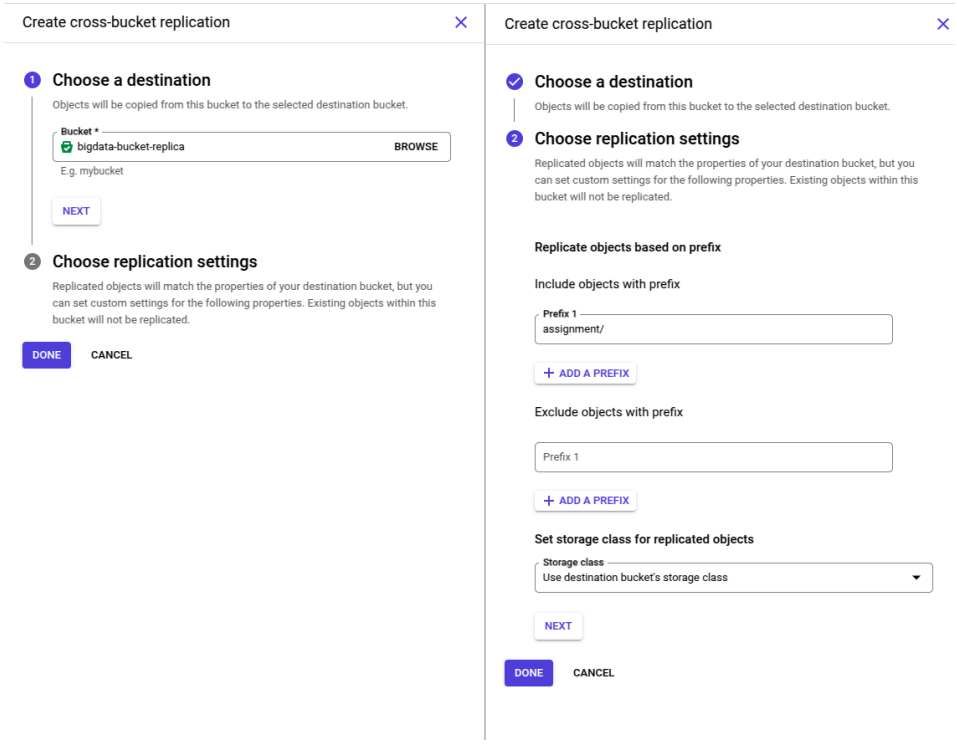

Figure 18: Assign The Replica Bucket to The Main One

4. Data Processing with Spark and Map Reduce

4.1. Data Processing using Map Reduce

Map reduce programming model, as its name shows, consists of mapper and reducer functions. The mapper function applies the transformation of data into the desired format and the reducer function aggregates the results.

This study uses PySpark RDD (Resilient Distributed Dataset) to apply a map and reduce operations to the data (Zaharia et al., 2012). At first, we need to start a PySpark Session and open the dataset (Figure 19). This function reads the CSV file then converts it to RDD and returns it. Meanwhile, it prints the dataset schema using show_schema function (Figure 20).

```python
from pyspark.sql import SparkSession, Row
import sys
from os import path
from general import *

@show_elapsed_time
def read_data(input_path, logger : Logger):
    # open spark session
    spark = SparkSession.builder \
            .appName("Data Preparation MapReduce") \
            .config("spark.driver.memory", "4g") \
            .config("spark.executor.memory", "4g") \
            .getOrCreate()

    # read data from csv file
    df = spark.read.csv(input_path, inferSchema = True, header = True)

    # show schema of original data
    msg = "The schema of data before preparation is"
    show_schema(df, msg, logger)

    # convert dataframe to RDD for mapreduce operations
    rdd = df.rdd

    return spark, rdd
```

Figure 19: Read The Dataset

All functions in this program are called through the main function (Figure 21). The main function gets the input and output paths from the terminal, then checks data, and finally prepares data for predictive analytics. As it is in the code, there are some extra functions and decorations that help the program to save the time elapsed and log the outputs of the terminal. All these extra functions are prepared in a separate module named general (Figure 23). These functions are used in all programs including PySpark preparation and machine learning scripts.

```python
def show_schema(df, msg : str, logger:Logger):
    # show data schema and log it
    print(msg)
    content = capture_terminal_output(df.printSchema)
    logger.log(msg, content)
```

Figure 20: Show The Dataset Schema

```python
@show_elapsed_time
def main():
    # check if two arguments has been passed to the program
    if len(sys.argv)<3:
        print("The program needs two parameters!")
        sys.exit(1)
    # retrieve arguments values
    input_path = sys.argv[1]
    output_path = sys.argv[2]

    # create a logger instance
    logger = Logger()

    # read data from the input file
    spark, rdd = read_data(input_path, logger)

    # check data quality and make a report
    check_data_quality(spark, rdd, logger)

    # prepare data for the analysis
    prepared_rdd = prepare_data(spark, rdd, output_path, logger)

    # Save log file
    logger.save_file(path.join(output_path,"log"),spark)

    # stop spark session
    spark.stop()

if __name__ == "__main__":
    main()
```

Figure 21: Main Function For Map Reduce Data Preparation

Function map_reduce is responsible for implementing map-reduce operations (Figure 22). This function gets a map function and applies it to the RDD. Afterward, it reduces the results by the keys. It is compatible with tuples with one or two values. The function sorts the values based on the keys and returns the whole or part of the results.

```python
def map_reduce(rdd, map_function, msg : str, logger : Logger, limit : int = -1, has_two_output : bool = False):
    # map reduce operation
    mapped_rdd = rdd.map(map_function)

    # reduce the mapped data based on keys
    if not has_two_output:
        # when each key has one value
        reduced_rdd = mapped_rdd.reduceByKey(lambda x, y: x + y)
    else:
        # when each key has a pair of values
        reduced_rdd = mapped_rdd.reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))

    # sort the categories by key in ascending order
    sorted_rdd = reduced_rdd.sortBy(lambda x: x[0], ascending=True)

    # gather up the results
    if limit > 0:
        # if limit is specified use take for better performance
        result = sorted_rdd.take(limit)
    else:
        # collect all records in result
        result = sorted_rdd.collect()

    # print the result
    content = ""
    print(msg)
    for title, count in result:
        # check value or one of the pair of values greater than zero
        if count>0 if not has_two_output else count[0]>0 or count[1]>0:
            content += f"{title}: {count}\n"
            print(f"{title}: {count}")

    # log the result
    logger.log(msg, content)
```

Figure 22: Map Reduce Function

```python
import time
from pyspark.sql import SparkSession
from pyspark.sql.types import StringType
import sys
from io import StringIO

class Logger:
    # create class variables
    logged = ""
    msg_counter : int
    def __init__(self):
        # class initialization
        self.logged = ""
        self.msg_counter = 0

    def log(self, msg : str, content : str):
        # set message counter
        self.msg_counter += 1
        # store msg title and its content
        self.logged += " ".join([str(self.msg_counter), msg, "\n"])
        self.logged += "\n".join([content, ""])

    def save_file(self, folder_path : str, spark : SparkSession):
        # save text file based on the logged string
        logged_df = spark.createDataFrame([self.logged], StringType()).toDF("logs")
        logged_df.coalesce(1).write.mode("overwrite").text(folder_path)

def capture_terminal_output(func, *args, **kwargs):
    # capture whatever is passed to the terminal, because pyspark dataframe functions only pass well structured output to the terminal
    try:
        # redirect sys.stdout to capture output
        buffer = StringIO()
        old_stdout = sys.stdout
        # redirect stdout to StringIO buffer
        sys.stdout = buffer

        # show as many rows as requested with passing proper parameters
        func(*args, **kwargs)
        # ensure all output is flushed to the buffer
        sys.stdout.flush()

        # retrieve the captured output as a string
        content = buffer.getvalue()
    finally:
        # restore the original stdout
        sys.stdout = old_stdout

    # print the content because we grab all of it from terminal
    print(content)

    return content

def is_float(num : str):
    # check if string is a float or not
    try:
        f_num = float(num)
    except:
        return False
    return True

def show_elapsed_time(original_func):
    # decorator function to define start and end of function and print elapsed time
    def wrapper_function(*args, **kwargs):
        print(f"\n>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>function {original_func.__name__} started\n")
        current_time = time.time()
        result = original_func(*args, **kwargs)
        elapsedtime = int(time.time() - current_time)
        print(f"\n>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>function {original_func.__name__} ended ==> Elapsed time: {elapsedtime} seconds\n")
        if result : return result

    return wrapper_function
```

Figure 23: General Module

A function named check_data_quality is used to extract exploratory information from the dataset (Figure 24). It uses the map_reduce function for each analysis.

```python
@show_elapsed_time
def check_data_quality(spark, rdd, logger:Logger):
    # check data quality

    # check how many fraud vs geniune transactions
    msg = "how many fraud vs geniune transactions"
    map_reduce(rdd, lambda record: ("fraud", 1) if record["is_fraud"] == True else ("genuine", 1), msg, logger)

    # show distinct merchants and their categories and types which fraud is detected with expectation
    msg = "merchants and their categories and types which fraud is detected with expectation"
    map_reduce(rdd, lambda record: ("-".join([record["merchant_category"], record["merchant_type"], record["merchant"]]), (1 if record["high_risk_merchant"] == True else 0

    # check where frauds are detected
    msg = "places where frauds are detected"
    map_reduce(rdd, lambda record: ("-".join([record["country"], record["city_size"], record["currency"]]), 1 if record["is_fraud"] == True else 0) , msg, logger)

    # check how much fraud amount is detected
    msg = "how much fraud amount is detected "
    map_reduce(rdd, lambda record: ("-".join([record["country"], record["city_size"], record["currency"]]), record["amount"] if record["is_fraud"] == True else 0) , msg, l

    # check on which devices and channels frauds are detected
    msg = "on which devices and channels frauds are detected"
    map_reduce(rdd, lambda record: ("-".join([record["channel"], record["device"]]), 1 if record["is_fraud"] == True else 0) , msg, logger)

    # check which customer_ids have the most fraudulant transactions
    msg = "customers with the most fraudulant transactions (limit 10)"
    map_reduce(rdd, lambda record: (record["customer_id"], 1 if record["is_fraud"] == True else 0) , msg, logger, limit=10)
```

Figure 24: Check Data Quality Function Using Map Reduce

After reviewing the code, it is time to submit the map-reduce job to the DataProc cluster (Figure 25)

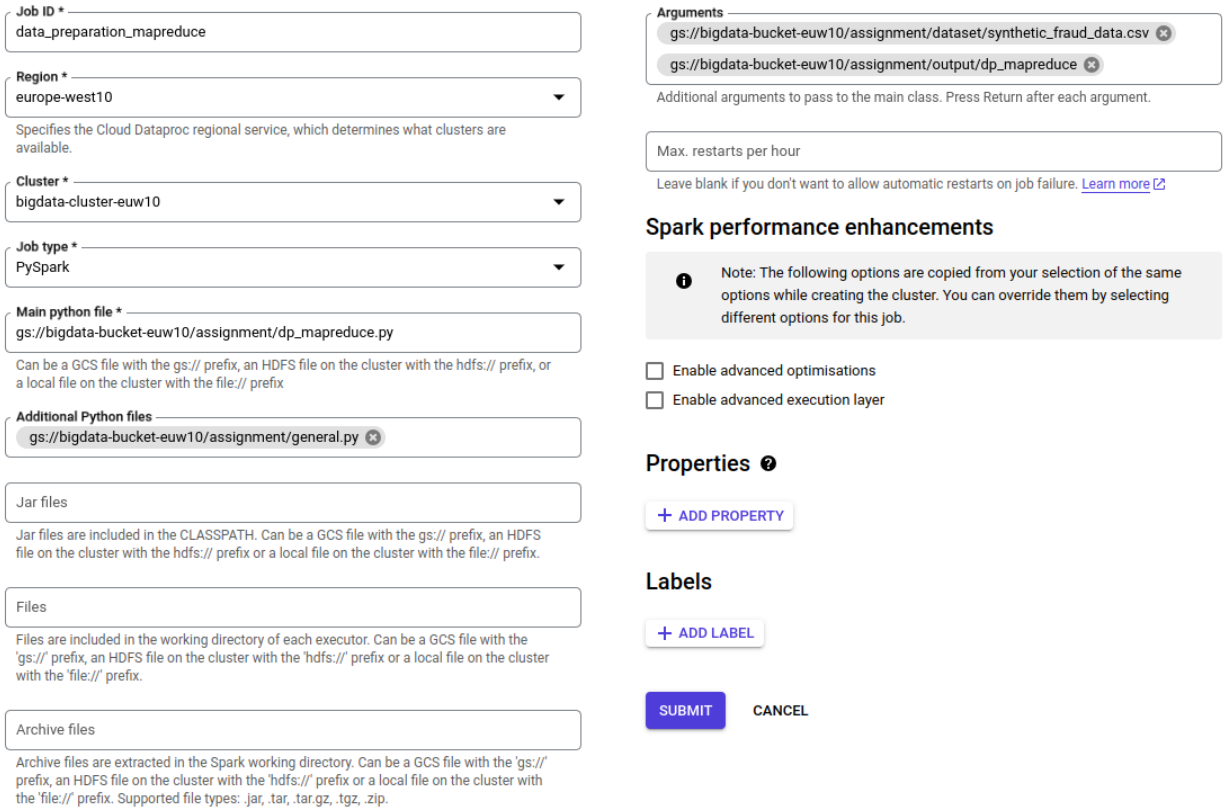

Figure 25: Submit The Map Reduce Job

Here we can explore the results of each analysis one by one. The first analysis is about checking how many fraudulent and genuine transactions exist in the dataset (Figure 26).

```
2 how many fraud vs geniune transactions
fraud: 1494719
genuine: 5989047
```

Figure 26: Number of Fraudulent vs Genuine Transactions (Map Reduce)

The second analysis groups data by the combination of merchant_category, merchant_type, and merchant then calculates the number of high-risk merchants and fraudulent transactions for each group (Figure 27). This analysis shows the relationship between being a high-risk merchant and facing fraudulent transactions.

```
3 merchants and their categories and types which fraud is detected with expectation
Education-online-Coursera: (0, 18658)
Education-online-MasterClass: (0, 18475)
Education-online-Skillshare: (0, 18663)
Education-online-Udemy: (0, 18622)
Education-online-edX: (0, 18545)
Education-supplies-Barnes & Noble: (0, 30933)
Education-supplies-Chegg: (0, 31171)
Education-supplies-University Bookstore: (0, 31136)
Entertainment-events-AMC Theaters: (62291, 12424)
Entertainment-events-LiveNation: (62350, 12464)
Entertainment-events-Regal Cinemas: (62519, 12361)
Entertainment-events-StubHub: (62696, 12710)
```

Figure 27: Some Rows of High Risk Merchant vs Fraud (Map Reduce)

The next analysis groups data by the combination of country, city_size, and currency to depict the number of fraudulent transactions for each area (Figure 28).

```
4 places where frauds are detected
Australia-medium-AUD: 37652
Brazil-medium-BRL: 298629
Canada-medium-CAD: 37278
France-medium-EUR: 37426
Germany-medium-EUR: 37205
Japan-medium-JPY: 37592
Mexico-medium-MXN: 298841
Nigeria-medium-NGN: 298600
Russia-medium-RUB: 299425
Singapore-medium-SGD: 37414
UK-medium-GBP: 37345
USA-large-USD: 14855
USA-medium-USD: 22457
```

Figure 28: Number of Fraud in Each Area (Map Reduce)

The other analysis is the same as the previous one except it shows the amount of fraud for each area instead of the number of them (Figure 29).

```
5 how much fraud amount is detected
Australia-medium-AUD: 57719986.45000002
Brazil-medium-BRL: 1683434472.5000012
Canada-medium-CAD: 53072063.569999985
France-medium-EUR: 36234521.29
Germany-medium-EUR: 36019634.64
Japan-medium-JPY: 4719137393.679998
Mexico-medium-MXN: 6767799466.030005
Nigeria-medium-NGN: 138728842683.58994
Russia-medium-RUB: 25321122840.909985
Singapore-medium-SGD: 57080250.000000015
UK-medium-GBP: 30619477.790000007
USA-large-USD: 16712723.540000001
USA-medium-USD: 25345965.02999999
```

Figure 29: Amount of Fraud in Each Area (Map Reduce)

This study offers another analysis that shows how many fraudulent transactions happen through each channel and device (Figure 30).

```
6 on which devices and channels frauds are detected
mobile-Android App: 140844
mobile-iOS App: 140306
pos-Chip Reader: 217324
pos-Magnetic Stripe: 217204
pos-NFC Payment: 216519
web-Chrome: 140087
web-Edge: 138885
web-Firefox: 142171
web-Safari: 141379
```

Figure 30: Number of Fraud Happens Through Each Channel and Device (Map Reduce)

The last information is about some customers who have fraudulent transactions (Figure 31).

```
7 customers with fraudulant transactions (limit 10)
CUST_10000: 311
CUST_10018: 256
CUST_10022: 366
CUST_10039: 314
CUST_10102: 360
CUST_10109: 258
CUST_10129: 364
CUST_10150: 216
CUST_10161: 241
CUST_10198: 340
```

Figure 31: Some Customers Who Have Fraudulent Transactions (Map Reduce)

After this analysis, it is time to prepare the dataset for machine learning (Figure 32). This function has two inner functions which receive a row and do some modifications on it. The RDD applies these inner functions for mapping its records to the new format. This operation transforms data in a map-reduced way. The split_timestamp inner function gets a record and extracts data, hour, minute, and second from its timestamp column to create new columns. The other inner function cast_bool_int converts boolean columns such as card_present, high_risk_merchant, weekend_transaction, and is_fraud to integers. The next step is to select only usable columns which is done by using a lambda function to extract only needed columns from each row. Afterward, the schema of the newly prepared dataset is printed (Figure 33) and the first 1000 rows of it build a new CSV file named processed_data_csv_top1000.

```python
@show_elapsed_time
def prepare_data(spark, rdd, output_path, logger):
    # preparing data and eliminate unused fields

    # split data-time column (timestamp)
    def split_timestamp(row):
        row_dict = row.asDict()
        timestamp = row_dict["timestamp"]
        row_dict["date"] = timestamp.year*10000+timestamp.month*100+timestamp.day
        row_dict["time_hour"] = timestamp.hour
        row_dict["time_minute"] = timestamp.minute
        row_dict["time_second"] = timestamp.second

        return Row(**row_dict)


    # change boolean columns to integer
    def cast_bool_int(row):
        row_dict = row.asDict()
        row_dict["card_present"] = 1 if row_dict["card_present"] == True else 0
        row_dict["high_risk_merchant"] = 1 if row_dict["high_risk_merchant"] == True else 0
        row_dict["weekend_transaction"] = 1 if row_dict["weekend_transaction"] == True else 0
        row_dict["is_fraud"] = 1 if row_dict["is_fraud"] == True else 0

        return Row(**row_dict)

    # apply transformations
    rdd = rdd.map(split_timestamp).map(cast_bool_int)

    # select only usable fields
    fields = ["customer_id", "card_number", "date", "time_hour", "time_minute", "time_second", "merchant_category", "merchant_type", "merchant", "amount", "currency", "cour
    rdd_cleaned = rdd.map(lambda row: Row(**{field: row[field] for field in fields}))

    # convert back to DataFrame for printing schema and writing to file
    df_cleaned = spark.createDataFrame(rdd_cleaned)

    # cleaned data schema
    msg = "The schema of data after preparation is"
    show_schema(df_cleaned, msg, logger)

    # save the first 100 rows of cleaned data as a CSV
    df_cleaned_top1000 = df_cleaned.limit(1000)
    df_cleaned_top1000.write.mode("overwrite").csv(path.join(output_path, "processed_data_csv_top1000"), header=True)
```

Figure 32: Function for Data Preparation Using Map Reduce

```
8 The schema of data after preparation is
root
 |-- customer_id: string (nullable = true)
 |-- card_number: long (nullable = true)
 |-- date: long (nullable = true)
 |-- time_hour: long (nullable = true)
 |-- time_minute: long (nullable = true)
 |-- time_second: long (nullable = true)
 |-- merchant_category: string (nullable = true)
 |-- merchant_type: string (nullable = true)
 |-- merchant: string (nullable = true)
 |-- amount: double (nullable = true)
 |-- currency: string (nullable = true)
 |-- country: string (nullable = true)
 |-- city: string (nullable = true)
 |-- city_size: string (nullable = true)
 |-- card_type: string (nullable = true)
 |-- card_present: long (nullable = true)
 |-- device: string (nullable = true)
 |-- channel: string (nullable = true)
 |-- distance_from_home: long (nullable = true)
 |-- high_risk_merchant: long (nullable = true)
 |-- transaction_hour: long (nullable = true)
 |-- weekend_transaction: long (nullable = true)
 |-- is_fraud: long (nullable = true)
```

Figure 33: Schema of The Dataset After Preparation

4.2. Data Processing using Spark

This section states all the steps to prepare data using Spark. It has a main function very similar to the map-reduce main function. It reads data and prints its schema, then checks data quality and prepares data for predictive analytics. So to avoid repetition, this paper does not include those figures here.

```
@show_elapsed_time
def check_data_quality(spark, df, logger:Logger):
    # check data quality

    # create a view based on the dataframe
    df.createOrReplaceTempView("transactions")

    # show all columns for 10 first rows
    result = spark.sql("select * from transactions limit 10")
    msg = "all columns for 10 first rows"
    show_query_result(result, msg, logger)

    # check how many fraud vs geniune transactions
    result = spark.sql("select is_fraud, count(*) as count from transactions group by is_fraud")
    msg = "how many fraud vs geniune transactions"
    show_query_result(result, msg, logger)

    # show distinct merchants and their categories and types which fraud is detected with expectation
    result = spark.sql("select distinct merchant_category, merchant_type, merchant, sum(case when high_risk_merchant = True then 1 else 0 end) as high_risk_count, sum(case
    msg = "merchants and their categories and types which fraud is detected with expectation"
    show_query_result(result, msg, logger)

    # check where frauds are detected
    result = spark.sql("select country, city_size, currency, sum(case when is_fraud = True then 1 else 0 end) as is_fraud_count from transactions group by country, city_si
    msg = "places where frauds are detected"
    show_query_result(result, msg, logger)

    # check how much fraud amount is detected
    result = spark.sql("select country, currency, format_number(cast(sum(amount) as decimal(18,2)),2) as sum_amount from transactions where is_fraud = True group by countr
    msg = "how much fraud amount is detected"
    show_query_result(result, msg, logger)

    # check on which devices and channels frauds are detected
    result = spark.sql("select channel, device, count(*) as count from transactions where is_fraud = True group by channel, device order by channel, device, count desc")
    msg = "on which devices and channels frauds are detected "
    show_query_result(result, msg, logger)

    # check which customer_ids have the most fraudulant transactions
    result = spark.sql("select customer_id, count(*) as count from transactions where is_fraud = 1 group by customer_id order by count desc limit 10")
    msg = "customers with the most fraudulant transactions (limit 10)"
    show_query_result(result, msg, logger)
```

Figure 34: Exploratory Analysis Using Spark

```
def show_query_result(result, msg : str, logger:Logger, limit : int = -1):
    # show query result and log it
    msg = " ==> ".join([msg, f"record count: {result.count()}"])
    print(msg)

    # show as many rows as requested
    content = capture_terminal_output(result.show, truncate=False, n=limit if limit > 0 else result.count())
    logger.log(msg, content)
```

Figure 35: Function for Showing SparkSQL Query Results

The check_data_quality function uses SparkSQL to analyze the dataset (Figure 34). At first, it needs to create a temporary view based on the PySpark dataframe named transactions. Afterward, it shows the top 10 records for all columns. The results of each query are shown using the show_query_result function (Figure 35).

Since the spark job is expected to do the exactly same as map-reduce does, to avoid repetition, this paper only shows the figures of spark job results. It is much easier to do the analysis via SparkSQL, and it also helps us to provide better results. At first, the spark job is submitted (Figure 36).

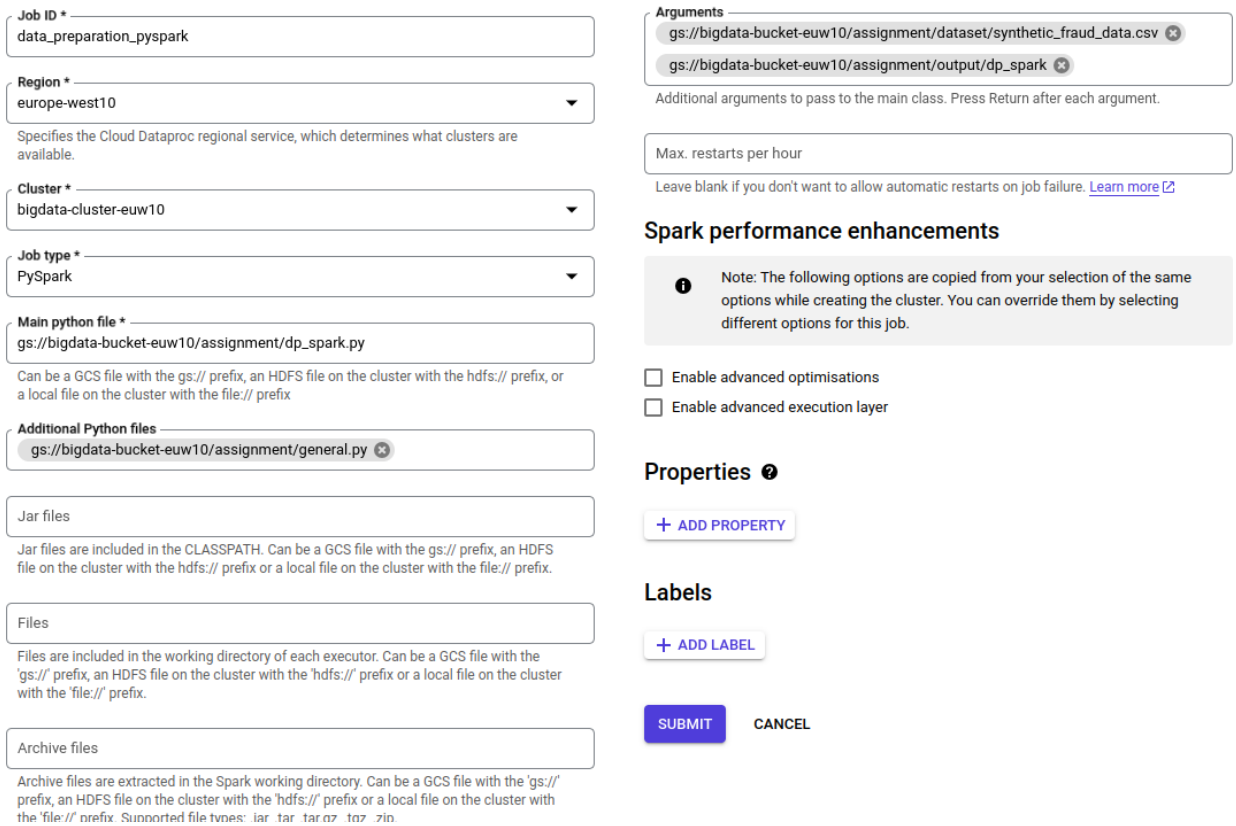

Figure 36: Submit The Spark Data Preparation Job

The below results come out after the job executes.

```
3 how many fraud vs geniune transactions ==> record count: 2
+--------+-------+
|is_fraud|count  |
+--------+-------+
|true    |1494719|
|false   |5989047|
+--------+-------+
```

Figure 37: Number of Fraudulent vs Genuine Transactions (Spark)

```
4 merchants and their categories and types which fraud is detected with expectation ==> record count: 105
+-----------------+-------------+--------------------+---------------+--------------+
|merchant_category|merchant_type|merchant            |high_risk_count|is_fraud_count|
+-----------------+-------------+--------------------+---------------+--------------+
|Education        |online       |Coursera            |0              |18658         |
|Education        |online       |MasterClass         |0              |18475         |
|Education        |online       |Skillshare          |0              |18663         |
|Education        |online       |Udemy               |0              |18622         |
|Education        |online       |edX                 |0              |18545         |
|Education        |supplies     |Barnes & Noble      |0              |30933         |
|Education        |supplies     |Chegg               |0              |31171         |
|Education        |supplies     |University Bookstore|0              |31136         |
|Entertainment    |events       |AMC Theaters        |62291          |12424         |
|Entertainment    |events       |LiveNation          |62350          |12464         |
|Entertainment    |events       |Regal Cinemas       |62519          |12361         |
|Entertainment    |events       |StubHub             |62696          |12710         |
```

Figure 38: Some Rows of High Risk Merchant vs Fraud (Spark)

```
5 places where frauds are detected ==> record count: 13
+---------+---------+--------+--------------+
|country  |city_size|currency|is_fraud_count|
+---------+---------+--------+--------------+
|Russia   |medium   |RUB     |299425        |
|Mexico   |medium   |MXN     |298841        |
|Brazil   |medium   |BRL     |298629        |
|Nigeria  |medium   |NGN     |298600        |
|Australia|medium   |AUD     |37652         |
|Japan    |medium   |JPY     |37592         |
|France   |medium   |EUR     |37426         |
|Singapore|medium   |SGD     |37414         |
|UK       |medium   |GBP     |37345         |
|Canada   |medium   |CAD     |37278         |
|Germany  |medium   |EUR     |37205         |
|USA      |medium   |USD     |22457         |
|USA      |large    |USD     |14855         |
+---------+---------+--------+--------------+
```

Figure 39: Number of Fraud in Each Area (Spark)

```
6 how much fraud amount is detected ==> record count: 12
+---------+--------+------------------+
|country  |currency|sum_amount        |
+---------+--------+------------------+
|Nigeria  |NGN     |138,728,842,683.59|
|Russia   |RUB     |25,321,122,840.91 |
|Mexico   |MXN     |6,767,799,466.03  |
|Japan    |JPY     |4,719,137,393.68  |
|Brazil   |BRL     |1,683,434,472.50  |
|Australia|AUD     |57,719,986.45     |
|Singapore|SGD     |57,080,250.00     |
|Canada   |CAD     |53,072,063.57     |
|USA      |USD     |42,058,688.57     |
|France   |EUR     |36,234,521.29     |
|Germany  |EUR     |36,019,634.64     |
|UK       |GBP     |30,619,477.79     |
+---------+--------+------------------+
```

Figure 40: Amount of Fraud in Each Area (Spark)

```
7 on which devices and channels frauds are detected  ==> record count: 9
+-------+---------------+------+
|channel|device         |count |
+-------+---------------+------+
|mobile |Android App    |140844|
|mobile |iOS App        |140306|
|pos    |Chip Reader    |217324|
|pos    |Magnetic Stripe|217204|
|pos    |NFC Payment    |216519|
|web    |Chrome         |140087|
|web    |Edge           |138885|
|web    |Firefox        |142171|
|web    |Safari         |141379|
+-------+---------------+------+
```

Figure 41: Number of Fraud Happens Through Each Channel and Device (Spark)

```
8 customers with the most fraudulant transactions (limit 10) ==> record count: 10
+-----------+-----+
|customer_id|count|
+-----------+-----+
|CUST_91730 |801  |
|CUST_24836 |753  |
|CUST_87928 |751  |
|CUST_81009 |748  |
|CUST_77434 |745  |
|CUST_12255 |739  |
|CUST_49715 |733  |
|CUST_88176 |732  |
|CUST_79625 |731  |
|CUST_41245 |724  |
+-----------+-----+
```

Figure 42: Some Customers Who Have Fraudulent Transactions (Spark)

The next step is to prepare data for machine learning. This is done by the prepare_data function which at first splits the timestamp and then converts the boolean columns to integers. Finally, it selects only usable columns and after printing the new schema, it stores the top 1000 rows of the new dataset into a CSV file named processed_data_csv_top1000.

```
@show_elapsed_time
def prepare_data(spark, df, output_path, logger):
    # preparing the data and eliminate unused fields

    # split data-time column (timestamp)
    df_cleaned = df.withColumn("date", regexp_replace(date_format(col("timestamp"), "yyyy-MM-dd"),"-","").cast("integer"))
    df_cleaned = df_cleaned.withColumn("time_hour", date_format(col("timestamp"), "HH").cast("integer"))
    df_cleaned = df_cleaned.withColumn("time_minute", date_format(col("timestamp"), "mm").cast("integer"))
    df_cleaned = df_cleaned.withColumn("time_second", date_format(col("timestamp"), "ss").cast("integer"))

    # change boolean columns to integer
    df_cleaned = df_cleaned.withColumn("card_present", col("card_present").cast("integer"))
    df_cleaned = df_cleaned.withColumn("high_risk_merchant", col("high_risk_merchant").cast("integer"))
    df_cleaned = df_cleaned.withColumn("weekend_transaction", col("weekend_transaction").cast("integer"))
    df_cleaned = df_cleaned.withColumn("is_fraud", col("is_fraud").cast("integer"))

    # select only usable fields
    df_cleaned = df_cleaned.select("customer_id", "card_number", "date", "time_hour", "time_minute", "time_second", "merchant_category", "merchant_type", "merchant", "amoun

    # processed data schema
    msg = "The schema of data after preparation is"
    show_schema(df_cleaned, msg, logger)

    # save first 1000 rows of the cleaned data as a csv file
    df_cleaned.createOrReplaceTempView("transactions")
    result = spark.sql("select * from transactions limit 1000")
    result.coalesce(1).write.mode("overwrite").csv(path.join(output_path,"processed_data_csv_top1000"), header = True)

    return df_cleaned
```

Figure 43: Function for Data Preparation Using Spark

4.3. Map Reduce and Spark Comparison

From the point of ease of use, Spark is much easier to use and offers vast functionalities. It is available to use both Spark dataframe functions or SparkSQL to perform data preparation tasks. From the speed aspect also spark is the leading approach as it takes about 50% less than the map-reduce model to get the results (Figure 44). Since it is much simpler to code using Spark it is much more scalable with less effort.

☐	data_preparation_pyspark	Succeeded	europe-west10	PySpark	bigdata-cluster-euw10	5 Jan 2025, 15:12:57	6 min 28 sec	None
☐	data_preparation_mapreduce	Succeeded	europe-west10	PySpark	bigdata-cluster-euw10	5 Jan 2025, 12:29:29	13 min 53 sec	None

Figure 44: Elapsed Time for Each Job

There is another time-base analysis which is done using @show_elapsed_time decoration on top of each function (Figure 45). These result confirm the previous results about job elapsed time.

```
Map Reduce Elapsed Time:
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>function read_data ended ==> Elapsed time: 51 seconds
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>function check_data_quality ended ==> Elapsed time: 762 seconds
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>function prepare_data ended ==> Elapsed time: 7 seconds

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>function main ended ==> Elapsed time: 824 seconds

Spark Elapsed Time:
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>function read_data ended ==> Elapsed time: 51 seconds
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>function check_data_quality ended ==> Elapsed time: 230 seconds
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>function prepare_data ended ==> Elapsed time: 3 seconds

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>function main ended ==> Elapsed time: 379 seconds
```

Figure 45: Elapsed Time for Each Function

5. Advanced Analytics and Machine Learning

5.1. Model Selection

The problem is a prediction of fraudulent transactions based on the given features. It is obvious that it is a binary classification problem. This study applies different classification models such as logistic regression, support vector machine, and random forest on the dataset and compares the results.

5.2. Machine Learning Process

For machine learning, like other modules, we should open the dataset first. This is done by the open_session function which has a slight difference with the read_data function mentioned previously. It can act in two ways, one working with processed saved dataset and the other with the unprocessed raw dataset. In the second way, it calls the prepare_data function from the dp_spark module and uses it to prepare the dataset before machine learning (Figure 46).

```
from pyspark.sql import SparkSession
from pyspark.ml.feature import StringIndexer, VectorAssembler
from pyspark.ml.classification import LogisticRegression, RandomForestClassifier, LinearSVC
from pyspark.ml.evaluation import BinaryClassificationEvaluator, MulticlassClassificationEvaluator
from dp_spark import *
from general import *


@show_elapsed_time
def open_session(input_path, output_path, logger : Logger, mode : int = 1):
    # start spark session and load data into a dataframe
    # mode = 1 --> start a spark session and load processed data from a csv file which has been provided before
    # mode = 2 --> call read_data an prepare_data function from dp_spark module and prepare the original data and use it

    if mode == 1:
        # open spark session
        spark = SparkSession.builder \
                .appName("Machine Learning") \
                .config("spark.driver.memory", "4g") \
                .config("spark.executor.memory", "4g") \
                .getOrCreate()

        # read data from csv file
        processed_data = spark.read.csv(input_path, inferSchema = True, header = True)

        # show schema of original data
        msg = "The schema of processed data is"
        show_schema(processed_data, msg, logger)

    elif mode == 2:
        # read data from the input file
        spark, df = read_data(input_path, logger)

        # prepare data for the analysis
        processed_data = prepare_data(spark, df, output_path, logger)

    return spark, processed_data
```

Figure 46: Open Session Function

Afterward, we need to do some preprocessing tasks such as converting categorical values to indexed numbers. This can be done using the StringIndexer class which is part of the PySpark.ml module (Figure 47).

```python
def encode_categorical_columns(processed_data):
    # encode categorical columns and change them to indexed numbers
    categorical_columns = [
        ("merchant_category", "merchant_category_index"),
        ("merchant_type", "merchant_type_index"),
        ("merchant", "merchant_index"),
        ("currency", "currency_index"),
        ("country", "country_index"),
        ("city", "city_index"),
        ("city_size", "city_size_index"),
        ("card_type", "card_type_index"),
        ("device", "device_index"),
        ("channel", "channel_index"),
    ]

    # in each iteration one categorical column is encoded
    indexed_data = processed_data
    for input_col, output_col in categorical_columns:
        indexer = StringIndexer(inputCol=input_col, outputCol=output_col)
        model = indexer.fit(indexed_data)
        indexed_data = model.transform(indexed_data)

    return indexed_data
```

Figure 47: Encode Categorical Values

Now the dataset is prepared for machine learning. The classification function (Figure 48) creates a VectorAssembler based on the columns needed to participate in model training. It creates a column named features which contains all these columns. At last, it selects only features and is_fraud columns and passes them to the evaluate_model function.

```python
@show_elapsed_time
def classification(processed_data):
    # prepare data for analysis then creare classifications model and train then for evaluation and comparison

    # encode categorical variables
    indexed_data = encode_categorical_columns(processed_data)

    # assemble all feature columns into a single vector named "features"
    assembler = VectorAssembler(inputCols=["date", "time_hour", "time_minute", "time_second", "merchant_category_index", "merchant_type_index", "merchant_index", "amount
    # assembler = VectorAssembler(inputCols=["time_hour", "amount", "card_present", "distance_from_home", "high_risk_merchant", "transaction_hour"], outputCol="features"

    # transform data using assembler
    transformed_data = assembler.transform(indexed_data)

    # select only features and label columns
    final_data = transformed_data.select("features", "is_fraud")

    evaluate_models(final_data)
```

Figure 48: Classification Function

The evaluate_models Function (Figure 49) is responsible for creating logistic regression, support vector machine, and random forest models. It trains these models based on the training set. Afterward, it predicts the output for the test set and evaluates the models based on the results.

```python
def evaluate_models(final_data, logger : Logger):
    # create a list of classifiers and their parameters
    classifiers = [
        LogisticRegression(featuresCol="features", labelCol="is_fraud", predictionCol="predicted_is_fraud"),
        LinearSVC(featuresCol="features", labelCol="is_fraud", predictionCol="predicted_is_fraud"),
        RandomForestClassifier(featuresCol="features", labelCol="is_fraud", predictionCol="predicted_is_fraud", maxBins=200)
    ]

    # split data into train and test sets
    train_data, test_data = final_data.randomSplit(weights=[0.8, 0.2], seed=42)

    for classifier in classifiers:
        # create and train the model for every classifier
        cl_model = classifier.fit(train_data)

        # predict test_data with the model
        pred_data = cl_model.transform(test_data)

        # print the name of classifier
        logger.log(f"{classifier.__class__.__name__} evaluation results","")

        # Area Under ROC metric
        BCE = BinaryClassificationEvaluator(rawPredictionCol="predicted_is_fraud", labelCol="is_fraud", metricName="areaUnderROC")
        areaUnderROC = BCE.evaluate(pred_data)
        logger.log("Area Under ROC", str(areaUnderROC))

        # Area Under PR metric
        BCE = BinaryClassificationEvaluator(rawPredictionCol="predicted_is_fraud", labelCol="is_fraud", metricName="areaUnderPR")
        areaUnderPR = BCE.evaluate(pred_data)
        logger.log("Area Under PR", str(areaUnderPR))

        # Accuracy metric
        MCE = MulticlassClassificationEvaluator(predictionCol="predicted_is_fraud", labelCol="is_fraud", metricName="accuracy")
        accuracy = MCE.evaluate(pred_data)
        logger.log("Accuracy", str(accuracy))

        # Weighted Precision metric
        MCE = MulticlassClassificationEvaluator(predictionCol="predicted_is_fraud", labelCol="is_fraud", metricName="weightedPrecision")
        weightedPrecision = MCE.evaluate(pred_data)
        logger.log("Weighted Precision", str(weightedPrecision))

        # Weighted Recall metric
        MCE = MulticlassClassificationEvaluator(predictionCol="predicted_is_fraud", labelCol="is_fraud", metricName="weightedRecall")
        weightedRecall = MCE.evaluate(pred_data)
        logger.log("Weighted Recall", str(weightedRecall))

        # F1 Score metric
        MCE = MulticlassClassificationEvaluator(predictionCol="predicted_is_fraud", labelCol="is_fraud", metricName="f1")
        f1 = MCE.evaluate(pred_data)
        logger.log("F1 Score", str(f1))
```

Figure 49: Evaluate Models Function

To get the results, we need to submit a PySpark job to the GCP (Figure 50). The results show that random forest creates the best evaluation among all algorithms. It gives about 93 percent of Accuracy, Weighted Precision, Weighted Recall, and F1 Score (Figure 51).

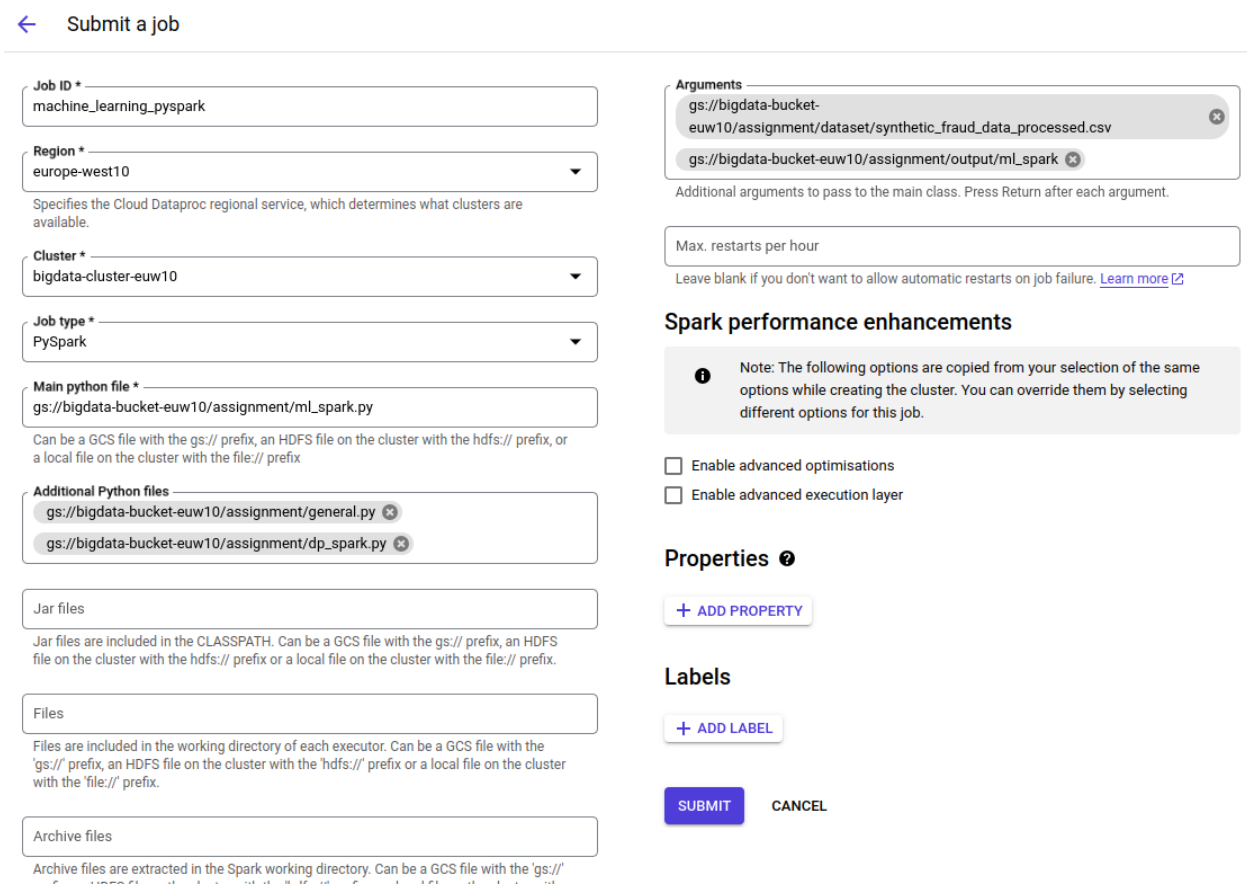

Figure 50: Machine Learning PySpark Job

```
LogisticRegression evaluation results:
Area Under ROC: 0.8620361805558636
Area Under PR: 0.7634962885627985
Accuracy: 0.9232418287026364
Weighted Precision: 0.9213125987588717
Weighted Recall: 0.9232418287026364
F1 Score: 0.9217701025587541

LinearSVC evaluation results:
Area Under ROC: 0.8614393082754632
Area Under PR: 0.7631366418032854
Accuracy: 0.9230480337497252
Weighted Precision: 0.9210937361939824
Weighted Recall: 0.9230480337497251
F1 Score: 0.921546686570235

RandomForestClassifier evaluation results:
Area Under ROC: 0.8399372608032037
Area Under PR: 0.8474684797402858
Accuracy: 0.9327030319554512
Weighted Precision: 0.9351251212318133
Weighted Recall: 0.9327030319554512
F1 Score: 0.928131050523295
```

Figure 51: Classification Evaluation Results

5.3. Visualization

This study provides some visualizations to have a better insight into the dataset.

```
def visualization(df, output_path : str):
    # create and save some visualizations

    fig = plt.figure(figsize=(16, 9), dpi=300)

    # select only needed columns to convert them to pandas df
    df_selected = df.select(["date", "time_hour", "merchant_type", "country", "channel","is_fraud"]).withColumn("year", substring(col(
    df_pandas = df_selected.toPandas()

    # countplot of fraud transaction based on day
    sns.countplot(df_pandas,x="day", hue="is_fraud")
    plt.savefig(fname=f"{output_path}/countplot_day.png", format="png", dpi = fig.dpi)
    plt.clf()
    # countplot of fraud transaction based on hour
    sns.countplot(df_pandas,x="time_hour", hue="is_fraud")
    plt.savefig(fname=f"{output_path}/countplot_hour.png", format="png", dpi = fig.dpi)
    plt.clf()
    # countplot of fraud transaction based on merchant_type
    sns.countplot(df_pandas,x="merchant_type", hue="is_fraud")
    plt.savefig(fname=f"{output_path}/countplot_merchant_type.png", format="png", dpi = fig.dpi)
    plt.clf()
    # countplot of fraud transaction based on country
    sns.countplot(df_pandas,x="country", hue="is_fraud")
    plt.savefig(fname=f"{output_path}/countplot_country.png", format="png", dpi = fig.dpi)
    plt.clf()
    # countplot of fraud transaction based on channel
    sns.countplot(df_pandas,x="channel", hue="is_fraud")
    plt.savefig(fname=f"{output_path}/countplot_channel.png", format="png", dpi = fig.dpi)
    plt.clf()

    # select only needed columns to convert them to pandas dataframe
    df_selected = df.groupBy(["country", "is_fraud"]).agg(sum("amount").alias("total_amount"))
    df_pandas = df_selected.toPandas()
    # scatter plot country and amount
    sns.barplot(df_pandas, x="country", y="total_amount",hue="is_fraud")
    legend = plt.legend()
    legend.set_title("is_fraud")
    legend.set_bbox_to_anchor((1,1))
    plt.savefig(fname=f"{output_path}/barplot_country_amount.png", format="png", dpi = fig.dpi)
    plt.clf()
    # select only needed columns to convert them to pandas df
    df_selected = df.groupBy(["merchant_category", "merchant_type", "is_fraud"]).agg(sum("amount").alias("total_amount"))
    df_pandas = df_selected.toPandas()
    # scatter plot merchant_category and merchant_type
    sns.scatterplot(df_pandas, x="merchant_category", y="merchant_type", hue="is_fraud", size="total_amount", sizes = (200,1000))
    legend = plt.legend()
    legend.set_title("legends")
    legend.set_bbox_to_anchor((1,1))
    plt.grid(visible=True)
    plt.savefig(fname=f"{output_path}/scatter_merchant_category_type_amountsize.png", format="png", dpi = fig.dpi)
    plt.clf()

    # select only needed columns to convert them to pandas df
    df_selected = df.groupBy(["currency", "channel", "is_fraud"]).agg(sum("amount").alias("total_amount"))
    df_pandas = df_selected.toPandas()
    # scatter plot currency and channel
    sns.scatterplot(df_pandas, x="currency", y="channel", hue="is_fraud", size="total_amount", sizes = (200,1000))
    legend = plt.legend()
    legend.set_title("legends")
    legend.set_bbox_to_anchor((1,1))
    plt.savefig(fname=f"{output_path}/scatter_currency_channel_amountsize.png", format="png", dpi = fig.dpi)
    plt.clf()

    # select only needed columns to convert them to pandas df
    df_selected = df.groupBy(["city_size", "card_type", "is_fraud"]).agg(sum("amount").alias("total_amount"))
    df_pandas = df_selected.toPandas()
    # scatter plot currency and channel
    sns.scatterplot(df_pandas, x="city_size", y="card_type", hue="is_fraud", size="total_amount", sizes = (200,1000))
    legend = plt.legend()
    legend.set_title("legends")
    legend.set_bbox_to_anchor((1,1))
    plt.savefig(fname=f"{output_path}/scatter_citysize_cardtype_amountsize.png", format="png", dpi = fig.dpi)
    plt.clf()

    numeric_cols = [col for col, dtype in df.dtypes if dtype in ["int", "bigint", "float", "double", "decimal"]]
    corr = df.select(numeric_cols).toPandas().corr()
    sns.heatmap(corr, cmap="coolwarm", annot=True)
    plt.savefig(fname=f"{output_path}/heatmap_correlation.png", format="png", dpi = fig.dpi)
    plt.clf()
```

Figure 52: Visualization Function

All visualizations are done using the visualization function (Figure 52). At first, we need to convert the PySpark dataframe into a Pandas dataframe (McKinney et al., 2011) and then use some visualization libraries such as Matplotlib (Hunter, 2007) and Seaborn. To do this more efficiently, it is very important to select only usable columns before conversion. There are some of these visualizations below.

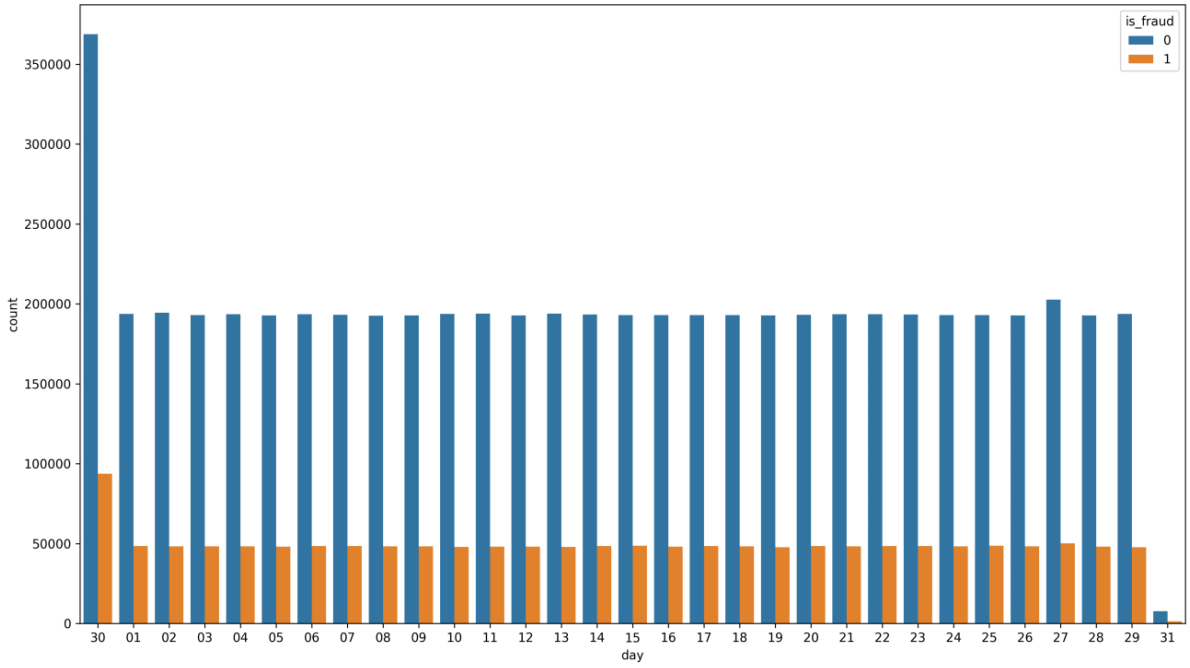

Figure 53: Fraudulent vs Genuine Transaction Count Based on Day

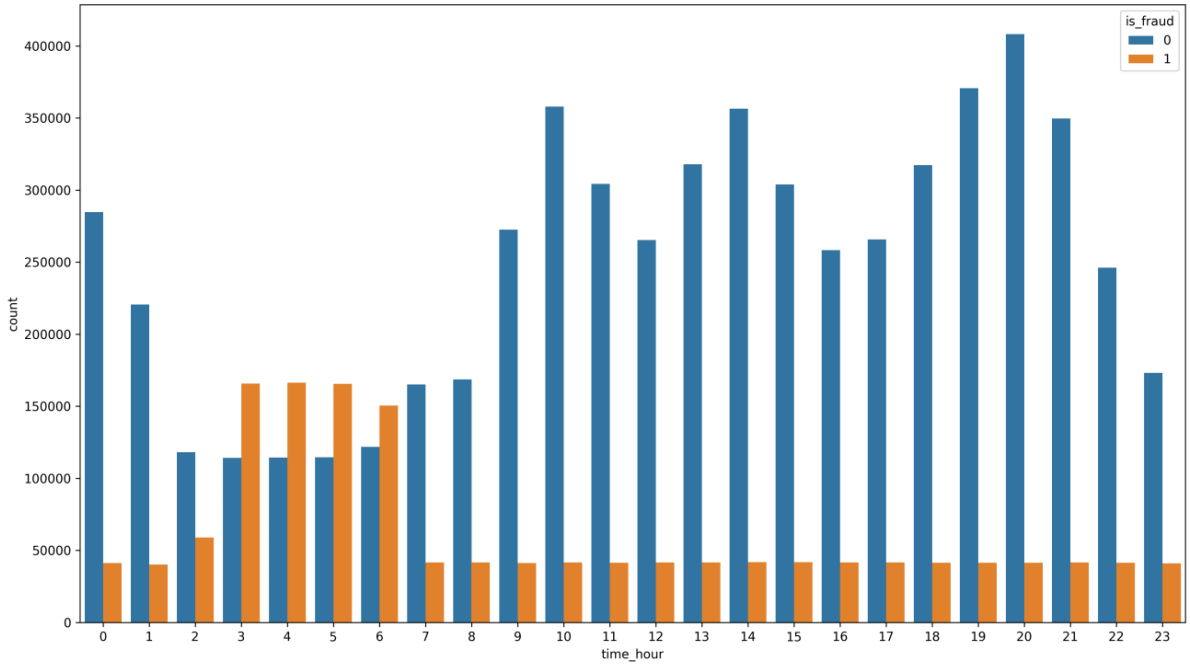

Figure 54: Fraudulent vs Genuine Transaction Count Based on Hour

The first two plots show the number of fraudulent and genuine transactions based on day (Figure 53) and hour (Figure 54). They depict that the 30th of the month has the most transaction number and also the most fraudulent ones. The hour plot shows that fraudulent transaction number goes up significantly between 2 and 6 AM.

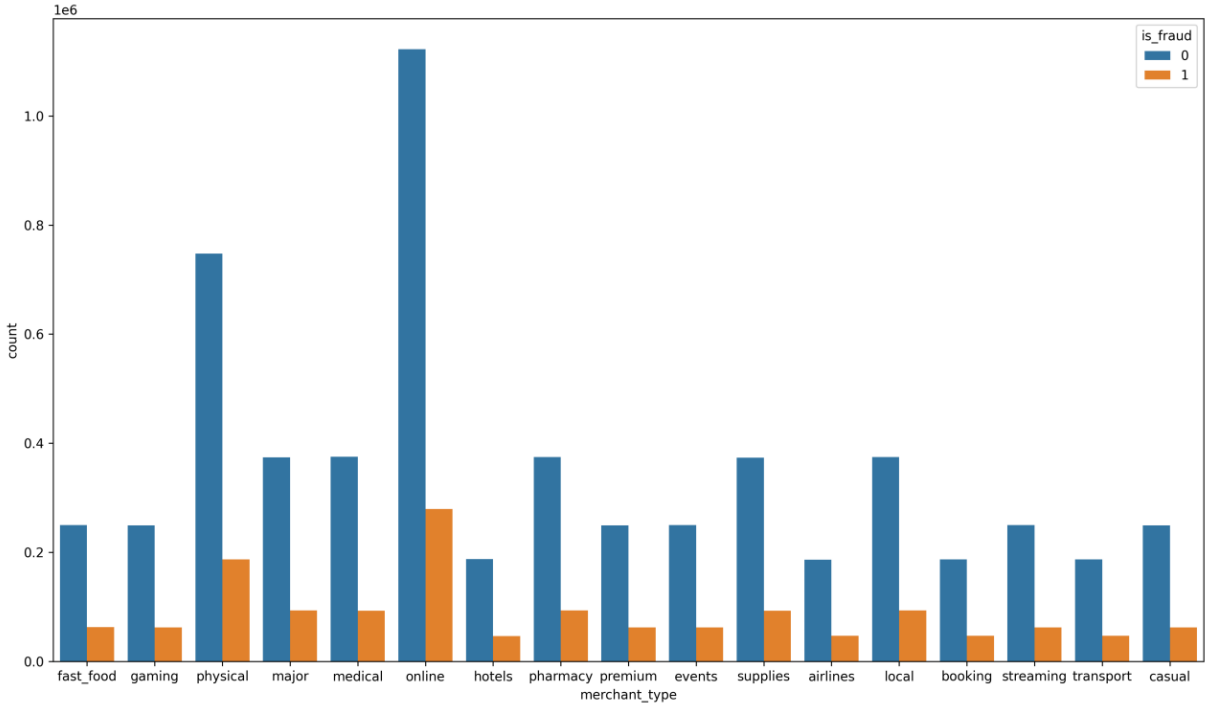

Figure 55: Fraudulent vs Genuine Transaction Count Based on Merchant_type

The next plot is about merchant types and the number of their transactions. It shows that the most fraudulent transactions are related to online and physical merchant types (Figure 55). The other one is country plots that depict Russia, Brazil, Mexico, and Nigeria having the most fraudulent transactions respectively (Figure 56) while Nigeria, Russia, Mexico, and Japan shape the order in the amount of fraudulent transactions (Figure 57). The other plot shows POS channel includes the most fraudulent transactions (Figure 58).

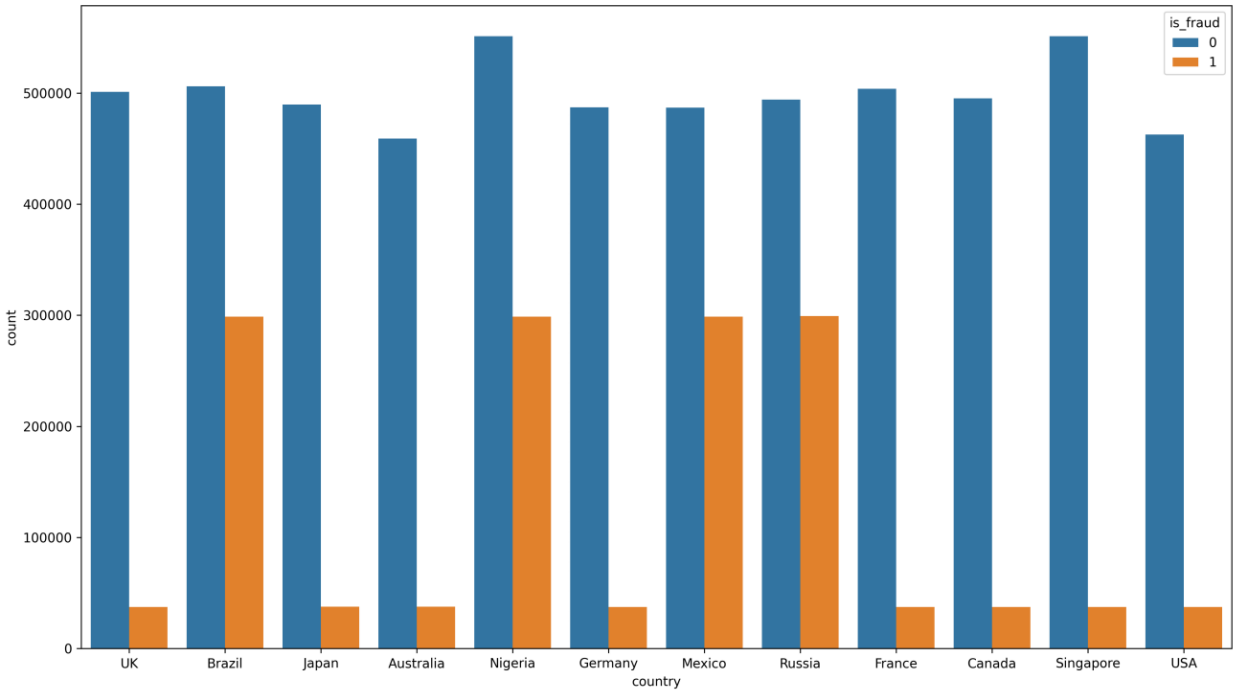

Figure 56: Fraudulent vs Genuine Transaction Count Based on Country

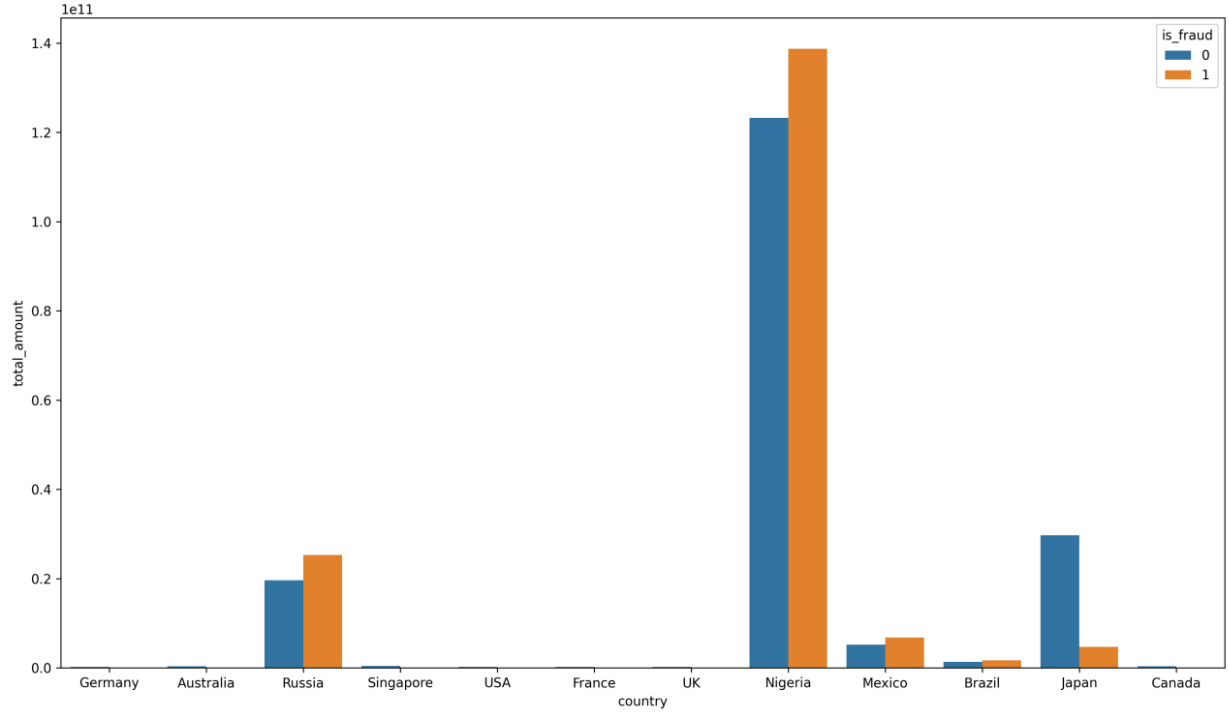

Figure 57: Fraudulent vs Genuine Transaction Amount Based on Country

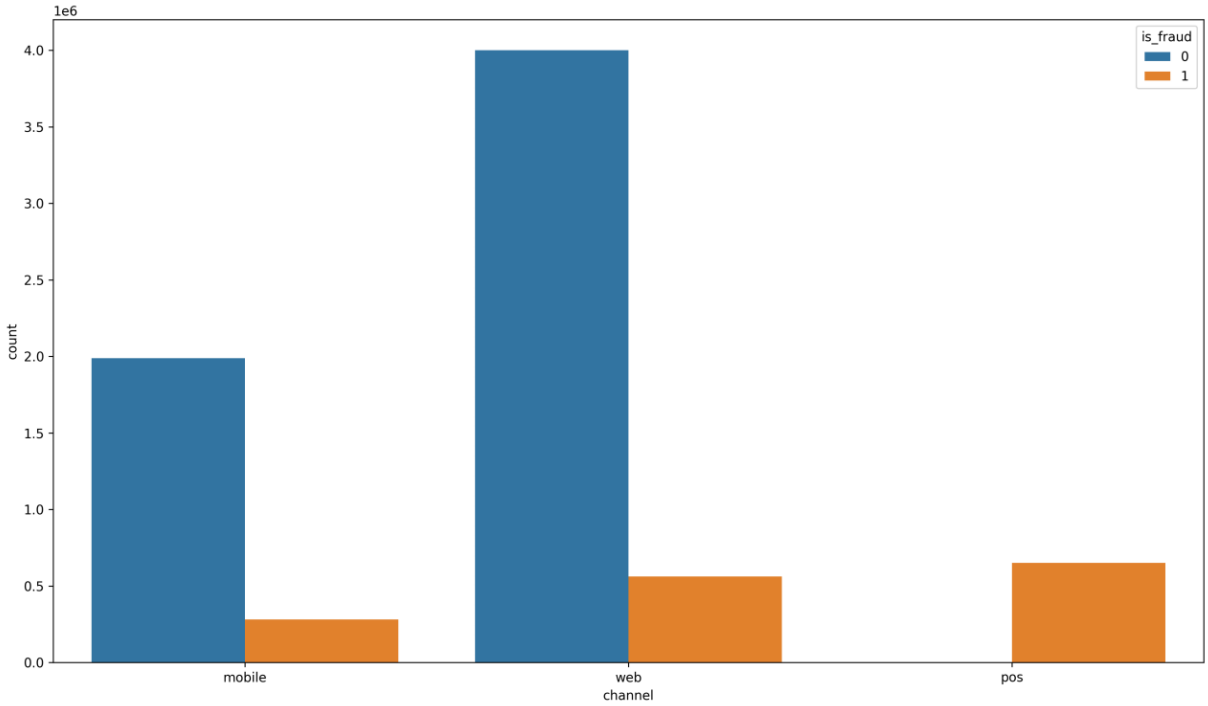

Figure 58: Fraudulent vs Genuine Transaction Count Based on Channel

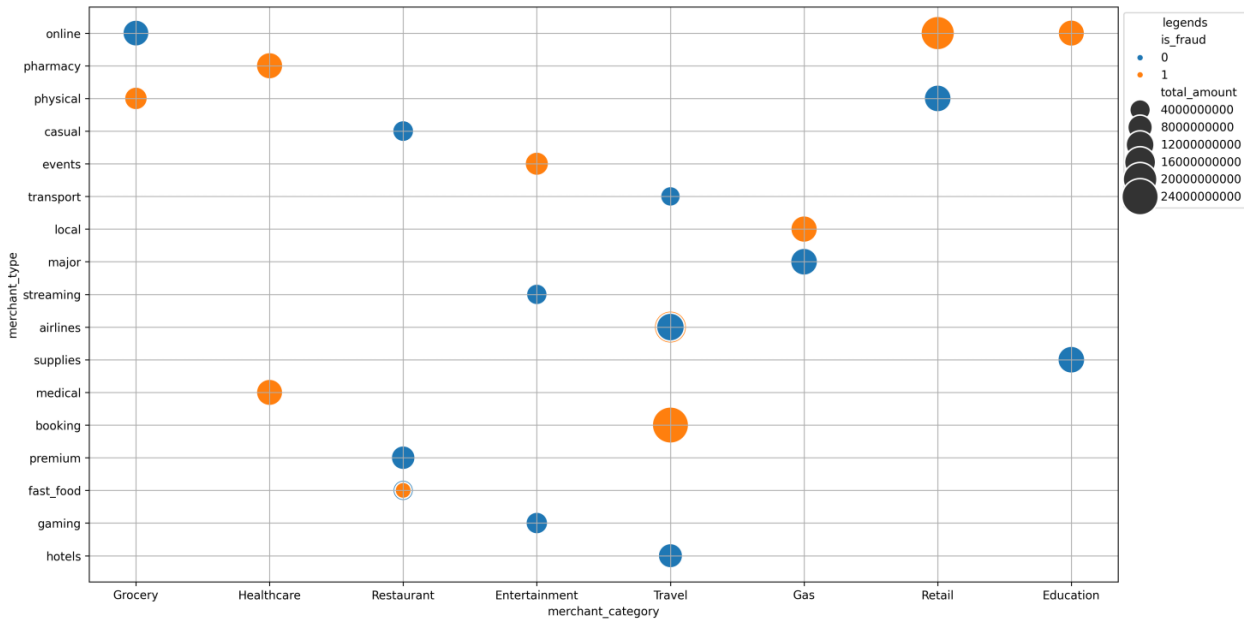

Figure 59: Transaction Amount Based on Merchant_category and Merchant_type

The other visualization shows the relationship between mechant_category and merchant_type to reveal the amount of genuine and fraudulent transactions (Figure 59). This plot states that a combination of (travel, booking), (retail, online), (healthcare, medical, and online) shape the most amount of fraudulent transactions.

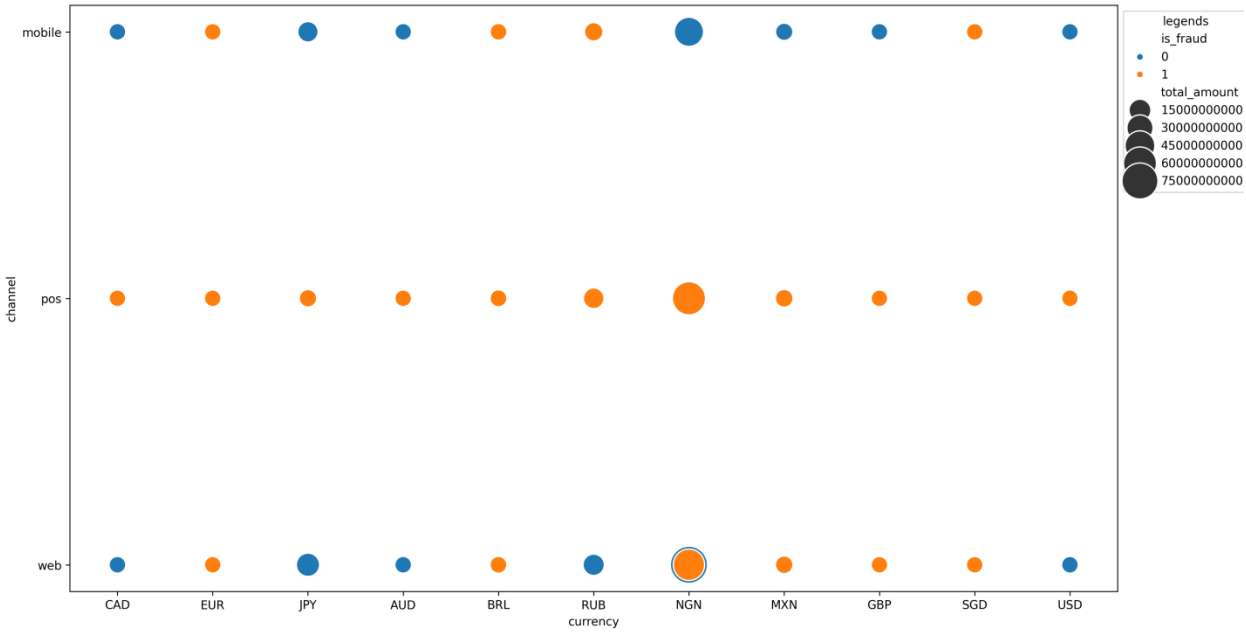

Figure 60: Transaction Amount Based on Channel and Currency

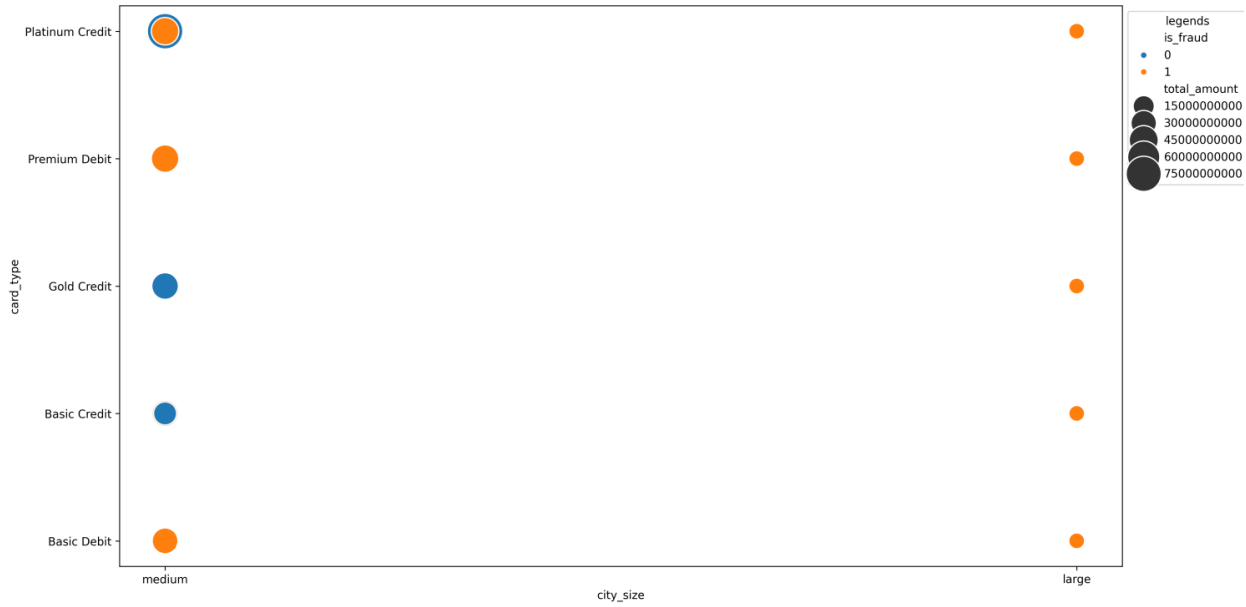

Figure 61: Transaction Amount Based on Card_type and City_size

The other visualizations show in Nigeria most fraud happens via pos and web channels (Figure 60) or medium cities have larger amounts of fraud, especially with premium credit, premium debit, and basic debit cards (Figure 61).

There is also a heat map of the correlation between all columns (Figure 62). It reveals that columns “time_hour“, “amount“, “card_present“, “distance_from_home“, “high_risk_merchant“, “transaction_hour“ have the most effect on the label variable, “isfraud”.

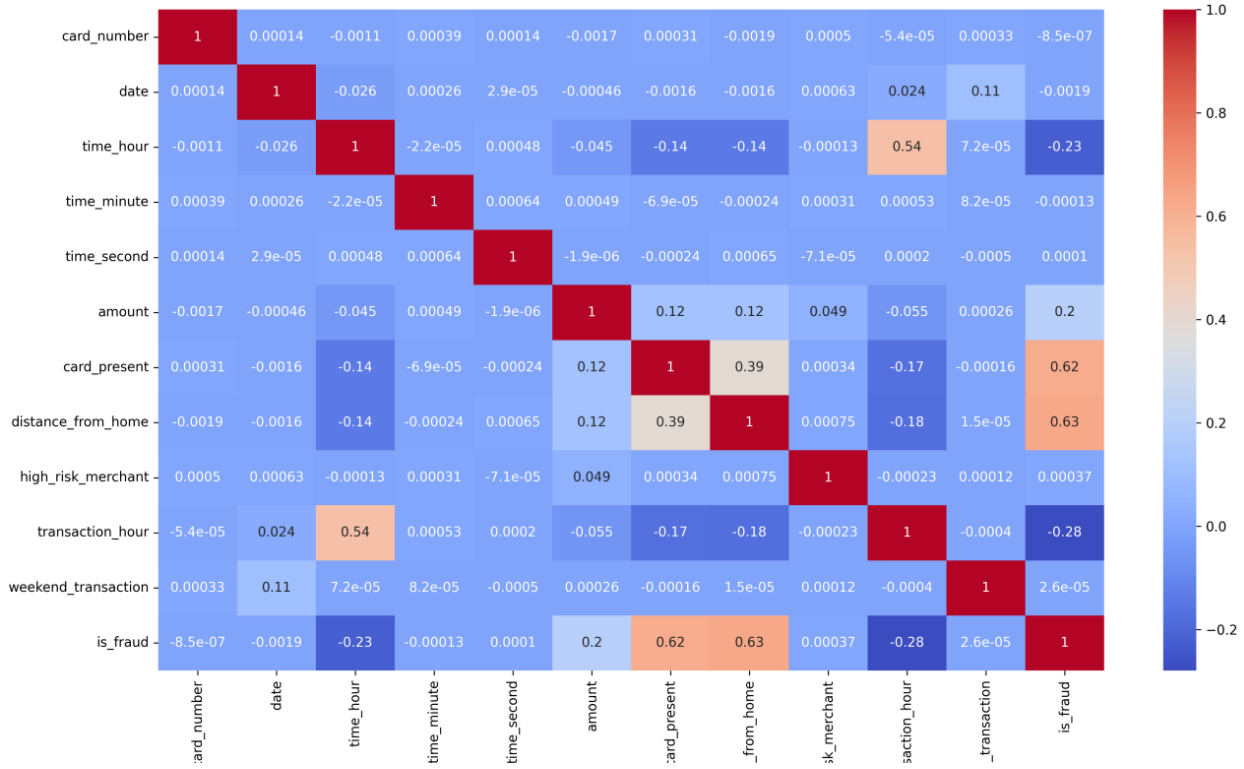

Figure 62: Heat Map of Correlation Between All Columns

5.4. Feature Selection

After training the models based on all usable features, it is time to select features that have more impact on the label column. This will reduce the time and computation cost for training the models since does not affect the accuracy of the predictions very much.

This study select the features based on their correlation with the label column (is_fraud). Higher correlations in both directions (positive affects positive class and negative affects negative class) show more impact on the label columns. Based on this fact, features "time_hour", "amount", "card_present", "distance_from_home", "high_risk_merchant", and "transaction_hour" are selected to participate in the model training operation. The results of the model evaluation are shown below (Figure 63). As it is shown, there is no big change in results after feature selection

but there is a huge impact on the execution time of the jobs (Figure 64). Time reduces from 1006 seconds to 709 seconds which shows 30% less execution time.

```
LogisticRegression evaluation results:
Area Under ROC: 0.855174090338115
Area Under PR: 0.7691546494681379
Accuracy: 0.9230988213925571
Weighted Precision: 0.9209048916028912
Weighted Recall: 0.9230988213925571
F1 Score: 0.9210310597851209

LinearSVC evaluation results:
Area Under ROC: 0.8547234135368856
Area Under PR: 0.7327885719048391
Accuracy: 0.9150630134313268
Weighted Precision: 0.913324770773318
Weighted Recall: 0.9150630134313268
F1 Score: 0.9139629353951457

RandomForestClassifier evaluation results:
Area Under ROC: 0.8537173637323341
Area Under PR: 0.8158709483137621
Accuracy: 0.9314467060538202
Weighted Precision: 0.930606276391742
Weighted Recall: 0.9314467060538202
F1 Score: 0.9283374150247412
```

Figure 63: Model Evaluation Results After Feature Selection

Job ID	Status	Region	Type	Cluster	Start time	Elapsed time	Labels
job-aeee69cc	Succeeded	europe-west10	PySpark	bigdata-cluster-euw10	15 Jan 2025, 11:30:52	11 min 49 sec	None
job-1b571536	Succeeded	europe-west10	PySpark	bigdata-cluster-euw10	5 Jan 2025, 16:19:50	16 min 46 sec	None

Figure 64: Jobs Execution Time Before and After Feature Selection

6. CONCLUSION

This study discusses the role of big data analytics in fraud detection. The first finding in this study is that GCP is a better choice for students based on its free tier although the billing process is unclear. GCP provides a Dataproc cluster to execute PySpark Jobs in a very smooth and promising process. This paper also shows that Spark is much easier and more efficient than the Hadoop Map Reduce programming model for data processing. It offers a vast range of modules including SparkSQL and ML to handle not only data preparation and analysis but also building machine learning models based on the data. It reduces the execution time by 53% compared to the Map Reduce model. This study applies three classification models including logistic regression, support vector machine, and random forest to the dataset. It shows random forest has the best results among them. It creates an F1-score of 93% in comparison with other models with an F1-score of 92%. In the end, feature selection, using the correlation between features and label columns, shows that we can gain similar results with fewer features. It creates 93% of the F1-score with 30% less execution time.

BIBLIOGRAPHY

ACFE (2022) *Report to the nations: 2022 global study on occupational fraud and abuse*. Austin, TX: Association of Certified Fraud Examiners. Available at: https://www.acfe.com (Accessed: 15 January 2025).

Ahmadi, S. (2024) 'A comprehensive study on integration of big data and AI in the financial industry and its effect on present and future opportunities', *International Journal of Current Science Research and Review*, 7(1), pp. 66–74. doi:10.47191/ijcsrr/V7-i1-07.

Bin Sulaiman, R., Schetinin, V. and Sant, P. (2022) 'Review of machine learning approach on credit card fraud detection', *Human-Centric Intelligent Systems*, 2(1), pp. 55–68.

Hunter, J.D. (2007) 'Matplotlib: A 2D graphics environment', *Computing in Science & Engineering*, 9(3), pp. 90–95.

McKinney, W., et al. (2011) *pandas: a foundational Python library for data analysis and statistics*.

Ngai, E.W.T., Hu, Y., Wong, Y.H., Chen, Y. and Sun, X. (2011) 'The application of data mining techniques in financial fraud detection: A classification framework and an academic review of literature', *Decision Support Systems*, 50(3), pp. 559–569. doi:10.1016/j.dss.2010.08.006.

Seaborn (2025) *Statistical data visualization.* Available at: https://seaborn.pydata.org/ (Accessed: 17 January 2025).

Samadov, I. (2024) *Transactions: Explore realistic patterns in financial transactions for fraud detection*. Kaggle. Available at: https://www.kaggle.com/datasets/ismetsemedov/transactions/data (Accessed: 12 December 2024).

Shabbir, M.Q. and Gardezi, S.B.W. (2020) 'Application of big data analytics and organizational performance: the mediating role of knowledge management practices', *Journal of Big Data*, 7(1), p. 47.

Udeh, E., Amajuoyi, P., Adeusi, K. and Scott, A. (2024) 'The role of big data in detecting and preventing financial fraud in digital transactions', *World Journal of Advanced Research and Reviews*, 22(2), pp. 1746–1760. doi:10.30574/wjarr.2024.22.2.1575.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M., Shenker, S. and Stoica, I. (2012) 'Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing', *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, pp. 2–2.

TABLE OF FIGURES