`ECC Assignment-2

Adeep Hande

After setting up a 3-node Hadoop cluster with HDFS, we proceeded to set up a PySpark 3-node cluster for distributed data processing using Apache Spark. Here are the steps we followed:

1. Install PySpark: We installed PySpark on each of the three nodes in the cluster by downloading and installing the appropriate version of Apache Spark for Python.
2. Configure Spark Environment: We configured the Spark environment by setting the necessary environment variables, such as SPARK_HOME, in the bashrc or bash_profile file on each node. These variables specify the location of the Spark installation and other required configurations.
3. Set Spark Master and Workers: We designated one of the nodes as the Spark master and the remaining two nodes as Spark workers. We updated the Spark configuration file (spark-env.sh) to specify the Spark master URL, which points to the designated Spark master node.
4. Start Spark Cluster: We started the Spark cluster by running the Spark master and worker processes on their respective nodes using the "start-master.sh" and "start-worker.sh" scripts provided by Spark.
5. Test Spark Cluster: We tested the Spark cluster by running Spark applications on the cluster and verifying the results. We also monitored the Spark cluster using the Spark web UI, which provides insights into cluster usage, performance, and resource allocation.

The codes to the assignment are currently pushed on GitHub: https://github.com/adeepH/ECC/Ass02

Output Logs: https://github.com/adeepH/ECC/Ass02/logs

Question 1:

Preprocessing:

As we observed data that had vehicle year as 0, and several non-NA values in street codes, we did the following preprocessing steps:

1. Dropping all Na/missing entries in the spark dataframe
2. Dropping vehicles with a vehicle year less than 1900: The code uses a threshold of 1900 to filter out records where the "Vehicle Year" column has a value less than 1900. This is done using the condition (col("Vehicle Year") >= threshold) in the filter() function.
3. records with a street codeX of 0: The code uses a threshold of 10^4 (As zip codes are usually between 10001 and 99999) to filter out records where the "Street Code1", "Street Code2", and "Street Code3" columns have values equal to 0. This is done using the conditions (col("Street Code1")//zip_code_threshold!=0), (col("Street Code2")//zip_code_threshold!=0), and (col("Street Code3")//zip_code_threshold != 0) respectively in the filter() function.

1.1 **when are tickets most likely to be issued? (15 pts)**

- Calculates the count of tickets issued for each "Issue Date" and "Violation Time" using the groupBy() and count() functions.
- Sorts the results in descending order based on the count using the sort(desc('count')) function.
- Retrieves the most frequent "Violation Time" and "Issue Date" using the first() function on the sorted count data.
- Output:

```
23/04/15 01:06:22 INFO DAGScheduler: Job 3 finished: first at /home/jovyan/ass2_q11.py:55, took 0.135760
The tickets are most likely to be issued on
 Row(Violation Time='0836A', count=11776) time of day
23/04/15 01:06:28 INFO DAGScheduler: Job 5 finished: first at /home/jovyan/ass2_q11.py:56, took 0.041458 s
The tickets are most likely to be issued on the date
 Row(Issue Date='11/25/2022', count=24214)
```

1.2 **What are the most common years and types of cars to be ticketed? (15 pts)**

- The DataFrame (df) is filtered to select only the "Vehicle Body Type" and "Vehicle Year" columns (df2).
- A groupBy operation is performed on these two columns to count the occurrences of each combination and alias the count as 'count'.
- The resulting DataFrame is ordered in descending order based on the count. Finally, the first row (with the highest count) is printed to show the most common years and types of cars to be ticketed.
- Output:

```
+-----------------+------------+------+
|Vehicle Body Type|Vehicle Year| count|
+-----------------+------------+------+
|             SUBN|        2021|210158|
+-----------------+------------+------+
only showing top 1 row
```

1.3 Where are tickets most commonly issued? (15 pts)

- We identify the most frequent location where tickets are issued by grouping the data by "Street Name" and "Violation County", aggregating the counts of occurrences, and then sorting in descending order.
- Finally, we display the top result with the highest count using the "show(1)" function.
- Output:

```
+-----------+----------------+-----+
|Street Name|Violation County|count|
+-----------+----------------+-----+
|   Broadway|              NY|55597|
+-----------+----------------+-----+
only showing top 1 row
```

1.4 Which color of the vehicle is most likely to get a ticket? (15 pts)

- We group the DataFrame (df) by the "Vehicle Color" column, counts the occurrences of each color, sorts the counts in descending order, and printthe color with the highest count, which is the most likely color of the vehicle to receive a ticket.
- Output:
-

```
The most likely color of the vehicle to get a ticket is
 Row(Vehicle Color='WH', count=940122)
```

1.5 Clustering to find the probability of black car getting the ticket

- For the data, we perform K-Means clustering with cluster ranges from 2 to 10.
- To select the optimal cluster count, we use ClusteringEvaluator() from pyspark, which calculates the silhouette scores of each all cluster models, and we select the cluster with value n, for which k-means clustering has the lowest silhouette score.
- In our case, the most optimal cluster count was 3.
- The preprocessing involved for clustering are:
- We performs StringIndexing on the "Vehicle Color" column, transforming it into a numerical indexed column called "Indexed Color".
- We then assemble the selected columns (Street Code1, Street Code2, Street Code3, Indexed Color) into a vector column called "features" using VectorAssembler.
- This then scales the "features" column using StandardScaler, creating a new column called "scaled_features".
- we perform K-means clustering on the "scaled_features" column using the KMeans algorithm for different values of K (ranging from K_min to K_max). It calculates the Silhouette score for each value of K using ClusteringEvaluator and stores it in a list called "silhouette_scores"
- We extract the cluster_id, c,of the predictions based on what was provided in the question, and then we calculate the percentage of black cars in the cluster with cluster_id c.
- Output:
-

```
23/04/14 20:38:42 WARN Utils: Your hostname, Adeeps-MacBook-Air.local resolves to a loopback address: 127.0.0.1
23/04/14 20:38:42 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
Based on Silhoutte Score Evaluation, the most optimal cluster count is
 3
The probability of the ticketed black car in the cluster is:

3676545
0.01257049757312912
```

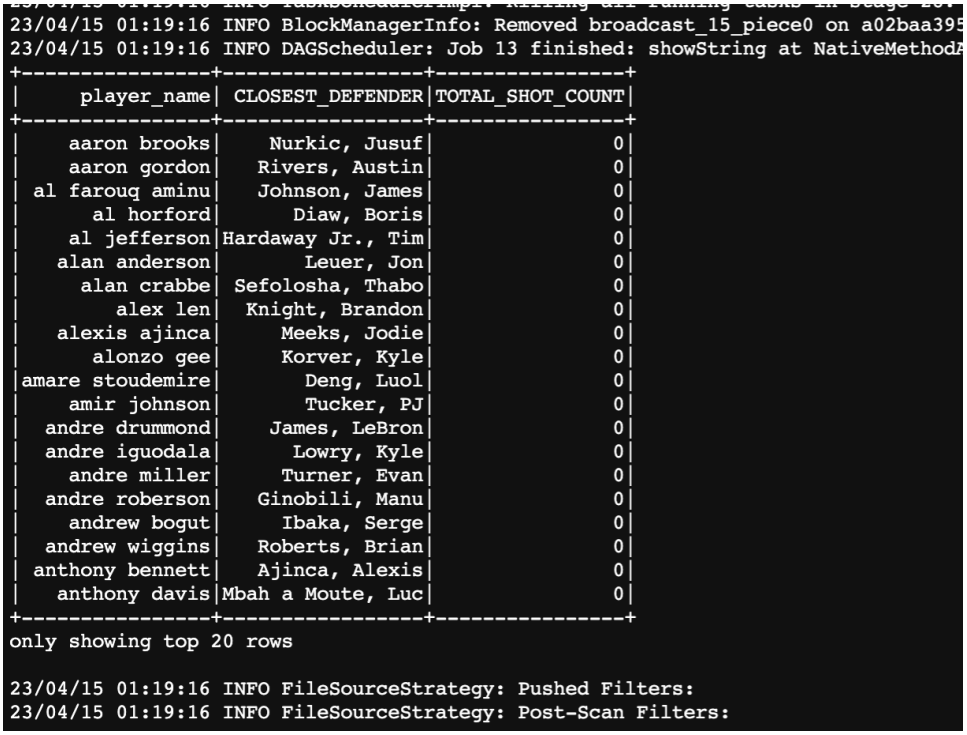
Question 2

Preprocessing:

- The data was quite well documented and did not have any missing values. However, we did drop nan values, if any.
- We encode the SHOT_RESULT as a binary variable, with **missed** as 0 and **made** as 1.

2.1 **Based on the fear score, for each player, please find out who is his ”most unwanted defender”. (10 pts)**

- Encoding of SHOT_RESULT: The 'SHOT_RESULT' column was encoded as a binary variable, with 1 representing a made shot and 0 representing a missed shot. This was done using the 'withColumn' method and the 'when' and 'otherwise' functions from PySpark.

- Grouping and Aggregating: The DataFrame was then grouped by 'player_name' and 'CLOSEST_DEFENDER', and the sum of 'SHOT_RESULT' was calculated for each group using the 'groupBy' and 'sum' methods. The resulting DataFrame was ordered in ascending order of the sum of 'SHOT_RESULT' using the 'orderBy' method.
- Renaming Column: The column name 'sum(SHOT_RESULT)' was renamed to 'TOTAL_SHOT_COUNT' using the 'withColumnRenamed' method.
- Window Specification: A window specification was created using the 'Window' function from PySpark, partitioning the data by 'player_name' and ordering it by 'TOTAL_SHOT_COUNT'.
- Row Number Assignment: A new column 'row_number' was added to the DataFrame using the 'row_number()' function over the window specification.
- Duplicate Row Filtering: The DataFrame was then filtered to get only the rows where 'row_number' is greater than 1, which indicates duplicate rows based on the window specification.
- Duplicate Index Extraction: The row numbers (indexes) of the duplicate rows were extracted from the filtered DataFrame.
- Duplicate Row Removal: The duplicate rows were dropped from the original DataFrame 'df' using the 'filter' method with 'row_number' equals to 1, effectively keeping only the rows with the highest 'TOTAL_SHOT_COUNT' for each 'player_name'.
- DataFrame Display: The resulting DataFrame with duplicate rows removed was displayed using the 'show' method.
- DataFrame Export: The resulting DataFrame was saved as a CSV file named 'player_unwanted_defender_pairs.csv' with header and overwrite mode using the 'write' method with 'csv' format.
- Output:
- 

```
23/04/15 01:19:16 INFO BlockManagerInfo: Removed broadcast_15_piece0 on a02baa395
23/04/15 01:19:16 INFO DAGScheduler: Job 13 finished: showString at NativeMethodA
+---------------+----------------+---------------+
|    player_name| CLOSEST_DEFENDER|TOTAL_SHOT_COUNT|
+---------------+----------------+---------------+
|   aaron brooks|    Nurkic, Jusuf|              0|
|   aaron gordon|   Rivers, Austin|              0|
|al farouq aminu|   Johnson, James|              0|
|     al horford|      Diaw, Boris|              0|
|   al jefferson|Hardaway Jr., Tim|              0|
|  alan anderson|       Leuer, Jon|              0|
|    alan crabbe| Sefolosha, Thabo|              0|
|       alex len|  Knight, Brandon|              0|
|  alexis ajinca|     Meeks, Jodie|              0|
|     alonzo gee|     Korver, Kyle|              0|
|amare stoudemire|       Deng, Luol|              0|
|   amir johnson|       Tucker, PJ|              0|
| andre drummond|    James, LeBron|              0|
| andre iguodala|      Lowry, Kyle|              0|
|   andre miller|     Turner, Evan|              0|
| andre roberson|   Ginobili, Manu|              0|
|   andrew bogut|     Ibaka, Serge|              0|
| andrew wiggins|   Roberts, Brian|              0|
|anthony bennett|   Ajinca, Alexis|              0|
|  anthony davis|Mbah a Moute, Luc|              0|
+---------------+----------------+---------------+
only showing top 20 rows

23/04/15 01:19:16 INFO FileSourceStrategy: Pushed Filters:
23/04/15 01:19:16 INFO FileSourceStrategy: Post-Scan Filters:
```

- The complete list of player-unwanted_defender pairs are here: https://github.com/adeepH/ECC/Ass02/unwanted_defender_list.csv

2.2 Find the most comfortable zone for given players

- Data Subsetting: The DataFrame df is subsetted to retain only the columns 'player_name', 'SHOT_DIST', 'CLOSE_DEF_DIST', and 'SHOT_CLOCK'. The resulting DataFrame is named subset_df.
- Feature Scaling: The numerical columns ('SHOT_DIST', 'CLOSE_DEF_DIST', 'SHOT_CLOCK') in subset_df are assembled into a vector column named 'comfortable_zone' using VectorAssembler. Then, the 'comfortable_zone' vector column is scaled using StandardScaler, and the resulting scaled vector column is named 'scaled_comfortable_zone'.
- K-means Clustering: A K-means clustering model with 4 clusters (k=4) is created using KMeans from the pyspark.ml.clustering module. The 'scaled_comfortable_zone' vector column in subset_df is used as the input features for clustering. The resulting model is named kmeans_model. The clustering is performed on subset_df and the resulting DataFrame is named clustered_df.
- Grouping by Players and Cluster Counts: The clustered_df DataFrame is grouped by 'player_name' and 'prediction' (the cluster label assigned by the K-means model) using groupBy(). The count of occurrences of each cluster for each player is computed using agg() with count("*").alias("cluster_counts"). The resulting DataFrame is named groupby_clustered_df.
- Finding Most Frequent Comfortable Zone for Players: For each player in the list 'name_list' (containing player names), the 'groupby_clustered_df' DataFrame is filtered to get the rows where 'player_name' matches the current player name. The resulting subset is then ordered by 'cluster_counts' in descending order using orderBy(desc('cluster_counts')). The most frequent comfortable zone (cluster) for the player is determined by selecting the 'prediction' value from the first row of the filtered subset. Finally, a print statement is used to display the most frequent comfortable zone (cluster) for each player in 'name_list'.
- Output 1:

```
23/04/15 01:24:06 INFO DAGScheduler: Job 40 finished: showString at Nativ
23/04/15 01:24:06 INFO CodeGenerator: Code generated in 3.871792 ms
+------------+----------+--------------+
| player_name|prediction|cluster_counts|
+------------+----------+--------------+
|james harden|         0|           357|
|james harden|         2|           281|
|james harden|         1|           207|
|james harden|         3|           161|
+------------+----------+--------------+
```

```
23/04/15 01:30:15 INFO CodeGenerator: Code generated in 2.639459 ms
From the cluster counts, we observe that the most comfortable zone for james harden is Zone: 0
```

- Output 2:

```
23/04/15 01:24:07 INFO DAGScheduler: Job 44 finished: showString at NativeMethodAccessorImpl.ja
+-----------+----------+--------------+
|player_name|prediction|cluster_counts|
+-----------+----------+--------------+
| chris paul|         0|           332|
| chris paul|         1|           235|
| chris paul|         2|           166|
| chris paul|         3|           118|
+-----------+----------+--------------+

23/04/15 01:24:07 INFO FileSourceStrategy: Pushed Filters:
```

```
23/04/15 01:30:16 INFO DAGScheduler: Job 46 finished: first at /home/jovyan/ass2_q22.py:58, took 0.015486 s
From the cluster counts, we observe that the most comfortable zone for chris paul is Zone: 0
23/04/15 01:30:16 INFO FileSourceStrategy: Pushed Filters:
```

- Output 3:

```
23/04/15 01:24:08 INFO DAGScheduler: Job 48 finished: showString at NativeMethod
+-------------+----------+--------------+
|  player_name|prediction|cluster_counts|
+-------------+----------+--------------+
|stephen curry|         1|           374|
|stephen curry|         2|           234|
|stephen curry|         0|           225|
|stephen curry|         3|           108|
+-------------+----------+--------------+

23/04/15 01:24:08 INFO FileSourceStrategy: Pushed Filters:
```

```
23/04/15 01:30:16 INFO DAGScheduler: Job 50 finished: first at /home/jovyan/ass2_q22.py:58, took 0.015155 s
From the cluster counts, we observe that the most comfortable zone for stephen curry is Zone: 1
23/04/15 01:30:16 INFO FileSourceStrategy: Pushed Filters:
```

- Output 4:

```
23/04/15 01:24:09 INFO DAGScheduler: Job 52 finished: showString at NativeMethodAccessorImpl.jav
+------------+----------+--------------+
| player_name|prediction|cluster_counts|
+------------+----------+--------------+
|lebron james|         0|           325|
|lebron james|         2|           250|
|lebron james|         3|           224|
|lebron james|         1|           148|
+------------+----------+--------------+

23/04/15 01:24:09 INFO FileSourceStrategy: Pushed Filters:
```

```
23/04/15 01:30:17 INFO DAGScheduler: Job 54 finished: first at /home/jovyan/ass2_q22.py:58, took 0.016762 s
From the cluster counts, we observe that the most comfortable zone for lebron james is Zone: 0
23/04/15 01:30:17 INFO SparkContext: Invoking stop() from shutdown hook
```

Final Output:

```
From the cluster counts, we observe that the most comfortable zone for james harden is Zone: 0

From the cluster counts, we observe that the most comfortable zone for chris paul is Zone: 0

From the cluster counts, we observe that the most comfortable zone for stephen curry is Zone: 1

From the cluster counts, we observe that the most comfortable zone for lebron james is Zone: 0
```