

(<https://databricks.com>)

LOAD THE DATASET

```
# Load the dataset
df1 = spark.read.format("csv").option("header", "true").load("dbfs:/FileStore/shared_uploads/vorsu@gmu.edu/fraudTest-2.csv")

# Show the first few rows of the DataFrame
df1.display()
```

Table							
	cc0	trans_date_trans_time	cc_num	merchant	category	amt	
1	0	21-06-2020 12:14	2.29116E+15	fraud_Kirlin and Sons	personal_care	2.86	
2	1	21-06-2020 12:14	3.57303E+15	fraud_Sporer-Keebler	personal_care	29.84	
3	2	21-06-2020 12:14	3.59822E+15	fraud_Swaniawski, Nitzsche and Welch	health_fitness	41.28	
4	3	21-06-2020 12:15	3.59192E+15	fraud_Haley Group	misc_pos	60.05	
5	4	21-06-2020 12:15	3.52683E+15	fraud_Johnston-Casper	travel	3.19	
6	5	21-06-2020 12:15	3.04077E+13	fraud_Daugherty LLC	kids_pets	19.55	
7	6	21-06-2020 12:15	2.13181E+14	fraud_Romaquera Ltd	health_fitness	133.93	

6,787 rows | Truncated data

```
# Print the schema to understand the data types
df1.printSchema()

# Display summary statistics
df1.describe().show()
```

```
root
|-- _c0: string (nullable = true)
|-- trans_date_trans_time: date (nullable = true)
|-- cc_num: string (nullable = true)
|-- merchant: string (nullable = true)
|-- category: string (nullable = true)
|-- amt: double (nullable = true)
|-- first: string (nullable = true)
|-- last: string (nullable = true)
|-- gender: string (nullable = true)
|-- street: string (nullable = true)
|-- city: string (nullable = true)
|-- state: string (nullable = true)
|-- zip: integer (nullable = true)
|-- lat: double (nullable = true)
|-- long: double (nullable = true)
|-- city_pop: integer (nullable = true)
|-- job: string (nullable = true)
|-- dob: date (nullable = true)
|-- trans_num: string (nullable = true)
|-- unix_time: integer (nullable = true)
```

```
from pyspark.sql.functions import col, count

# Count missing values in each column
df1.select([count(when(col(c).isNull(), c)).alias(c) for c in df1.columns]).show()
```

[illegible]

VISUALISATIONS

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, to_date, year, month

# Start Spark session
spark = SparkSession.builder.appName("DataVisualizations").getOrCreate()

# Convert transaction date time to a more usable format
df1 = df1.withColumn("trans_date", to_date(col("trans_date_trans_time"), "yyyy-MM-dd HH:mm:ss"))
```

FRAUD PERCENTAGE VS TRANSACTION CATEGORY

```
from pyspark.sql.functions import count, sum, when

# Group by the transaction category and calculate the necessary statistics
fraud_stats = df1.groupBy("category").agg(
    count("*").alias("total_transactions"),
    sum("is_fraud").alias("fraudulent_transactions")
)

# Calculate the fraud percentage
fraud_percentage = fraud_stats.withColumn(
    "fraud_percentage",
    (col("fraudulent_transactions") / col("total_transactions")) * 100
)
display(fraud_percentage)
```

Table	Visualization 1				
	category ▲	total_transactions ▲	fraudulent_transactions ▲	fraud_percentage ▲	
1	travel	17449	40	0.22923949796549944	
2	misc_net	27367	267	0.975627580662842	
3	grocery_pos	52553	485	0.9228778566399636	
4	kids_pets	48692	65	0.1334921547687505	
5	shopping_net	41779	506	1.2111347806314177	
6	grocery_net	19426	41	0.21105734582518274	
7	food_dining	39268	54	0.13751655291840686	

14 rows

AVERAGE TRANSACTION AMOUNT VS TRANSACTION TYPE

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

# Ensure the data types are correct
df1 = df1.withColumn("is_fraud", col("is_fraud").cast("integer"))
df1 = df1.withColumn("amt", col("amt").cast("float"))
display(df1.limit(5))
```

Table							
	cc0 ▲	trans_date_trans_time ▲	cc_num ▲	merchant ▲	category ▲	amt ▲	fir
1	0	21-06-2020 12:14	2.29116E+15	fraud_Kirlin and Sons	personal_care	2.86	Jet

2	1	21-06-2020 12:14	3.57303E+15	fraud_Soorer-Keebler	personal_care	29.84	Jo
3	2	21-06-2020 12:14	3.59822E+15	fraud_Swaniawski, Nitzsche and Welch	health_fitness	41.28	As
4	3	21-06-2020 12:15	3.59192E+15	fraud_Haley Group	misc_pos	60.05	Bri
5	4	21-06-2020 12:15	3.52683E+15	fraud_Johnston-Casper	travel	3.19	Næ

5 rows

```
from pyspark.sql.functions import when, col, avg, count, sum

# Map 'is_fraud' to 'Non-Fraudulent' or 'Fraudulent'
df1_labeled = df1.withColumn(
    "is_fraud_label",
    when(col("is_fraud") == 1, "Fraudulent").otherwise("Non-Fraudulent")
)

# Group by the new label and calculate the average, total, and count of transactions
fraud_amount_stats = df1_labeled.groupBy("is_fraud_label").agg(
    count("*").alias("number_of_transactions"),
    sum("amt").alias("total_amount"),
    avg("amt").alias("average_amount")
)

# Display the results
fraud_amount_stats.display()
```

Table	Visualization 1			
	is_fraud_label ▲	number_of_transactions ▲	total_amount ▲	average_amount ▲
1	Non-Fraudulent	553574	37429578.42305112	67.61440823277668
2	Fraudulent	2145	1133324.6799902916	528.3564941679681

2 rows

Number of Credit Card Transactions By Gender

```
from pyspark.sql.functions import count
from pyspark.sql.functions import col, count

# Filter DataFrame for only fraudulent transactions
fraud_transactions = df1.filter(col("is_fraud") == 1)

# Group by gender and count fraudulent transactions
gender_fraud_transactions = fraud_transactions.groupBy("gender").agg(count("*").alias("fraud_transaction_count"))

# Display the fraudulent transaction count by gender
gender_fraud_transactions.display() # or display(gender_fraud_transactions) for Jupyter Notebooks
```

Table	Visualization 1	
	gender ▲	fraud_transaction_count ▲
1	F	1164
2	M	981

2 rows

Number of Fraud Transactions by Job Category

```

fraud_df = df1.filter(col("is_fraud") == 1)

# Group by job title and count the number of fraudulent transactions
fraud_by_job = fraud_df.groupBy("job").count()

# Order by the count of fraudulent transactions in descending order and take the top 10
top10_fraud_jobs = fraud_by_job.orderBy(col("count").desc()).limit(10)

# Display the bar chart for the top 10 jobs
display(top10_fraud_jobs)

```

Table	Visualization 1		
	job	count	
1	Science writer	30	
2	Systems developer	29	
3	Licensed conveyancer	29	
4	Engineer, biomedical	28	
5	Therapist, occupational	27	
6	Colour technologist	27	
7	Counsellor	26	

10 rows

Fraud Distribution

```

# Map 'is_fraud' to 'Non-Fraudulent' or 'Fraudulent'
df1_labeled = df1.withColumn(
    "is_fraud_label",
    when(col("is_fraud") == 1, "Fraudulent").otherwise("Non-Fraudulent")
)

# Group by the 'is_fraud' column and count each group
fraud_counts = df1_labeled.groupBy("is_fraud_label").count()

# Visualize the counts as a bar chart
display(fraud_counts)

```

Table	Visualization 1		
	is_fraud_label	count	
1	Non-Fraudulent	553574	
2	Fraudulent	2145	

2 rows

Number of Transactions

```

# Visualize transactions by state
display(df1.groupBy("state").count().orderBy("count", ascending=False))

```

Table	Visualization 1	Visualization 2
	state	count
1	TX	40393
2	NY	35918
3	PA	34326
4	CA	24135
5	OH	20147
6	MI	19671

50 rows

```
import pandas as pd
import plotly.express as px
from pyspark.sql.functions import col, count

# Group by state and count occurrences
state_count = df1.groupBy("state").agg(count("*").alias("state_count"))

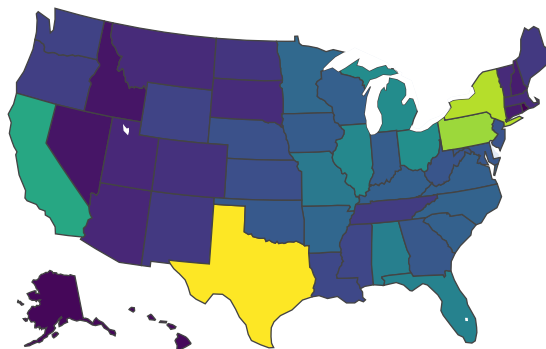
# Convert to Pandas DataFrame for easier visualization
state_count_pd = state_count.toPandas()

# Create a US heatmap with Plotly, with custom color scale labels
fig = px.choropleth(
    state_count_pd,
    locations='state', # State column in your dataset
    locationmode='USA-states', # Specify it's US states
    color='state_count', # Column with counts
    scope='usa', # Limit scope to the US
    color_continuous_scale='Viridis', # Default color scale
    title='Heatmap of All Transactions by State',
    labels={'state_count': 'Transaction Count'}, # Label for the color scale
)

# Customize the colorbar
fig.update_layout(
    coloraxis_colorbar=dict(
        title='Transaction Count', # Title of the color scale
        tickvals=[state_count_pd['state_count'].min(), state_count_pd['state_count'].max()], # Custom ticks
        ticktext=['Low', 'High'], # Custom tick labels
    )
)

# Display the heatmap
fig.show()
```

Heatmap of All Transactions by State



Number of Fraudulent Transactions

```
# Filter the DataFrame for only fraudulent transactions
fraudulent_df = df1.filter(df1.is_fraud == 1)

# Group by state and count the number of fraudulent transactions
fraud_by_state = fraudulent_df.groupBy("state").count()

# Display the results, ordered by count in descending order
fraud_by_state_sorted = fraud_by_state.orderBy("count", ascending=False)

# Display the summary of fraudulent transactions by state
display(fraud_by_state_sorted)
```

Table	Visualization 1	Visualization 2
	state ▲	count ▲
1	NY	175
2	PA	114
3	TX	113
4	CA	76
5	IL	76
6	IN	75
7	VA	75

45 rows

```
from pyspark.sql.functions import count

# Filtering the DataFrame to only include rows where 'is_fraud' is 1
df1_filtered = df1.filter(col('is_fraud') == 1)

# Group by state and count occurrences of fraud
state_fraud_count = df1_filtered.groupBy("state").agg(count("*").alias("state_fraud_count"))

# Convert to Pandas DataFrame for easier visualization
state_fraud_count_pd = state_fraud_count.toPandas()

state_fraud_count_pd.head() # Display the top rows of the Pandas DataFrame
```

	state	state_fraud_count
0	SC	43
1	LA	22
2	MN	73
3	NJ	43
4	KY	67

```

from pyspark.sql.functions import count
import plotly.express as px

# Ensure the DataFrame has the 'is_fraud' column and filter for fraud
if 'is_fraud' in df1.columns:
    # Filtering the DataFrame to only include rows where 'is_fraud' is 1
    df1_filtered = df1.filter(col('is_fraud') == 1)

    # Group by state and count occurrences of fraud
    state_fraud_count = df1_filtered.groupBy("state").agg(count("*").alias("state_fraud_count"))

    # Convert to Pandas DataFrame for easier visualization
    state_fraud_count_pd = state_fraud_count.toPandas()

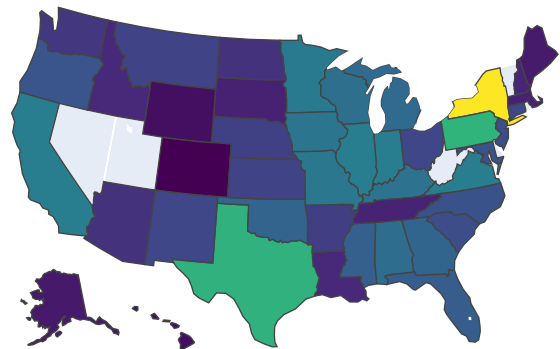
    # Create a US heatmap with Plotly
    fig = px.choropleth(
        state_fraud_count_pd,
        locations='state', # State column in your dataset
        locationmode='USA-states', # Specify it's US states
        color='state_fraud_count', # Column with counts
        scope='usa', # Limit scope to the US
        color_continuous_scale='Viridis', # Color scale
        title='Heatmap of Fraudulent Transactions by State',
        labels={'state_fraud_count': 'Fraud Count'}, # Label for the color scale
    )

    # Customize the colorbar
    fig.update_layout(
        coloraxis_colorbar=dict(
            title='Fraud Count', # Title for the colorbar
            tickvals=[state_fraud_count_pd['state_fraud_count'].min(), state_fraud_count_pd['state_fraud_count'].max()], #
Custom ticks
            ticktext=['Low', 'High'], # Labels for the ticks
        )
    )

    # Display the heatmap
    fig.show()
else:
    raise KeyError("The DataFrame does not have the 'is_fraud' column.")

```

Heatmap of Fraudulent Transactions by State



```
pip install folium
```

Note: you may need to restart the kernel using `dbutils.library.restartPython()` to use updated packages.
Collecting folium

```
Downloading folium-0.16.0-py2.py3-none-any.whl (100 kB)
100.0/100.0 kB 2.6 MB/s eta 0:00:00
Collecting branca>=0.6.0
  Downloading branca-0.7.1-py3-none-any.whl (25 kB)
Requirement already satisfied: numpy in /databricks/python3/lib/python3.10/site-packages (from folium) (1.23.5)
Collecting xyzservices
  Downloading xyzservices-2024.4.0-py3-none-any.whl (81 kB)
82.0/82.0 kB 10.3 MB/s eta 0:00:00
Requirement already satisfied: requests in /databricks/python3/lib/python3.10/site-packages (from folium) (2.28.1)
Requirement already satisfied: Jinja2>=2.9 in /databricks/python3/lib/python3.10/site-packages (from folium) (3.1.2)
Requirement already satisfied: MarkupSafe>=2.0 in /databricks/python3/lib/python3.10/site-packages (from Jinja2>=2.9->folium) (2.1.1)
Requirement already satisfied: charset-normalizer<3,>=2 in /databricks/python3/lib/python3.10/site-packages (from requests->folium) (2.0.4)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /databricks/python3/lib/python3.10/site-packages (from requests->folium) (1.26.14)
Requirement already satisfied: certifi>=2017.4.17 in /databricks/python3/lib/python3.10/site-packages (from requests->folium) (2022.12.7)
```

GAZE HEATMAP OF FRAUD TRANSACTIONS

```
from pyspark.sql.functions import col, split
import folium
from folium.plugins import HeatMap
from pyspark.sql import SparkSession

# Initialize Spark Session
spark = SparkSession.builder.appName("Fraud Analysis").getOrCreate()
# Assuming that 'merch_long' and 'merch_lat' are the fields for longitude and latitude
# Ensuring they are of the correct data type (float)
df1 = df1.withColumn('merch_lat', col('merch_lat').cast('float'))
df1 = df1.withColumn('merch_long', col('merch_long').cast('float'))

# Filter for fraudulent transactions
fraud_df1 = df1.filter(df1.is_fraud == 1)

# Convert Spark DataFrame to Pandas DataFrame for visualization
fraud_pd = fraud_df1.toPandas()


# Generate a base map focused on the USA
usa_center_lat = 37.0902
usa_center_long = -95.7129
map = folium.Map(location=[usa_center_lat, usa_center_long], zoom_start=5)

# Add heat map layer on the base map
HeatMap(data=fraud_pd[['merch_lat', 'merch_long']], radius=10).add_to(map)

map
```




usted to load map: File -> Trust Notebook

 Leaflet (<https://leafletjs.com>) | © OpenStreetMap (<https://www.openstreetmap.org/copyright>) contributors

Machine Learning Models

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, explode, array, lit, count, when, radians, sin, cos, atan2, sqrt
from pyspark.sql.types import IntegerType, DoubleType, DateType
from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler, StringIndexer, OneHotEncoder
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
```

```
Spark = SparkSession.builder.appName("Machine Learning with Spark").getOrCreate()
```

Data Processing

```
df1.printSchema()
```

```

root
|-- _c0: string (nullable = true)
|-- trans_date_trans_time: string (nullable = true)
|-- cc_num: string (nullable = true)
|-- merchant: string (nullable = true)
|-- category: string (nullable = true)
|-- amt: string (nullable = true)
|-- first: string (nullable = true)
|-- last: string (nullable = true)
|-- gender: string (nullable = true)
|-- street: string (nullable = true)
|-- city: string (nullable = true)
|-- state: string (nullable = true)
|-- zip: string (nullable = true)
|-- lat: string (nullable = true)
|-- long: string (nullable = true)
|-- city_pop: string (nullable = true)
|-- job: string (nullable = true)
|-- dob: string (nullable = true)
|-- trans_num: string (nullable = true)
|-- unix time: string (nullable = true)

```

```

# Count missing values in each column
df1.select([count(when(col(c).isNull(), c)).alias(c) for c in df1.columns]).show()

```

Table									
	_c0	trans_date_trans_time	cc_num	merchant	category	amt	first	last	
1	0	555719	0	0	0	0	0	0	

1 row

```

# Cast numeric and date columns early
df1 = df1.withColumn("amt", col("amt").cast(DoubleType())) \
    .withColumn("lat", col("lat").cast(DoubleType())) \
    .withColumn("long", col("long").cast(DoubleType())) \
    .withColumn("zip", col("zip").cast(IntegerType())) \
    .withColumn("city_pop", col("city_pop").cast(IntegerType())) \
    .withColumn("unix_time", col("unix_time").cast(IntegerType())) \
    .withColumn("merch_lat", col("merch_lat").cast(DoubleType())) \
    .withColumn("merch_long", col("merch_long").cast(DoubleType())) \
    .withColumn("is_fraud", col("is_fraud").cast(IntegerType())) \
    .withColumn("trans_date_trans_time", col("trans_date_trans_time").cast(DateType())) \
    .withColumn("dob", col("dob").cast(DateType()))

```

```

from pyspark.sql.functions import col, radians, sin, cos, atan2, sqrt
# Calculate geographic distance
def calculate_distance(df):
    R = 6371 # Radius of Earth in kilometers
    df = df.withColumn("lat_rad", radians(col("lat")))
    df = df.withColumn("long_rad", radians(col("long")))
    df = df.withColumn("merch_lat_rad", radians(col("merch_lat")))
    df = df.withColumn("merch_long_rad", radians(col("merch_long")))
    dlon = col("merch_long_rad") - col("long_rad")
    dlat = col("merch_lat_rad") - col("lat_rad")
    a = (sin(dlat / 2) ** 2) + cos(col("lat_rad")) * cos(col("merch_lat_rad")) * (sin(dlon / 2) ** 2)
    c = 2 * atan2(sqrt(a), sqrt(1 - a))
    distance = R * c
    return df.withColumn("geo_distance", distance)

```

```

# Calculate geographic distance
df1 = calculate_distance(df1)

```

```
# Splitting the dataset into training and testing sets
train_data, test_data = df1.randomSplit([0.7, 0.3], seed=42)
```

Balancing Class Imbalance with Oversampling in the Data

```
# Perform oversampling on training data only
major_df = train_data.filter(col("is_fraud") == 0)
minor_df = train_data.filter(col("is_fraud") == 1)
ratio = major_df.count() / minor_df.count()
oversampled_minor_df = minor_df.withColumn("dummy", explode(array([lit(x) for x in range(int(ratio))]))).drop("dummy")
balanced_train_data = major_df.unionAll(oversampled_minor_df)
```

Building a Data Preprocessing Pipeline with Spark ML for Feature Engineering

```
from pyspark.sql import SparkSession
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler

# Define string categorical features that need indexing and possibly encoding
categorical_features = ["merchant", "category", "gender", "city", "state", "job"]

# Create StringIndexer stages for each categorical feature
indexers = [StringIndexer(inputCol=col, outputCol=col + "_index", handleInvalid='skip') for col in categorical_features]

# Define which indexed categories should be one-hot encoded
encoder_input_cols = [col + "_index" for col in ["category", "gender", "state"]] # Only encode these categories
encoders = [OneHotEncoder(inputCols=[col], outputCols=[col.replace("_index", "_encoded")]) for col in encoder_input_cols]

# Assemble all feature columns (indexed or encoded) together with numerical features
assembler_inputs = [col + "_encoded" if col in encoder_input_cols else col + "_index" for col in categorical_features]
assembler_inputs += ["amt", "zip", "lat", "long", "city_pop", "unix_time", "merch_lat", "merch_long", "geo_distance"] #
Add numerical features and the new geo_distance
assembler = VectorAssembler(inputCols=assembler_inputs, outputCol="features")

# Verify the inputs to the VectorAssembler are correct
print("Assembler Inputs: ", assembler_inputs)

pipeline = Pipeline(stages=indexers + encoders + [assembler])
```

Assembler Inputs: ['merchant_index', 'category_index', 'gender_index', 'city_index', 'state_index', 'job_index', 'amt', 'zip', 'lat', 'long', 'city_pop', 'unix_time', 'merch_lat', 'merch_long', 'geo_distance']

Random Forest Classifier with Custom MaxBins: Pipeline, Training, and Evaluation

```

# Adjusting maxBins parameter
rf = RandomForestClassifier(featuresCol="features", labelCol="is_fraud", numTrees=10, maxBins=850)

# Create a pipeline
pipeline_rf = Pipeline(stages=indexers + encoders + [assembler, rf])

# Train the model
model_rf = pipeline_rf.fit(balanced_train_data)

# Make predictions
predictions_rf = model_rf.transform(test_data)

# Area under PR curve and ROC curve
bce_pr = BinaryClassificationEvaluator(labelCol="is_fraud", metricName='areaUnderPR')
bce_roc = BinaryClassificationEvaluator(labelCol="is_fraud", metricName="areaUnderROC")

# Print evaluations
print("Random Forest PR:", bce_pr.evaluate(predictions_rf))
print("Random Forest ROC:", bce_roc.evaluate(predictions_rf))

```

Random Forest PR: 0.4335124296299378
Random Forest ROC: 0.9755840273515113

Cross-Validation for Random Forest: Parameter Grid, Model Training, and Performance Evaluation

```

#Cross Validation of Random Forest
# Define parameter grid for RandomForest
paramGrid = ParamGridBuilder() \
    .addGrid(rf.numTrees, [10, 20]) \
    .addGrid(rf.maxDepth, [5, 10]) \
    .build()

# Set up cross-validator
crossval_rf = CrossValidator(estimator=pipeline_rf,
                             estimatorParamMaps=paramGrid,
                             evaluator=BinaryClassificationEvaluator(labelCol="is_fraud"),
                             numFolds=3)

# Fit model using cross-validation
cvModel_rf = crossval_rf.fit(balanced_train_data)

# Make predictions on test data
predictions_rf_cv = cvModel_rf.transform(test_data)

# Evaluate the best model
print("Best Random Forest PR:", bce_pr.evaluate(predictions_rf_cv))
print("Best Random Forest ROC:", bce_roc.evaluate(predictions_rf_cv))

```

Best Random Forest PR: 0.5548698793801452
Best Random Forest ROC: 0.9716189084437228

Comparing Logistic Regression Models with Different Regularization Techniques: Default, Lasso, and Ridge

```

from pyspark.ml.classification import LogisticRegression
from pyspark.ml import feature, classification
from pyspark.ml.evaluation import BinaryClassificationEvaluator, MulticlassClassificationEvaluator

# default parameters, regParam = 0.0, elasticNetParam = 0.0
lr = LogisticRegression(featuresCol="features", labelCol="is_fraud")
pipeline_lr = Pipeline(stages=indexers + encoders + [assembler, lr])
model_lr = pipeline_lr.fit(balanced_train_data)
predictions_lr = model_lr.transform(test_data)

# Lasso (L1) Regularization, regParam = 0.5, elasticNetParam = 1.0
lr_lasso = LogisticRegression(featuresCol="features", labelCol="is_fraud", regParam=0.5, elasticNetParam=1.0)
pipeline_lr_lasso = Pipeline(stages=indexers + encoders + [assembler, lr_lasso])
model_lr_lasso = pipeline_lr_lasso.fit(balanced_train_data)
predictions_lr_lasso = model_lr_lasso.transform(test_data)

# Ridge (L2) Regularization, regParam = 0.5, elasticNetParam = 0.0
lr_ridge = LogisticRegression(featuresCol="features", labelCol="is_fraud", regParam=0.5, elasticNetParam=0.0)
pipeline_lr_ridge = Pipeline(stages=indexers + encoders + [assembler, lr_ridge])
model_lr_ridge = pipeline_lr_ridge.fit(balanced_train_data)
predictions_lr_ridge = model_lr_ridge.transform(test_data)

# Evaluators for PR, ROC, and other metrics
bce_pr = BinaryClassificationEvaluator(labelCol="is_fraud", metricName='areaUnderPR')
bce_roc = BinaryClassificationEvaluator(labelCol="is_fraud", metricName="areaUnderROC")

# Evaluate models and print the PR and ROC metrics
print("Default Logistic Regression PR:", bce_pr.evaluate(predictions_lr))
print("Default Logistic Regression ROC:", bce_roc.evaluate(predictions_lr))
print("Lasso Logistic Regression PR:", bce_pr.evaluate(predictions_lr_lasso))
print("Lasso Logistic Regression ROC:", bce_roc.evaluate(predictions_lr_lasso))
print("Ridge Logistic Regression PR:", bce_pr.evaluate(predictions_lr_ridge))
print("Ridge Logistic Regression ROC:", bce_roc.evaluate(predictions_lr_ridge))

```

```

Default Logistic Regression PR: 0.17992188257478633
Default Logistic Regression ROC: 0.9352241019613765
Lasso Logistic Regression PR: 0.003920156085847216
Lasso Logistic Regression ROC: 0.5
Ridge Logistic Regression PR: 0.18232671156327385
Ridge Logistic Regression ROC: 0.9139409967963509

```

Training and Evaluating a Decision Tree Classifier with Increased maxBins in PySpark

```

from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.evaluation import BinaryClassificationEvaluator, MulticlassClassificationEvaluator

# Increase maxBins in the Decision Tree Classifier
dt = DecisionTreeClassifier(featuresCol="features", labelCol="is_fraud", maxBins=850)

# Create a pipeline including the assembler and the model
pipeline_dt = Pipeline(stages=indexers + encoders + [assembler, dt])

# Fit the model on the training data
model_dt = pipeline_dt.fit(balanced_train_data)

# Make predictions on the test data
predictions_dt = model_dt.transform(test_data)

# Evaluators for PR, ROC, and other metrics
bce_pr = BinaryClassificationEvaluator(labelCol="is_fraud", metricName='areaUnderPR', rawPredictionCol="probability")
bce_roc = BinaryClassificationEvaluator(labelCol="is_fraud", metricName="areaUnderROC", rawPredictionCol="probability")

# Print evaluations
print("Decision Tree PR:", bce_pr.evaluate(predictions_dt))
print("Decision Tree ROC:", bce_roc.evaluate(predictions_dt))

```

Decision Tree PR: 0.17479154760404672
Decision Tree ROC: 0.9071965271306931

Cross-Validation for Decision Tree: Pipeline Setup, Parameter Tuning, and Evaluation in PySpark

```

# Cross validation: Decision Tree
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

# Define the Decision Tree Classifier
dt = DecisionTreeClassifier(featuresCol="features", labelCol="is_fraud")

# Create a pipeline
pipeline_dt = Pipeline(stages=indexers + encoders + [assembler, dt])

paramGrid = ParamGridBuilder() \
    .addGrid(dt.maxDepth, [5, 10]) \
    .addGrid(dt.maxBins, [850]) \
    .build()

# Evaluator setup for PR and ROC
bce_pr = BinaryClassificationEvaluator(labelCol="is_fraud", metricName='areaUnderPR', rawPredictionCol="probability")
bce_roc = BinaryClassificationEvaluator(labelCol="is_fraud", metricName="areaUnderROC", rawPredictionCol="probability")

# CrossValidator
crossval_dt = CrossValidator(estimator=pipeline_dt,
                             estimatorParamMaps=paramGrid,
                             evaluator=BinaryClassificationEvaluator(labelCol="is_fraud"),
                             numFolds=3)

# Fit the model using cross-validation
cvModel_dt = crossval_dt.fit(balanced_train_data)

# Use the best model found to make predictions on test data
predictions_dt_cv = cvModel_dt.bestModel.transform(test_data)

# Print evaluations for PR and ROC
print("cv_Decision Tree PR:", bce_pr.evaluate(predictions_dt_cv))
print("cv_Decision Tree ROC:", bce_roc.evaluate(predictions_dt_cv))

```

```

cv_Decision Tree PR: 0.22406467651251838
cv_Decision Tree ROC: 0.891778554063826

```

Training and Evaluating a Gradient Boosted Tree Classifier with Custom maxBins and maxIter in PySpark

```

from pyspark.sql import SparkSession
from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler, StringIndexer
from pyspark.ml.classification import GBTClassifier
from pyspark.ml.evaluation import BinaryClassificationEvaluator, MulticlassClassificationEvaluator

# Define the Gradient Boosted Tree Classifier
gbt = GBTClassifier(featuresCol="features", labelCol="is_fraud", maxIter=10, maxBins=900)

# Create a pipeline including the assembler and the model
pipeline_gbt = Pipeline(stages=indexers + encoders + [assembler, gbt])

# Fit the model on the training data
model_gbt = pipeline_gbt.fit(balanced_train_data)

# Make predictions on the test data
predictions_gbt = model_gbt.transform(test_data)

# Evaluators for PR, ROC, and other metrics
bce_pr = BinaryClassificationEvaluator(labelCol="is_fraud", metricName='areaUnderPR')
bce_roc = BinaryClassificationEvaluator(labelCol="is_fraud", metricName="areaUnderROC")

# Print evaluations
print("Gradient Boosting PR:", bce_pr.evaluate(predictions_gbt))
print("Gradient Boosting ROC:", bce_roc.evaluate(predictions_gbt))

```

Gradient Boosting PR: 0.4223360107291217

Gradient Boosting ROC: 0.895623823353002

```
pip install handyspark
```

Note: you may need to restart the kernel using `dbutils.library.restartPython()` to use updated packages.

Collecting handyspark

Downloading handyspark-0.2.2a1-py2.py3-none-any.whl (39 kB)

Requirement already satisfied: seaborn in /databricks/python3/lib/python3.10/site-packages (from handyspark) (0.11.2)

Requirement already satisfied: numpy in /databricks/python3/lib/python3.10/site-packages (from handyspark) (1.21.5)

Requirement already satisfied: pyarrow in /databricks/python3/lib/python3.10/site-packages (from handyspark) (8.0.0)

Requirement already satisfied: scipy in /databricks/python3/lib/python3.10/site-packages (from handyspark) (1.9.1)

Requirement already satisfied: scikit-learn in /databricks/python3/lib/python3.10/site-packages (from handyspark) (1.1.1)

Requirement already satisfied: pandas in /databricks/python3/lib/python3.10/site-packages (from handyspark) (1.4.4)

Collecting findspark

Downloading findspark-2.0.1-py2.py3-none-any.whl (4.4 kB)

Collecting pyspark

Downloading pyspark-3.5.1.tar.gz (317.0 MB)

317.0/317.0 MB 825.8 kB/s eta 0:00:00

Preparing metadata (setup.py): started

Preparing metadata (setup.py): finished with status 'done'

Requirement already satisfied: matplotlib in /databricks/python3/lib/python3.10/site-packages (from handyspark) (3.5.2)

Requirement already satisfied: kiwisolver<=1.0.1 in /databricks/python3/lib/python3.10/site-packages (from matplotlib->handyspark) (1.4.2)

Requirement already satisfied: cycler<=0.10 in /databricks/python3/lib/python3.10/site-packages (from matplotlib->handyspark) (0.11.0)

```

from handyspark import *
import matplotlib.pyplot as plt

```

Comparing ROC and PR Curves for Various Classifiers: Random Forest, Decision Tree, Gradient Boosting, and Logistic Regression


```

# 'predictions_rf_cv' is the DataFrame resulting from the RandomForest classifier's predictions
bcm_rf = BinaryClassificationMetrics(predictions_rf_cv, scoreCol='probability', labelCol='is_fraud')

# Print area under ROC and PR curves
print("Area under ROC Curve: {:.4f}".format(bcm_rf.areaUnderROC))
print("Area under PR Curve: {:.4f}".format(bcm_rf.areaUnderPR))

# Plot both ROC and PR curves
fig, axs = plt.subplots(1, 2, figsize=(12, 4))
bcm_rf.plot_roc_curve(ax=axs[0])
bcm_rf.plot_pr_curve(ax=axs[1])

# Display the plots
display(plt.show())

# 'predictions_dt_cv' is the DataFrame resulting from the Decision Tree classifier's predictions
bcm_dt = BinaryClassificationMetrics(predictions_dt_cv, scoreCol='probability', labelCol='is_fraud')

# Print area under ROC and PR curves
print("Area under ROC Curve: {:.4f}".format(bcm_dt.areaUnderROC))
print("Area under PR Curve: {:.4f}".format(bcm_dt.areaUnderPR))

# Plot both ROC and PR curves
fig, axs = plt.subplots(1, 2, figsize=(12, 4))
bcm_dt.plot_roc_curve(ax=axs[0])
bcm_dt.plot_pr_curve(ax=axs[1])

# Display the plots
display(plt.show())

# 'predictions_gbt' is the DataFrame resulting from the Gradient Boosting classifier's predictions
bcm_gbt = BinaryClassificationMetrics(predictions_gbt, scoreCol='probability', labelCol='is_fraud')

# Print area under ROC and PR curves
print("Area under ROC Curve: {:.4f}".format(bcm_gbt.areaUnderROC))
print("Area under PR Curve: {:.4f}".format(bcm_gbt.areaUnderPR))

# Plot both ROC and PR curves
fig, axs = plt.subplots(1, 2, figsize=(12, 4))
bcm_gbt.plot_roc_curve(ax=axs[0])
bcm_gbt.plot_pr_curve(ax=axs[1])

# Display the plots
display(plt.show())

# 'predictions_lr' is the DataFrame resulting from the Logistic Regression classifier's predictions
bcm_lr = BinaryClassificationMetrics(predictions_lr, scoreCol='probability', labelCol='is_fraud')

# Print area under ROC and PR curves
print("Area under ROC Curve: {:.4f}".format(bcm_lr.areaUnderROC))
print("Area under PR Curve: {:.4f}".format(bcm_lr.areaUnderPR))

# Plot both ROC and PR curves
fig, axs = plt.subplots(1, 2, figsize=(12, 4))
bcm_lr.plot_roc_curve(ax=axs[0])
bcm_lr.plot_pr_curve(ax=axs[1])

# Display the plots
display(plt.show())

```

/databricks/spark/python/pyspark/sql/context.py:165: FutureWarning: Deprecated in 3.0.0. Use SparkSession.builder.getOrCreate() instead.

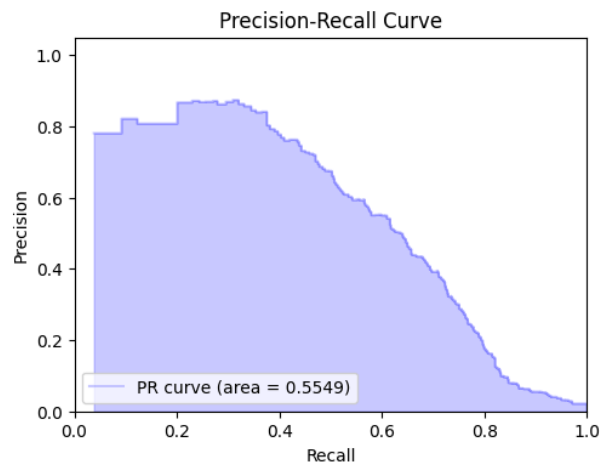
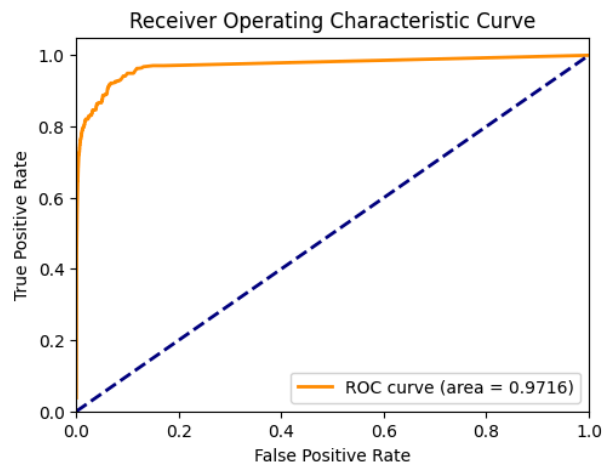
```
warnings.warn(
```

Area under ROC Curve: 0.9716

Area under PR Curve: 0.5549

/databricks/spark/python/pyspark/sql/context.py:165: FutureWarning: Deprecated in 3.0.0. Use SparkSession.builder.getOrCreate() instead.

```
warnings.warn(
```



/databricks/spark/python/pyspark/sql/context.py:165: FutureWarning: Deprecated in 3.0.0. Use SparkSession.builder.getOrCreate() instead.

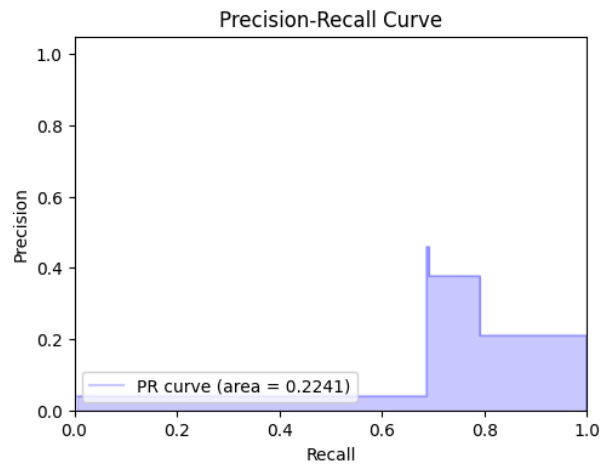
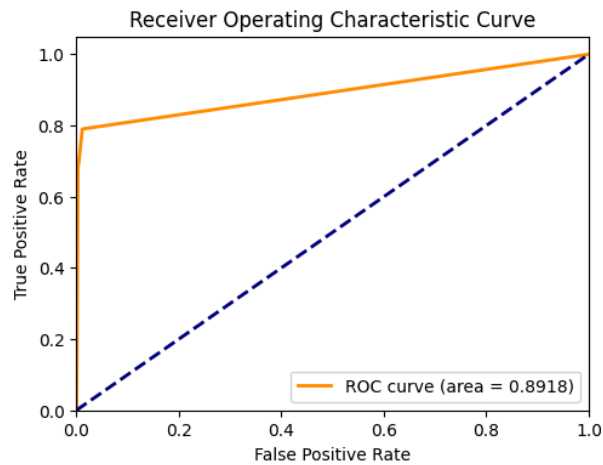
warnings.warn(

Area under ROC Curve: 0.8918

Area under PR Curve: 0.2241

/databricks/spark/python/pyspark/sql/context.py:165: FutureWarning: Deprecated in 3.0.0. Use SparkSession.builder.getOrCreate() instead.

warnings.warn(



/databricks/spark/python/pyspark/sql/context.py:165: FutureWarning: Deprecated in 3.0.0. Use SparkSession.builder.getOrCreate() instead.

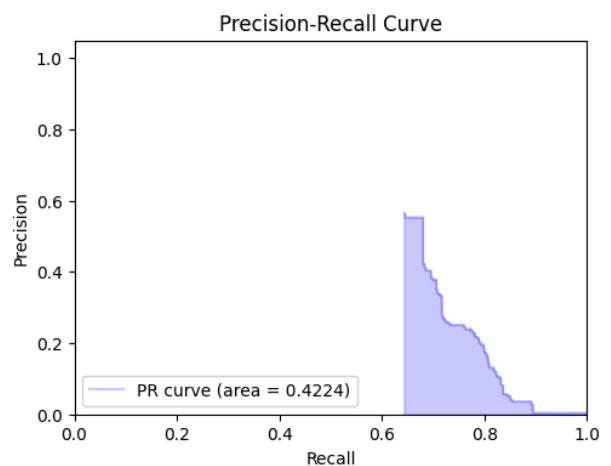
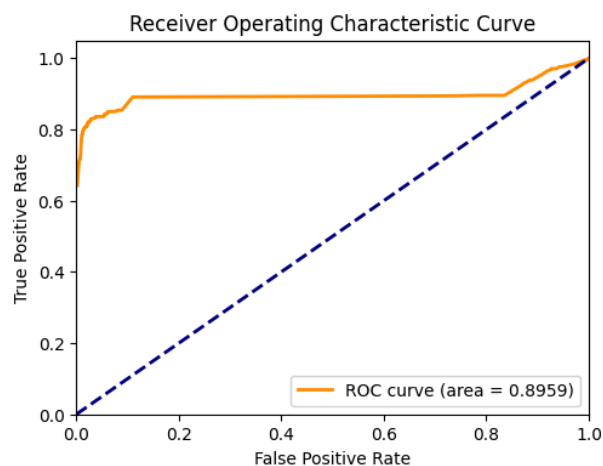
warnings.warn(

Area under ROC Curve: 0.8959

Area under PR Curve: 0.4224

/databricks/spark/python/pyspark/sql/context.py:165: FutureWarning: Deprecated in 3.0.0. Use SparkSession.builder.getOrCreate() instead.

warnings.warn(



```
/databricks/spark/python/pyspark/sql/context.py:165: FutureWarning: Deprecated in 3.0.0. Use SparkSession.builder.getOrCreate() instead.
```

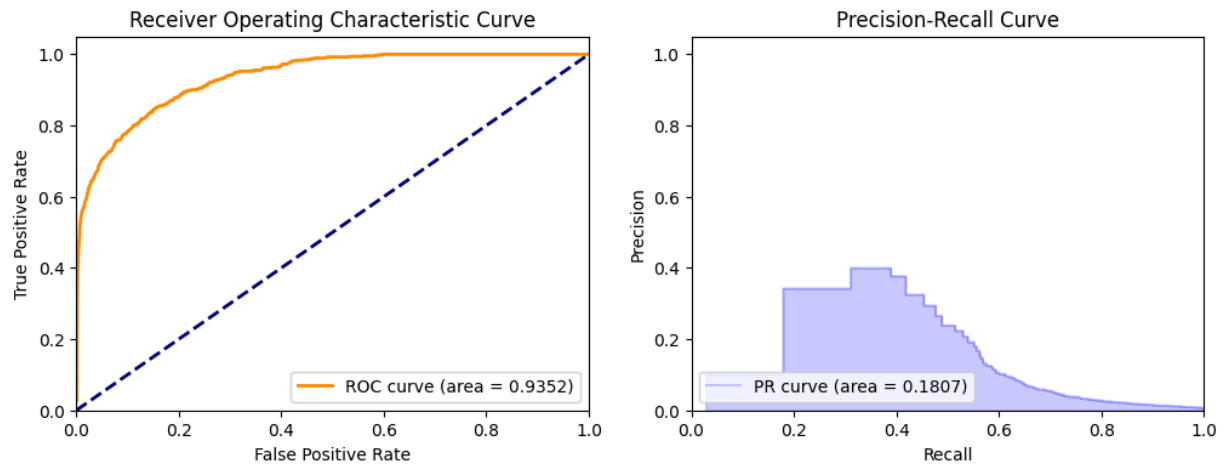
```
warnings.warn(
```

Area under ROC Curve: 0.9352

Area under PR Curve: 0.1807

```
/databricks/spark/python/pyspark/sql/context.py:165: FutureWarning: Deprecated in 3.0.0. Use SparkSession.builder.getOrCreate() instead.
```

```
warnings.warn(
```



Precision-Recall Curve for Random Forest with Optimal Threshold Analysis

```

import pandas as pd
from sklearn.metrics import precision_recall_curve, f1_score
import numpy as np

# Extract the probability of the positive class and the true labels
probabilities_rf = predictions_rf_cv.select("probability").rdd.map(lambda row: row[0][1]).collect()
true_labels_rf = predictions_rf_cv.select("is_fraud").rdd.map(lambda row: row[0]).collect()

# Create a pandas DataFrame with the probabilities and true labels
df_rf = pd.DataFrame({
    "probability": probabilities_rf,
    "is_fraud": true_labels_rf
})

from sklearn.metrics import precision_recall_curve

# Compute precision, recall, and thresholds
precision_rf, recall_rf, thresholds_rf = precision_recall_curve(df_rf['is_fraud'], df_rf['probability'])

# Calculate F1 score for each threshold
f1_scores_rf = 2 * (precision_rf * recall_rf) / (precision_rf + recall_rf)

# Find the index of the maximum F1 score
idx_max_f1_rf = np.argmax(f1_scores_rf)

# Extract the optimal threshold and corresponding F1 score
thresholdOpt_rf = thresholds_rf[idx_max_f1_rf]
fscoreOpt_rf = f1_scores_rf[idx_max_f1_rf]
recallOpt_rf = recall_rf[idx_max_f1_rf]
precisionOpt_rf = precision_rf[idx_max_f1_rf]

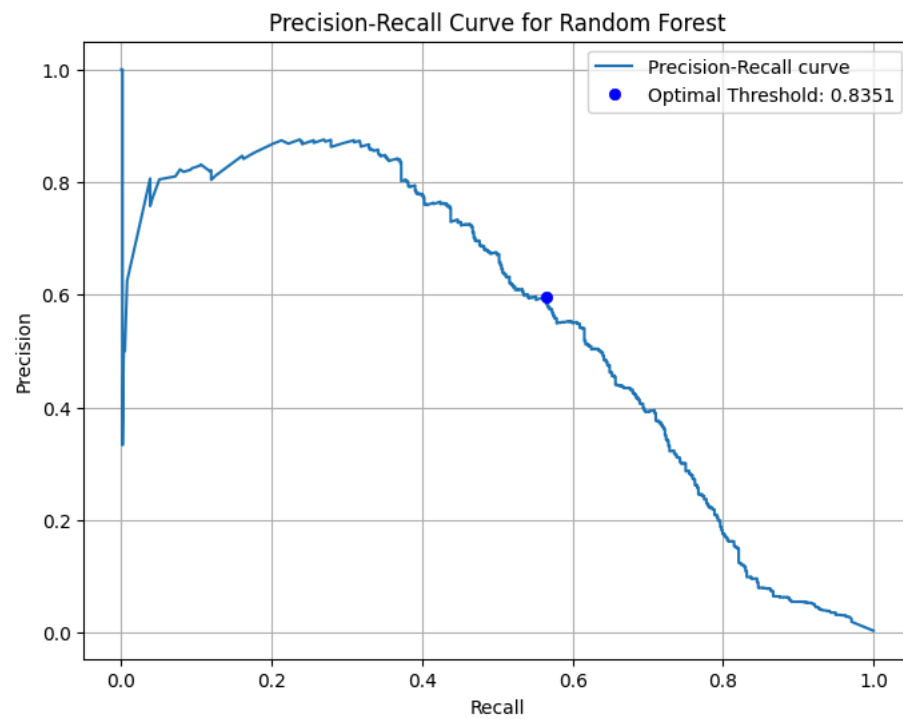
print('Best Threshold for Random Forest: {:.4f} with F-Score: {:.4f}'.format(thresholdOpt_rf, fscoreOpt_rf))
print('Recall: {:.4f}, Precision: {:.4f}'.format(recallOpt_rf, precisionOpt_rf))

import matplotlib.pyplot as plt

plt.figure(figsize=(8, 6))
plt.plot(recall_rf, precision_rf, label='Precision-Recall curve')
plt.plot(recallOpt_rf, precisionOpt_rf, 'bo', label='Optimal Threshold: {:.4f}'.format(thresholdOpt_rf))
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve for Random Forest')
plt.legend(loc="best")
plt.grid(True)
plt.show()

```

Best Threshold for Random Forest: 0.8351 with F-Score: 0.5797
Recall: 0.5651, Precision: 0.5952



Precision-Recall Curve for Decision Tree: Finding the Optimal Threshold with F1 Score Analysis

```

import pandas as pd
from sklearn.metrics import precision_recall_curve, f1_score
import numpy as np

# Select the probability of the positive class and the true labels
probabilities_dt = predictions_dt_cv.select("probability").rdd.map(lambda row: row[0][1]).collect()
true_labels_dt = predictions_dt_cv.select("is_fraud").rdd.map(lambda row: row[0]).collect()

# Now create a pandas DataFrame with the probabilities and true labels
df_dt = pd.DataFrame(list(zip(probabilities_dt, true_labels_dt)), columns=["probability", "is_fraud"])

# Compute precision, recall, and thresholds
precision_dt, recall_dt, thresholds_dt = precision_recall_curve(df_dt['is_fraud'], df_dt['probability'])

# Compute F1 score for each threshold
f1_scores_dt = 2 * recall_dt * precision_dt / (recall_dt + precision_dt)

# Find the index of the maximum F1 score
idx_max_f1_dt = np.argmax(f1_scores_dt)

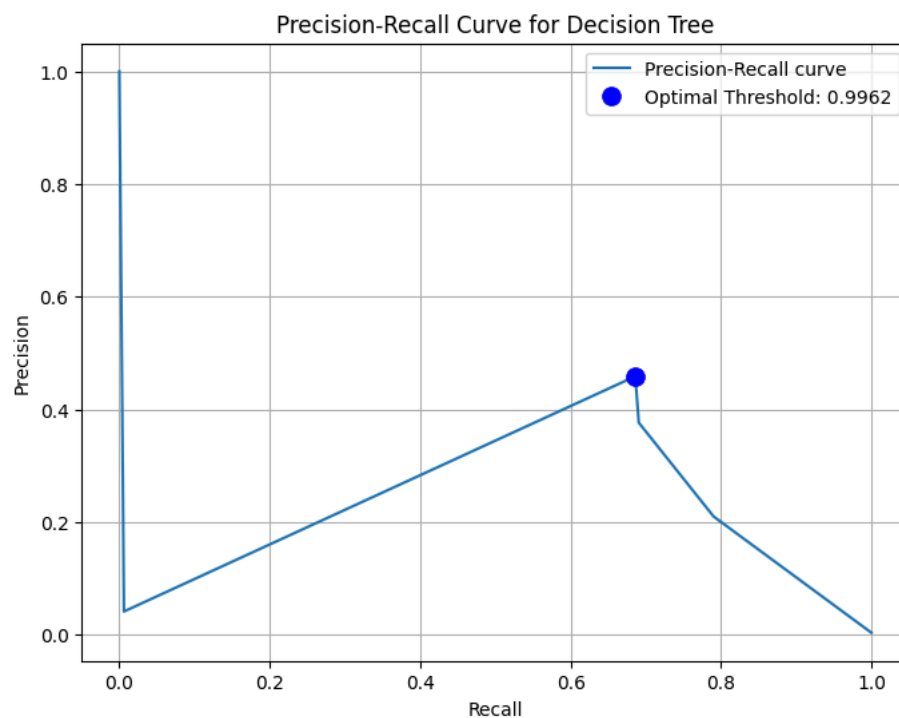
# Extract the optimal threshold and corresponding F1 score
thresholdOpt_dt = thresholds_dt[idx_max_f1_dt]
fscoreOpt_dt = f1_scores_dt[idx_max_f1_dt]
recallOpt_dt = recall_dt[idx_max_f1_dt]
precisionOpt_dt = precision_dt[idx_max_f1_dt]

print('Best Threshold for Decision Tree: {} with F-Score: {}'.format(thresholdOpt_dt, fscoreOpt_dt))
print('Recall: {}, Precision: {}'.format(recallOpt_dt, precisionOpt_dt))

plt.figure(figsize=(8, 6))
plt.plot(recall_dt, precision_dt, label='Precision-Recall curve')
plt.plot(recallOpt_dt, precisionOpt_dt, 'bo', markersize=10, label='Optimal Threshold: {:.4f}'.format(thresholdOpt_dt))
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve for Decision Tree')
plt.legend(loc="upper right")
plt.grid(True)
plt.show()

```

Best Threshold for Decision Tree: 0.9961923010784386 with F-Score: 0.5500306936771026
Recall: 0.6860643185298622, Precision: 0.45901639344262296



Precision-Recall Curve for Gradient Boosting Tree: Optimal Threshold Determination and F1 Score Analysis

```
import pandas as pd
from sklearn.metrics import precision_recall_curve, f1_score
import numpy as np

# Select the probability of the positive class and the true labels
probabilities_gbt = predictions_gbt.select("probability").rdd.map(lambda row: row[0][1]).collect()
true_labels_gbt = predictions_gbt.select("is_fraud").rdd.map(lambda row: row[0]).collect()

# Now create a pandas DataFrame with the probabilities and true labels
df_gbt = pd.DataFrame(list(zip(probabilities_gbt, true_labels_gbt)), columns=["probability", "is_fraud"])

# Compute precision, recall, and thresholds
precision_gbt, recall_gbt, thresholds_gbt = precision_recall_curve(df_gbt['is_fraud'], df_gbt['probability'])

# Compute F1 score for each threshold
f1_scores_gbt = 2 * recall_gbt * precision_gbt / (recall_gbt + precision_gbt)

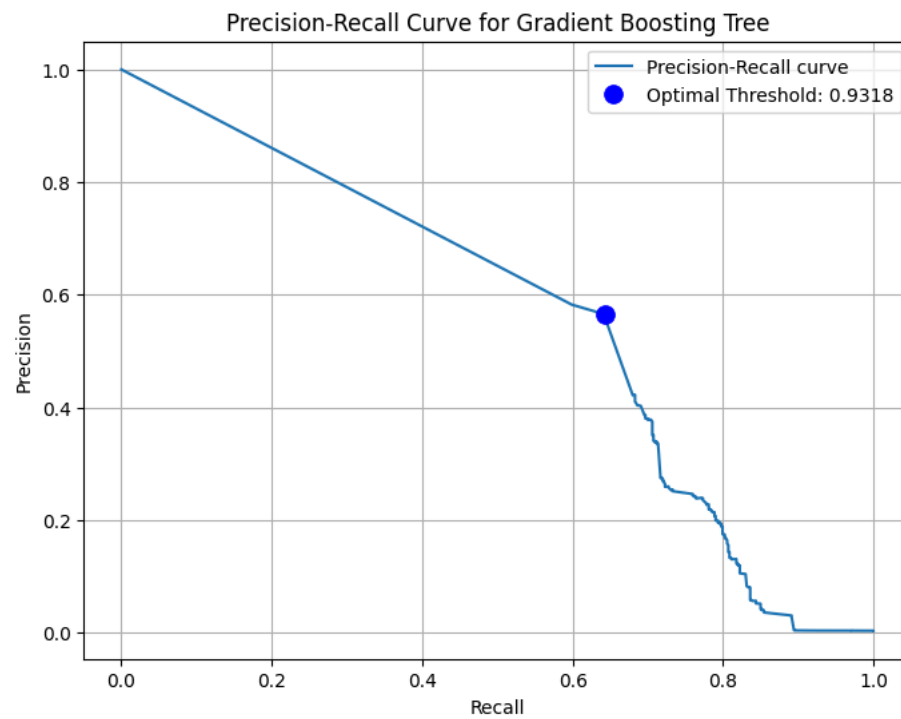
# Find the index of the maximum F1 score
idx_max_f1_gbt = np.argmax(f1_scores_gbt)

# Extract the optimal threshold and corresponding F1 score
thresholdOpt_gbt = thresholds_gbt[idx_max_f1_gbt]
fscoreOpt_gbt = f1_scores_gbt[idx_max_f1_gbt]
recallOpt_gbt = recall_gbt[idx_max_f1_gbt]
precisionOpt_gbt = precision_gbt[idx_max_f1_gbt]

print('Best Threshold for Gradient Boosting Tree: {} with F-Score: {}'.format(thresholdOpt_gbt, fscoreOpt_gbt))
print('Recall: {}, Precision: {}'.format(recallOpt_gbt, precisionOpt_gbt))

plt.figure(figsize=(8, 6))
plt.plot(recall_gbt, precision_gbt, label='Precision-Recall curve')
plt.plot(recallOpt_gbt, precisionOpt_gbt, 'bo', markersize=10, label='Optimal Threshold: {:.4f}'.format(thresholdOpt_gbt))
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve for Gradient Boosting Tree')
plt.legend(loc="upper right")
plt.grid(True)
plt.show()
```

Best Threshold for Gradient Boosting Tree: 0.9318437514164648 with F-Score: 0.6017191977077364
Recall: 0.6431852986217458, Precision: 0.5652759084791387



Precision-Recall Curve for Logistic Regression: Optimal Threshold Analysis with F1 Score and Visualization


```

import pandas as pd
from sklearn.metrics import precision_recall_curve, f1_score
import numpy as np
import matplotlib.pyplot as plt

# Assume 'predictions_lr' is already defined
probabilities_lr = predictions_lr.select("probability").rdd.map(lambda row: row[0][1]).collect()
true_labels_lr = predictions_lr.select("is_fraud").rdd.map(lambda row: row[0]).collect()

# Create a pandas DataFrame
df_lr = pd.DataFrame(list(zip(probabilities_lr, true_labels_lr)), columns=["probability", "is_fraud"])

# Compute precision, recall, and thresholds
precision_lr, recall_lr, thresholds_lr = precision_recall_curve(df_lr['is_fraud'], df_lr['probability'])

# Adjusted F1 score calculation to handle division by zero
f1_scores_lr = np.where((recall_lr + precision_lr) > 0,
                        2 * (precision_lr * recall_lr) / (precision_lr + recall_lr),
                        0)

# Find the index of the maximum F1 score
idx_max_f1_lr = np.argmax(f1_scores_lr)

# Extract the optimal threshold and corresponding F1 score
thresholdOpt_lr = thresholds_lr[idx_max_f1_lr]
fscoreOpt_lr = f1_scores_lr[idx_max_f1_lr]
recallOpt_lr = recall_lr[idx_max_f1_lr]
precisionOpt_lr = precision_lr[idx_max_f1_lr]

print('Best Threshold for Logistic Regression: {} with F-Score: {}'.format(thresholdOpt_lr, fscoreOpt_lr))
print('Recall: {}, Precision: {}'.format(recallOpt_lr, precisionOpt_lr))

plt.figure(figsize=(8, 6))
plt.plot(recall_lr, precision_lr, label='Precision-Recall curve')
plt.scatter(recallOpt_lr, precisionOpt_lr, color='red', s=100, label='Optimal Threshold: {:.4f}'.format(thresholdOpt_lr))
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve for Logistic Regression')
plt.legend(loc="best")
plt.grid(True)
plt.show()

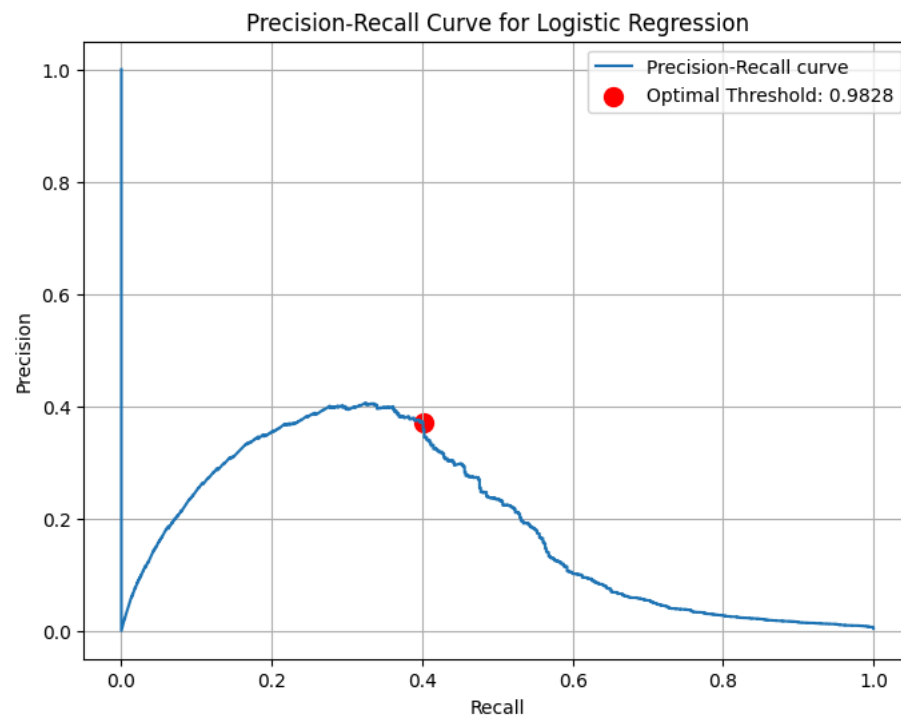
```

/root/.ipykernel/1914/command-2736780779264673-4145757624:18: RuntimeWarning: invalid value encountered in true_divide

```
2 * (precision_lr * recall_lr) / (precision_lr + recall_lr),
```

Best Threshold for Logistic Regression: 0.9828233700227759 with F-Score: 0.3858615611192931

Recall: 0.4012251148545176, Precision: 0.37163120567375885



Confusion Matrix for Logistic Regression: Visualizing and Analyzing Model Performance Metrics

```

from pyspark.sql.functions import lit

# Get counts of true positives, true negatives, false positives, and false negatives
TP = predictions_lr.filter('prediction = 1 AND is_fraud = 1').count()
TN = predictions_lr.filter('prediction = 0 AND is_fraud = 0').count()
FP = predictions_lr.filter('prediction = 1 AND is_fraud = 0').count()
FN = predictions_lr.filter('prediction = 0 AND is_fraud = 1').count()

# Create a DataFrame with the confusion matrix elements
conf_matrix_lr = [[TN, FP],
                  [FN, TP]]

import seaborn as sns
import matplotlib.pyplot as plt

# Plot the confusion matrix
plt.figure(figsize=(8, 8))
sns.heatmap(conf_matrix_lr, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Not Fraud', 'Fraud'],
            yticklabels=['Not Fraud', 'Fraud'])

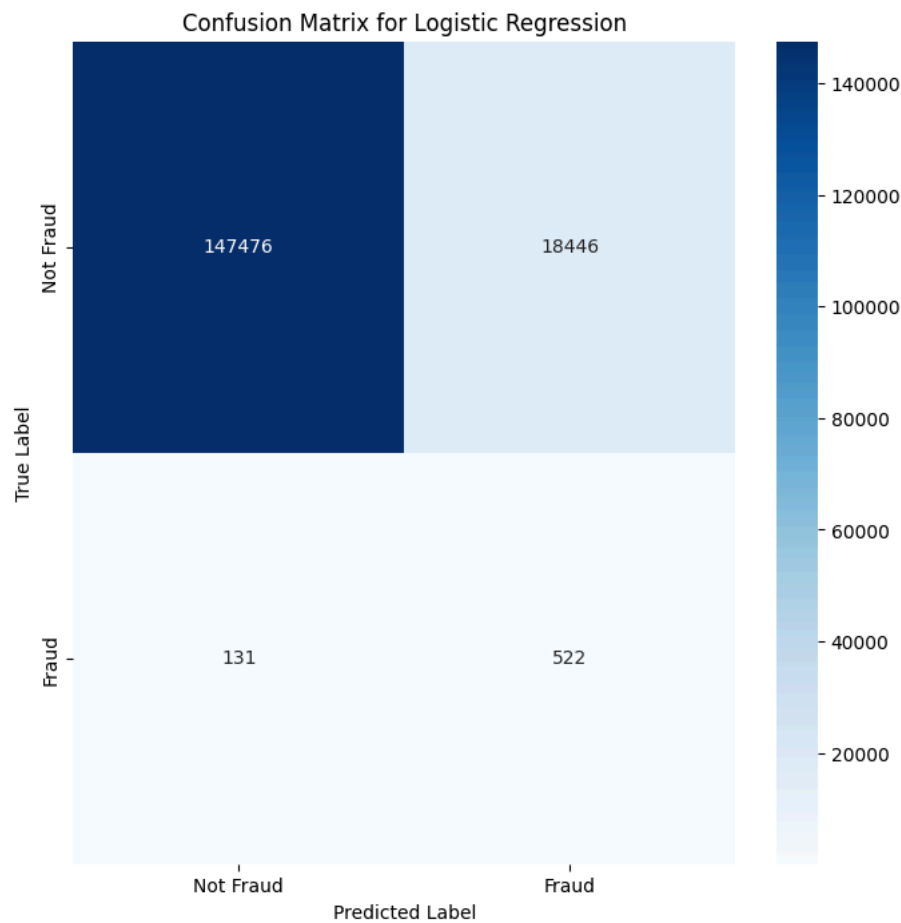
plt.title('Confusion Matrix for Logistic Regression')
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.show()

# Calculating the metrics based on the provided confusion matrix values
TP_lr = 522
TN_lr = 147476
FP_lr = 18446
FN_lr = 131

# Calculations
precision = TP_lr / (TP_lr + FP_lr)
recall = TP_lr / (TP_lr + FN_lr)
f1_score = 2 * (precision * recall) / (precision + recall)
accuracy = (TP_lr + TN_lr) / (TP_lr + FP_lr + FN_lr + TN_lr)

precision, recall, f1_score, accuracy

```



(0.027520033741037536,
0.7993874425727412,
0.053208297232556954,
0.8884766621641903)

Confusion Matrix for Random Forest: Visualization and Model Performance Metrics

```

# Calculate the elements of the confusion matrix
TP_rf = predictions_rf_cv.filter('prediction = 1 AND is_fraud = 1').count()
TN_rf = predictions_rf_cv.filter('prediction = 0 AND is_fraud = 0').count()
FP_rf = predictions_rf_cv.filter('prediction = 1 AND is_fraud = 0').count()
FN_rf = predictions_rf_cv.filter('prediction = 0 AND is_fraud = 1').count()

# Create a DataFrame with the confusion matrix elements
conf_matrix_rf = [[TN_rf, FP_rf],
                  [FN_rf, TP_rf]]

import seaborn as sns
import matplotlib.pyplot as plt

# Plot the confusion matrix
plt.figure(figsize=(8, 8))
sns.heatmap(conf_matrix_rf, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Not Fraud', 'Fraud'],
            yticklabels=['Not Fraud', 'Fraud'])

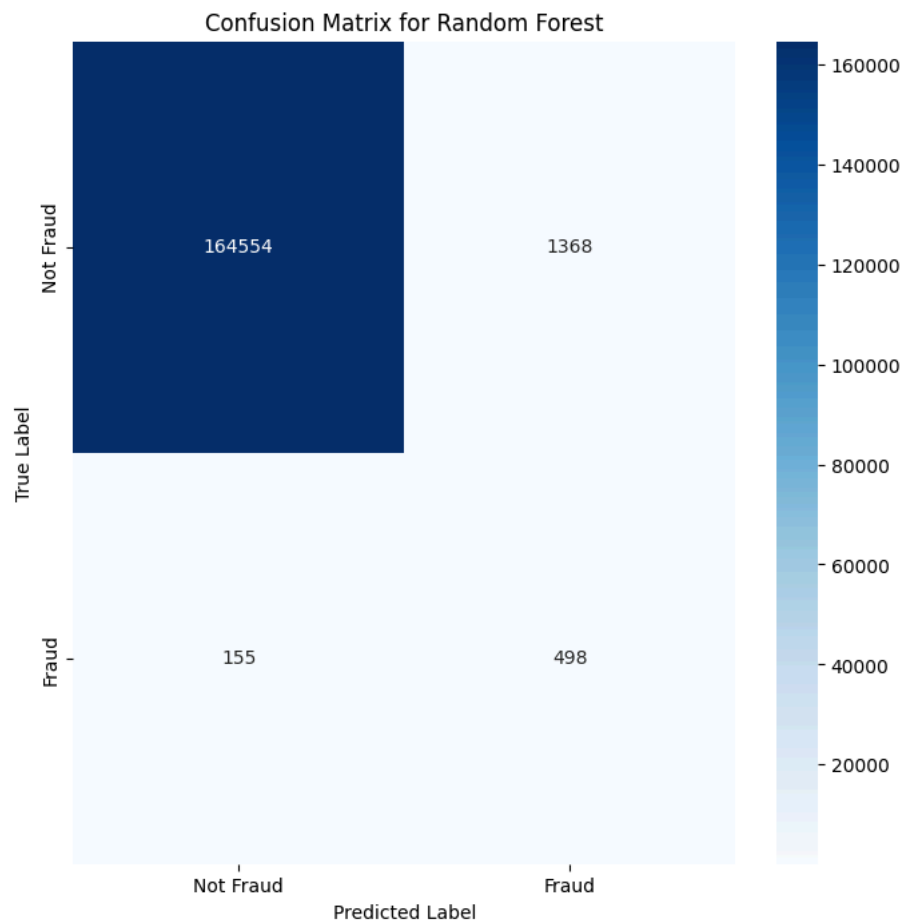
plt.title('Confusion Matrix for Random Forest')
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.show()

# Calculating the metrics based on the provided confusion matrix values for Random Forest
TP_rf = 498
TN_rf = 164554
FP_rf = 1368
FN_rf = 155

# Calculating precision, recall, and F1 score based on the confusion matrix
precision_rf = TP_rf / (TP_rf + FP_rf) if (TP_rf + FP_rf) > 0 else 0
recall_rf = TP_rf / (TP_rf + FN_rf) if (TP_rf + FN_rf) > 0 else 0
f1_score_rf = 2 * (precision_rf * recall_rf) / (precision_rf + recall_rf) if (precision_rf + recall_rf) > 0 else 0
accuracy_rf = (TP_rf + TN_rf) / (TP_rf + FP_rf + FN_rf + TN_rf) if (TP_rf + FP_rf + FN_rf + TN_rf) > 0 else 0

precision_rf, recall_rf, f1_score_rf, accuracy_rf

```



(0.26688102893890675,
0.7626339969372129,
0.3953949980150853,
0.9908569713342338)

Confusion Matrix for Decision Tree: Visualization and Key Performance Metrics

```

# Calculate the elements of the confusion matrix
TP_dt = predictions_dt_cv.filter('prediction = 1 AND is_fraud = 1').count()
TN_dt = predictions_dt_cv.filter('prediction = 0 AND is_fraud = 0').count()
FP_dt = predictions_dt_cv.filter('prediction = 1 AND is_fraud = 0').count()
FN_dt = predictions_dt_cv.filter('prediction = 0 AND is_fraud = 1').count()

# Create a DataFrame with the confusion matrix elements
conf_matrix_dt = [[TN_dt, FP_dt],
                  [FN_dt, TP_dt]]

import seaborn as sns
import matplotlib.pyplot as plt

# Plot the confusion matrix
plt.figure(figsize=(8, 8))
sns.heatmap(conf_matrix_dt, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Not Fraud', 'Fraud'],
            yticklabels=['Not Fraud', 'Fraud'])

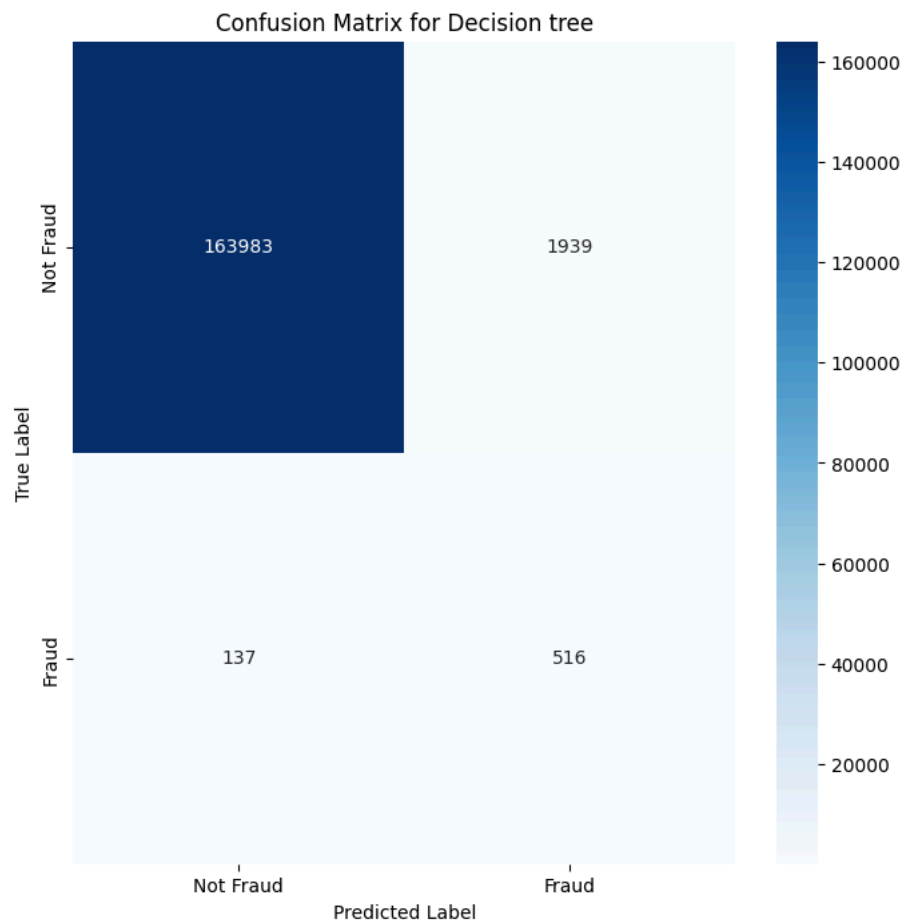
plt.title('Confusion Matrix for Decision tree')
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.show()

# Calculating the metrics based on the provided confusion matrix values for Decision Tree
TP_dt = 516
TN_dt = 163983
FP_dt = 1939
FN_dt = 137

# Calculations for Decision Tree
precision_dt = TP_dt / (TP_dt + FP_dt)
recall_dt = TP_dt / (TP_dt + FN_dt)
f1_score_dt = 2 * (precision_dt * recall_dt) / (precision_dt + recall_dt)
accuracy_dt = (TP_dt + TN_dt) / (TP_dt + FP_dt + FN_dt + TN_dt)

precision_dt, recall_dt, f1_score_dt, accuracy_dt

```



(0.21018329938900204,
0.7901990811638591,
0.33204633204633205,
0.9875371454299865)

Confusion Matrix for Gradient Boosting: Visualization and Model Performance Metrics


```

# Calculate the elements of the confusion matrix
TP_gbt = predictions_gbt.filter('prediction = 1 AND is_fraud = 1').count()
TN_gbt = predictions_gbt.filter('prediction = 0 AND is_fraud = 0').count()
FP_gbt = predictions_gbt.filter('prediction = 1 AND is_fraud = 0').count()
FN_gbt = predictions_gbt.filter('prediction = 0 AND is_fraud = 1').count()

# Create a DataFrame with the confusion matrix elements
conf_matrix_gbt = [[TN_gbt, FP_gbt],
                   [FN_gbt, TP_gbt]]

import seaborn as sns
import matplotlib.pyplot as plt

# Plot the confusion matrix
plt.figure(figsize=(8, 8))
sns.heatmap(conf_matrix_gbt, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Not Fraud', 'Fraud'],
            yticklabels=['Not Fraud', 'Fraud'])

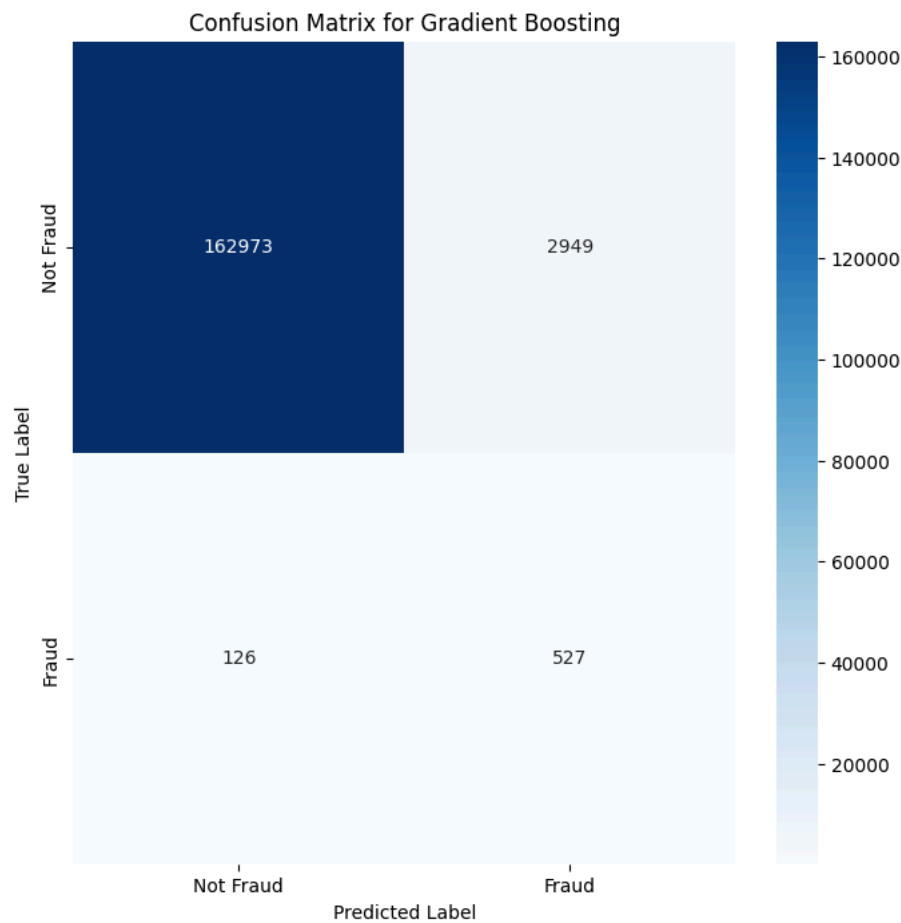
plt.title('Confusion Matrix for Gradient Boosting')
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.show()

# Calculating the metrics based on the provided confusion matrix values for Gradient Boosting
TP_gb = 527
TN_gb = 162973
FP_gb = 2949
FN_gb = 126

# Calculations for Gradient Boosting
precision_gb = TP_gb / (TP_gb + FP_gb)
recall_gb = TP_gb / (TP_gb + FN_gb)
f1_score_gb = 2 * (precision_gb * recall_gb) / (precision_gb + recall_gb)
accuracy_gb = (TP_gb + TN_gb) / (TP_gb + FP_gb + FN_gb + TN_gb)

precision_gb, recall_gb, f1_score_gb, accuracy_gb

```



(0.15161104718066742,
0.8070444104134763,
0.2552676192782756,
0.9815398469158036)

Comparison of Classification Metrics: Logistic Regression, Random Forest, Gradient Boosting Trees, and Decision Trees

```

# Logistic Regression
precision_lr = 0.027520033741037536
recall_lr = 0.7993874425727412
f1_score_lr = 0.053208297232556954
accuracy_lr = 0.8884766621641903
auroc_lr = 0.9352
auprc_lr = 0.1807

# Random Forest
precision_rf = 0.26688102893890675
recall_rf = 0.7626339969372129
f1_score_rf = 0.3953949980150853
accuracy_rf = 0.9908569713342338
auroc_rf = 0.9715
auprc_rf = 0.5549

# Gradient Boosting Trees
precision_gbt = 0.15161104718066742
recall_gbt = 0.8070444104134763
f1_score_gbt = 0.2552676192782756
accuracy_gbt = 0.9815398469158036
auroc_gbt = 0.8959
auprc_gbt = 0.4224

# Decision Trees
precision_dt = 0.21018329938900204
recall_dt = 0.7901990811638591
f1_score_dt = 0.33204633204633205
accuracy_dt = 0.9875371454299865
auroc_dt = 0.8918
auprc_dt = 0.2241

import pandas as pd

# Data for the DataFrame, with each model's metrics as a row
data = [
    [precision_lr, recall_lr, f1_score_lr, accuracy_lr, auroc_lr, auprc_lr],
    [precision_rf, recall_rf, f1_score_rf, accuracy_rf, auroc_rf, auprc_rf],
    [precision_gbt, recall_gbt, f1_score_gbt, accuracy_gbt, auroc_gbt, auprc_gbt],
    [precision_dt, recall_dt, f1_score_dt, accuracy_dt, auroc_dt, auprc_dt]
]

# Column names for the DataFrame
columns = ['Precision', 'Recall', 'F1 Score', 'Accuracy', 'AUROC', 'AUPRC']

# Create the DataFrame
results_df = pd.DataFrame(data, columns=columns, index=['Logistic Regression', 'Random Forest', 'Gradient Boosting Trees', 'Decision Trees'])

results_df

```

	Precision	Recall	F1 Score	Accuracy	AUROC	AUPRC
Logistic Regression	0.027520	0.799387	0.053208	0.888477	0.9352	0.1807
Random Forest	0.266881	0.762634	0.395395	0.990857	0.9715	0.5549
Gradient Boosting Trees	0.151611	0.807044	0.255268	0.981540	0.8959	0.4224
Decision Trees	0.210183	0.790199	0.332046	0.987537	0.8918	0.2241

Comparing Classification Models: Line Plot of Key Metrics for Logistic Regression, Random Forest, Gradient Boosting Trees, and Decision Trees

