databricks BDP_CW2_PartA

(https://databricks.com)

Student Information:

Name: Mahek Kathiriya

Student Number: s2231752

Design and Prototype of a Data Pipeline using Databricks

Introduction

In this assignment, we designed and implemented a prototype data pipeline within a Databricks notebook. The primary goal of the pipeline was to process an open dataset, perform data transformations, store the processed data, and conduct meaningful queries and visualizations. The entire process is documented in the notebook to provide transparency and clarity in our approach.

Part A: Prototype within Databricks

1. Source Data - Storage and Ingest

The first step of the data pipeline is to load the assigned dataset into Databricks. We will read the dataset from a CSV file, as it as our source data.

```
df_train = spark.read.format("csv").option("header",
"true").load("dbfs:/FileStore/shared_uploads/mahekkathiriya91@gmail.com/train.c
sv")
df_test = spark.read.format("csv").option("header",
"true").load("dbfs:/FileStore/shared_uploads/mahekkathiriya91@gmail.com/test.cs
v")
```

2. ETL Transformations

To enhance the quality of the data and prepare it for analysis, we perform several ETL (Extract, Transform, Load) operations. In this section, we demonstrate two types of transformations: standard scaling of numeric features and encoding of categorical features.

2.1 Standard Scaling of Numeric Features

We standardize numeric features, including 'battery_power', 'clock_speed', 'ram', and 'mobile_wt'. This step ensures that all numeric features have a mean of 0 and a standard deviation of 1, which is essential for some machine learning algorithms.

```
from pyspark.sql.functions import col

# Convert columns to double data type
df_train = df_train.withColumn("battery_power",
col("battery_power").cast("double"))
df_train = df_train.withColumn("clock_speed",
col("clock_speed").cast("double"))
df_train = df_train.withColumn("ram", col("ram").cast("double"))
df_train = df_train.withColumn("mobile_wt", col("mobile_wt").cast("double"))

# Convert columns to double data type for df_test
df_test = df_test.withColumn("battery_power",
col("battery_power").cast("double"))
df_test = df_test.withColumn("clock_speed", col("clock_speed").cast("double"))
df_test = df_test.withColumn("ram", col("ram").cast("double"))
df_test = df_test.withColumn("mobile_wt", col("mobile_wt").cast("double"))

# Standard scaling of numeric features
# Define the columns to be scaled
columns_to_scale = ['battery_power', 'clock_speed', 'ram', 'mobile_wt']

# Assemble the features into a vector column
assembler = VectorAssembler(inputCols=columns_to_scale, outputCol='features')
df_train_assembled = assembler.transform(df_train)
df_test_assembled = assembler.transform(df_test)

# Create a StandardScaler object
scaler = StandardScaler(inputCol='features', outputCol='scaled_features')

# Fit and transform the scaler
scaler_model = scaler.fit(df_train_assembled)
df_train_scaled = scaler_model.transform(df_train_assembled)
df_test_scaled = scaler_model.transform(df_test_assembled)
```

2.2 Encoding Categorical Features

We encode categorical features using StringIndexer and OneHotEncoder techniques. Categorical columns such as 'blue', 'dual_sim', 'four_g', 'three_g', 'touch_screen', and 'wifi' are transformed into numerical representations.

```
# Encoding categorical features
# Categorical columns to be encoded
categorical_columns = ['blue', 'dual_sim', 'four_g', 'three_g', 'touch_screen',
'wifi']

# Apply StringIndexer for categorical columns
indexers = [StringIndexer(inputCol=column, outputCol=column + '_index') for
column in categorical_columns]
indexer_models = [indexer.fit(df_train_scaled) for indexer in indexers]
df_train_indexed = df_train_scaled
df_test_indexed = df_test_scaled

# Transform data using the trained indexers
for model in indexer_models:
    df_train_indexed = model.transform(df_train_indexed)
    df_test_indexed = model.transform(df_test_indexed)

# Apply OneHotEncoder for indexed categorical columns
encoders = [OneHotEncoder(inputCol=column + '_index', outputCol=column +
'_encoded') for column in categorical_columns]
encoder_models = [encoder.fit(df_train_indexed) for encoder in encoders]
df_train_encoded = df_train_indexed
df_test_encoded = df_test_indexed

# Transform data using the trained encoders
for model in encoder_models:
    df_train_encoded = model.transform(df_train_encoded)
    df_test_encoded = model.transform(df_test_encoded)
```

3. Storage of Data for Analytics

After performing ETL transformations, we need to store the preprocessed data for further analysis. We save the processed train and test datasets in Parquet format for efficient querying.

```
# Store preprocessed train data, overwriting existing files
train_parquet_location = "dbfs:/FileStore/tables/preprocessed_train.parquet"
df_train_encoded.write.mode("overwrite").parquet(train_parquet_location)

# Store preprocessed test data, overwriting existing files
test_parquet_location = "dbfs:/FileStore/tables/preprocessed_test.parquet"
df_test_encoded.write.mode("overwrite").parquet(test_parquet_location)
```

4. Query and Visualization

To gain insights from the data, we perform non-trivial queries and visualize the results. We demonstrate queries that calculate the average battery power, distribution of mobile weights, and average screen size by price range. Each query is followed by an appropriate visualization.

4.1 Average Battery Power by Price Range

```
from pyspark.sql import functions as F

# Example query: Calculate average RAM by price range
average_ram_by_price =
df_train_encoded.groupBy('price_range').agg(F.avg('ram').alias('avg_ram'))

# Visualization: Bar plot of average RAM by price range
display(average_ram_by_price)
```

Table Visualization 1

	price_range	avg_ram
1	3	3449.232
2	0	785.314
3	1	1679.49
4	2	2582.816

4 rows

4.2 Average Screen Size by Price Range

```
# Query: Calculate the average screen height and width by price range
average_screen_size_by_price = df_train_encoded.groupBy('price_range').agg(
    F.avg('sc_h').alias('avg_screen_height'),
    F.avg('sc_w').alias('avg_screen_width')
)

# Visualization: Scatter plot of average screen size by price range
display(average_screen_size_by_price)
```

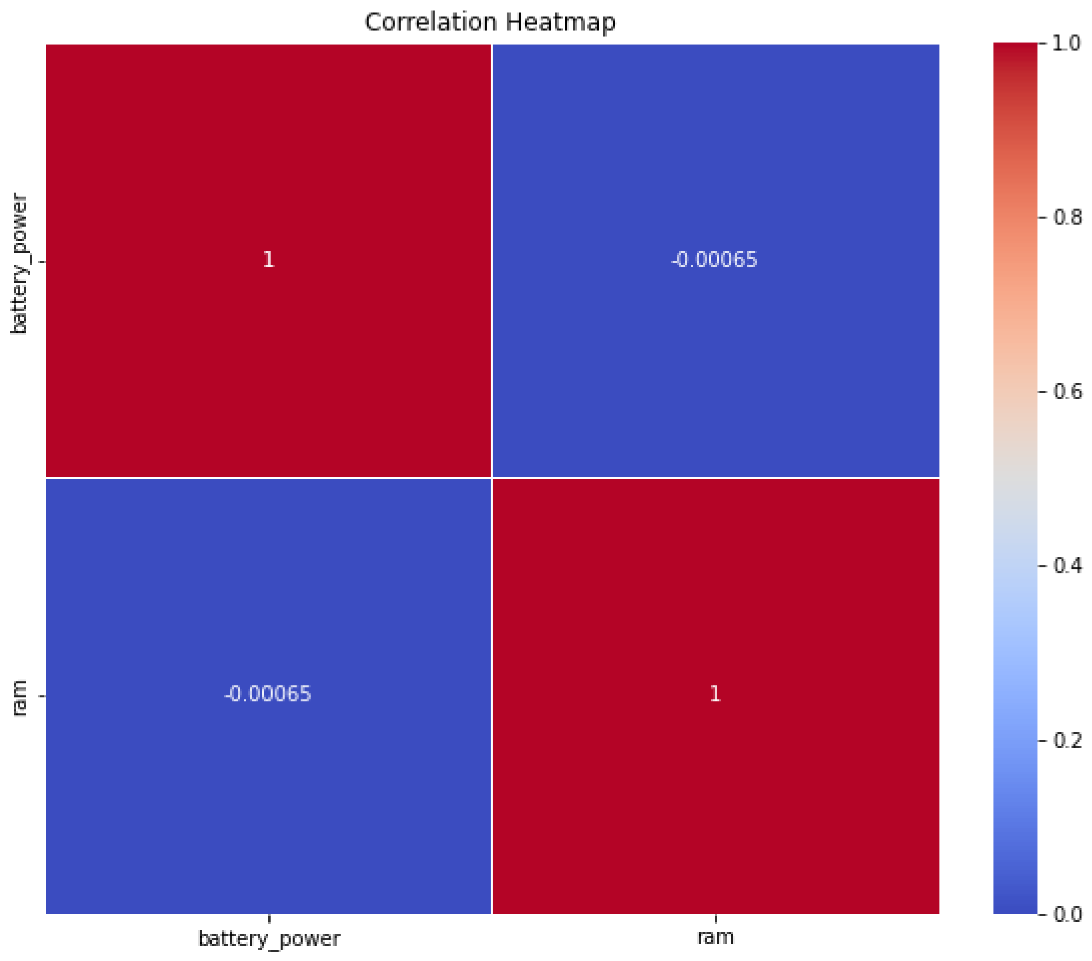
Table Visualization 1

	price_range	avg_screen_height	avg_screen_width
1	3	12.68	6.128
2	0	12.324	5.682
3	1	12.212	5.544
4	2	12.01	5.714

4 rows

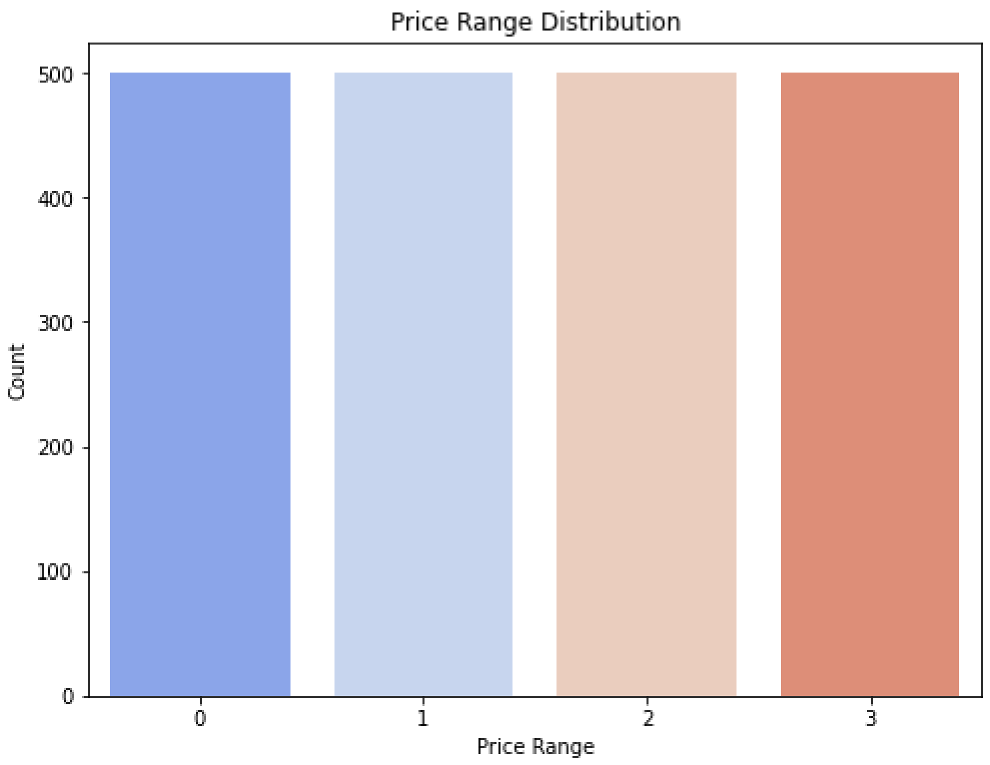
4.3 Correlation between Ram and Battery Power

```
import matplotlib.pyplot as plt
import seaborn as sns

# Select the numerical columns for correlation analysis
numerical_columns = ['battery_power', 'ram', 'px_height', 'px_width']

# Calculate the correlation matrix
correlation_matrix =
df_train_encoded.select(numerical_columns).toPandas().corr()

# Visualization: Heatmap of the correlation matrix
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', linewidths=0.5)
plt.title('Correlation Heatmap')
plt.show()
```



4.4 Price Range Distibution

```
# Visualization: Bar plot of price range distribution
price_range_distribution =
df_train_encoded.groupBy('price_range').count().orderBy('price_range')
price_range_distribution = price_range_distribution.toPandas()

plt.figure(figsize=(8, 6))
sns.barplot(x='price_range', y='count', data=price_range_distribution,
palette='coolwarm')
plt.title('Price Range Distribution')
plt.xlabel('Price Range')
plt.ylabel('Count')
plt.show()
```



Conclusion

In this Databricks notebook, we have successfully designed and implemented a prototype data pipeline. We loaded the source data, performed ETL transformations, stored the preprocessed data, and conducted meaningful queries with visualizations. This pipeline serves as a foundation for further analysis and modeling on the dataset.