ENGR 516 - Assignment 2

by sdiware@iu.edu

Spark Installation:

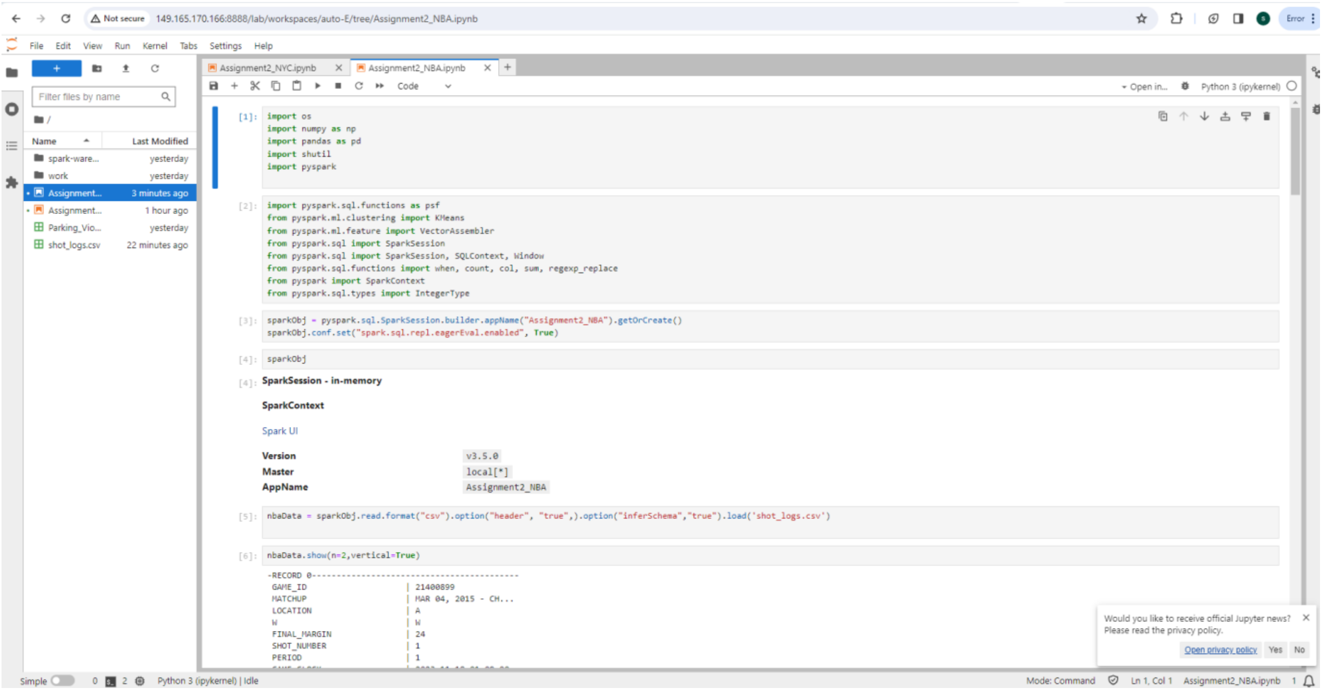
Below are the installation steps for Spark:

1. I have created a docker file called docker-compose.yaml.
2. I added below content in the docker file.

```
exouser@sdiware-ecc:~/Spark$ cat docker-compose.yaml
version: '3'
services:
    spark:
      image: jupyter/pyspark-notebook
      ports:
        - "8888:8888"
        - "4040-4080:4040-4080"
      volumes:
        - ./notebooks:/home/jovyan/work/notebooks/
exouser@sdiware-ecc:~/Spark$
```

3. Using this docker image I have installed PySpark in Jupyter.
4. I started the docker container using sudo docker-compose up.

```
exouser@sdiware-ecc:~/Spark$ sudo docker-compose up
Starting spark_spark_1 ... done
Attaching to spark_spark_1
spark_1  | Entered start.sh with args: jupyter lab
spark_1  | Running hooks in: /usr/local/bin/start-notebook.d as uid: 1000 gid: 100
spark_1  | Done running hooks in: /usr/local/bin/start-notebook.d
spark_1  | Running hooks in: /usr/local/bin/before-notebook.d as uid: 1000 gid: 100
spark_1  | Sourcing shell script: /usr/local/bin/before-notebook.d/spark-config.sh
spark_1  | Done running hooks in: /usr/local/bin/before-notebook.d
spark_1  | Executing the command: jupyter lab
spark_1  | [I 2023-11-17 23:03:24.381 ServerApp] Package jupyterlab took 0.0000s to import
spark_1  | [I 2023-11-17 23:03:24.412 ServerApp] Package jupyter_lsp took 0.0306s to import
spark_1  | [W 2023-11-17 23:03:24.412 ServerApp] A `_jupyter_server_extension_points` function was not found in jupyter_lsp. Instead, a `_jupyter_server_extension_paths` function was found and will be used for now. This function name wi
l be deprecated in future releases of Jupyter Server.
spark_1  | [I 2023-11-17 23:03:24.416 ServerApp] Package jupyter_server_mathjax took 0.0028s to import
spark_1  | [I 2023-11-17 23:03:24.429 ServerApp] Package jupyter_server_terminals took 0.0129s to import
spark_1  | [I 2023-11-17 23:03:24.501 ServerApp] Package jupyterlab_git took 0.0712s to import
spark_1  | [I 2023-11-17 23:03:24.509 ServerApp] Package nbclassic took 0.0081s to import
spark_1  | [W 2023-11-17 23:03:24.513 ServerApp] A `_jupyter_server_extension_points` function was not found in nbclassic. Instead, a `_jupyter_server_extension_paths` function was found and will be used for now. This function name will
be deprecated in future releases of Jupyter Server.
spark_1  | [I 2023-11-17 23:03:24.514 ServerApp] Package nbdime took 0.0000s to import
spark_1  | [I 2023-11-17 23:03:24.514 ServerApp] Package notebook took 0.0000s to import
spark_1  | [I 2023-11-17 23:03:24.517 ServerApp] Package notebook_shim took 0.0000s to import
spark_1  | [W 2023-11-17 23:03:24.517 ServerApp] A `_jupyter_server_extension_points` function was not found in notebook_shim. Instead, a `_jupyter_server_extension_paths` function was found and will be used for now. This function name w
ill be deprecated in future releases of Jupyter Server.
spark_1  | [I 2023-11-17 23:03:24.518 ServerApp] jupyter_lsp | extension was successfully linked.
spark_1  | [I 2023-11-17 23:03:24.522 ServerApp] jupyter_server_mathjax | extension was successfully linked.
spark_1  | [I 2023-11-17 23:03:24.526 ServerApp] jupyter_server_terminals | extension was successfully linked.
spark_1  | [I 2023-11-17 23:03:24.531 ServerApp] jupyterlab | extension was successfully linked.
spark_1  | [I 2023-11-17 23:03:24.531 ServerApp] jupyterlab_git | extension was successfully linked.
spark_1  | [I 2023-11-17 23:03:24.535 ServerApp] nbclassic | extension was successfully linked.
spark_1  | [I 2023-11-17 23:03:24.535 ServerApp] nbdime | extension was successfully linked.
spark_1  | [I 2023-11-17 23:03:24.540 ServerApp] notebook | extension was successfully linked.
spark_1  | [I 2023-11-17 23:03:24.960 ServerApp] notebook_shim | extension was successfully linked.
spark_1  | [I 2023-11-17 23:03:25.025 ServerApp] notebook_shim | extension was successfully loaded.
spark_1  | [I 2023-11-17 23:03:25.027 ServerApp] jupyter_lsp | extension was successfully loaded.
spark_1  | [I 2023-11-17 23:03:25.027 ServerApp] jupyter_server_mathjax | extension was successfully loaded.
spark_1  | [I 2023-11-17 23:03:25.028 ServerApp] jupyter_server_terminals | extension was successfully loaded.
spark_1  | [I 2023-11-17 23:03:25.042 LabApp] JupyterLab extension loaded from /opt/conda/lib/python3.11/site-packages/jupyterlab
spark_1  | [I 2023-11-17 23:03:25.042 LabApp] JupyterLab application directory is /opt/conda/share/jupyter/lab
spark_1  | [I 2023-11-17 23:03:25.042 LabApp] Extension Manager is 'pypi'.
spark_1  | [I 2023-11-17 23:03:25.045 ServerApp] jupyterlab | extension was successfully loaded.
spark_1  | [I 2023-11-17 23:03:25.050 ServerApp] jupyterlab_git | extension was successfully loaded.
spark_1  | [I 2023-11-17 23:03:25.058 ServerApp] nbclassic | extension was successfully loaded.
spark_1  | [I 2023-11-17 23:03:25.167 ServerApp] nbdime | extension was successfully loaded.
spark_1  | [I 2023-11-17 23:03:25.170 ServerApp] notebook | extension was successfully loaded.
spark_1  | [I 2023-11-17 23:03:25.170 ServerApp] Serving notebooks from local directory: /home/jovyan
spark_1  | [I 2023-11-17 23:03:25.170 ServerApp] Jupyter Server 2.8.0 is running at:
spark_1  | [I 2023-11-17 23:03:25.170 ServerApp] http://45d2c2899c2c:8888/lab?token=a5e4c93a606cfcc3beecdcfcc91ec5548c003729397063fe
spark_1  | [I 2023-11-17 23:03:25.170 ServerApp]     http://127.0.0.1:8888/lab?token=a5e4c93a606cfcc3beecdcfcc91ec5548c003729397063fe
spark_1  | [I 2023-11-17 23:03:25.171 ServerApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
spark_1  | [C 2023-11-17 23:03:25.175 ServerApp]
spark_1  |
spark_1  |     To access the server, open this file in a browser:
spark_1  |         file:///home/jovyan/.local/share/jupyter/runtime/jpserver-7-open.html
spark_1  |     Or copy and paste one of these URLs:
spark_1  |         http://45d2c2899c2c:8888/lab?token=a5e4c93a606cfcc3beecdcfcc91ec5548c003729397063fe
spark_1  |         http://127.0.0.1:8888/lab?token=a5e4c93a606cfcc3beecdcfcc91ec5548c003729397063fe
spark_1  | [W 2023-11-17 23:03:25.360 ServerApp] 404 GET /api/kernels/f7564d93-c2d4-4025-9113-b8d0627dd48a?1700262199874 (66.244.81.176): Kernel does not exist: f7564d93-c2d4-4025-9113-b8d0627dd48a
spark_1  | [W 2023-11-17 23:03:25.361 ServerApp] wrote error: 'Kernel does not exist: f7564d93-c2d4-4025-9113-b8d0627dd48a'
spark_1  |     Traceback (most recent call last):
```



PART 1 - NY Parking Violations

Spark Session:

```
[1]: import os
     import numpy as np
     import pandas as pd
     import shutil
     import pyspark
```

```
[2]: import pyspark.sql.functions as psf
     from pyspark.ml.clustering import KMeans
     from pyspark.ml.feature import VectorAssembler
     from pyspark.sql import SparkSession
     from pyspark.sql import SparkSession, SQLContext, Window
     from pyspark.sql.functions import when, count, col, sum, regexp_replace
     from pyspark import SparkContext
     from pyspark.sql.types import IntegerType
```

```
[9]: sparkObj = pyspark.sql.SparkSession.builder.appName("Assignment2-NYC").getOrCreate()
     sparkObj.conf.set("spark.sql.repl.eagerEval.enabled", True)
```

```
[10]: sparkObj
```

[10]: **SparkSession - in-memory**

SparkContext

Spark UI

Version	v3.5.0
Master	local[*]
AppName	Assignment2-NYC

Data File: Parking_Violations_Issued_-_Fiscal_Year_2023_20231115.csv

```
print("Total Rows: " , nyParkingData.count())
print("Total Columns: " , len(nyParkingData.columns))
```

```
Total Rows:  21563502
Total Columns:  43
```

Loading Data:

[17]:

```
nyParkingData = sparkObj.read.format("csv") \
    .option("header", "true") \
    .option("inferSchema", "true") \
    .load('Parking_Violations_Issued_-_Fiscal_Year_2023_20231115.csv')
```

[18]:

```
nyParkingData.show(n=2, truncate=False, vertical=True)
```

```
-RECORD 0--------------------------------------
 Summons Number                    | 1484697303
 Plate ID                          | JER1863
 Registration State                | NY
 Plate Type                        | PAS
 Issue Date                        | 06/10/2022
 Violation Code                    | 67
 Vehicle Body Type                 | SDN
 Vehicle Make                      | TOYOT
 Issuing Agency                    | P
 Street Code1                      | 34330
 Street Code2                      | 179
 Street Code3                      | 0
 Vehicle Expiration Date           | 20221210
 Violation Location                | 10
 Violation Precinct                | 10
 Issuer Precinct                   | 1
 Issuer Code                       | 160195
 Issuer Command                    | 0001
 Issuer Squad                      | 0000
 Violation Time                    | 1037A
 Time First Observed               | NULL
 Violation County                  | NY
 Violation In Front Of Or Opposite | F
 House Number                      | NULL
 Street Name                       | W 28TH ST
 Intersecting Street               | CHELSEA PK
 Date First Observed               | 0
 Law Section                       | 408
 Sub Division                      | E3
 Violation Legal Code              | NULL
 Days Parking In Effect            | BBBBBBB
 From Hours In Effect              | ALL
 To Hours In Effect                | ALL
 Vehicle Color                     | BLK
 Unregistered Vehicle?             | 0
 Vehicle Year                      | 2004
```

Dataset Schema:

[20]:
```
nyParkingData.printSchema()
```

```
root
 |-- Summons Number: long (nullable = true)
 |-- Plate ID: string (nullable = true)
 |-- Registration State: string (nullable = true)
 |-- Plate Type: string (nullable = true)
 |-- Issue Date: string (nullable = true)
 |-- Violation Code: integer (nullable = true)
 |-- Vehicle Body Type: string (nullable = true)
 |-- Vehicle Make: string (nullable = true)
 |-- Issuing Agency: string (nullable = true)
 |-- Street Code1: integer (nullable = true)
 |-- Street Code2: integer (nullable = true)
 |-- Street Code3: integer (nullable = true)
 |-- Vehicle Expiration Date: integer (nullable = true)
 |-- Violation Location: integer (nullable = true)
 |-- Violation Precinct: integer (nullable = true)
 |-- Issuer Precinct: integer (nullable = true)
 |-- Issuer Code: integer (nullable = true)
 |-- Issuer Command: string (nullable = true)
 |-- Issuer Squad: string (nullable = true)
 |-- Violation Time: string (nullable = true)
 |-- Time First Observed: string (nullable = true)
 |-- Violation County: string (nullable = true)
 |-- Violation In Front Of Or Opposite: string (nullable = true)
 |-- House Number: string (nullable = true)
 |-- Street Name: string (nullable = true)
 |-- Intersecting Street: string (nullable = true)
 |-- Date First Observed: integer (nullable = true)
 |-- Law Section: integer (nullable = true)
 |-- Sub Division: string (nullable = true)
 |-- Violation Legal Code: string (nullable = true)
 |-- Days Parking In Effect: string (nullable = true)
 |-- From Hours In Effect: string (nullable = true)
 |-- To Hours In Effect: string (nullable = true)
 |-- Vehicle Color: string (nullable = true)
 |-- Unregistered Vehicle?: integer (nullable = true)
 |-- Vehicle Year: integer (nullable = true)
 |-- Meter Number: string (nullable = true)
 |-- Feet From Curb: integer (nullable = true)
 |-- Violation Post Code: string (nullable = true)
 |-- Violation Description: string (nullable = true)
 |-- No Standing or Stopping Violation: string (nullable = true)
 |-- Hydrant Violation: string (nullable = true)
 |-- Double Parking Violation: string (nullable = true)
```

Checking Null Values in Dataset:

[22]:

```
from pyspark.sql.functions import isnan, when, count, col
nyParkingData.select([count(when(col(c).isNull(), c)).alias(c) for c in nyParkingData.columns]).show(vertical=True)
```

```
-RECORD 0------------------------------------
 Summons Number                    | 0
 Plate ID                          | 2
 Registration State                | 0
 Plate Type                        | 0
 Issue Date                        | 0
 Violation Code                    | 0
 Vehicle Body Type                 | 51924
 Vehicle Make                      | 19151
 Issuing Agency                    | 0
 Street Code1                      | 0
 Street Code2                      | 0
 Street Code3                      | 0
 Vehicle Expiration Date           | 0
 Violation Location                | 9838473
 Violation Precinct                | 0
 Issuer Precinct                   | 0
 Issuer Code                       | 0
 Issuer Command                    | 9829279
 Issuer Squad                      | 10569424
 Violation Time                    | 254
 Time First Observed               | 20321541
 Violation County                  | 128443
 Violation In Front Of Or Opposite | 9941557
 House Number                      | 10025221
 Street Name                       | 2517
 Intersecting Street               | 10142840
 Date First Observed               | 0
 Law Section                       | 0
 Sub Division                      | 3277
 Violation Legal Code              | 11734219
 Days Parking In Effect            | 10021582
 From Hours In Effect              | 14783704
 To Hours In Effect                | 14783724
 Vehicle Color                     | 1942398
 Unregistered Vehicle?             | 21125746
 Vehicle Year                      | 0
 Meter Number                      | 19034672
 Feet From Curb                    | 0
 Violation Post Code               | 11007044
 Violation Description             | 439036
 No Standing or Stopping Violation | 21563502
 Hydrant Violation                 | 21563502
 Double Parking Violation          | 21563502
```

Preprocessing and Handling Null Values:

```
[23]: nyParkingData = nyParkingData.dropna(subset=['Violation Time'])
```

```
[24]: nyParkingData = nyParkingData.dropna(subset=['Vehicle Body Type'])
```

```
[25]: nyParkingData = nyParkingData.dropna(subset=['Violation Location'])
      nyParkingData = nyParkingData.dropna(subset=['Vehicle Color'])
```

```
[26]: nyParkingData.select([count(when(col(c).isNull(), c)).alias(c) for c in nyParkingData.columns]).show(vertical=True)
```

```
-RECORD 0-----------------------------------
 Summons Number                    | 0
 Plate ID                          | 1
 Registration State                | 0
 Plate Type                        | 0
 Issue Date                        | 0
 Violation Code                    | 0
 Vehicle Body Type                 | 0
 Vehicle Make                      | 9373
 Issuing Agency                    | 0
 Street Code1                      | 0
 Street Code2                      | 0
 Street Code3                      | 0
 Vehicle Expiration Date           | 0
 Violation Location                | 0
 Violation Precinct                | 0
 Issuer Precinct                   | 0
 Issuer Code                       | 0
 Issuer Command                    | 0
 Issuer Squad                      | 737132
 Violation Time                    | 0
 Time First Observed               | 10429913
 Violation County                  | 31512
 Violation In Front Of Or Opposite | 102820
 House Number                      | 183903
 Street Name                       | 837
 Intersecting Street               | 9657967
 Date First Observed               | 0
 Law Section                       | 0
 Sub Division                      | 1388
 Violation Legal Code              | 11658506
 Days Parking In Effect            | 191133
 From Hours In Effect              | 4950319
 To Hours In Effect                | 4950339
 Vehicle Color                     | 0
 Unregistered Vehicle?             | 11287015
```

Creating View for NY Parking Dataset:

```
nyParkingData = nyParkingData.withColumn('Issue Year', psf.year(psf.to_date(nyParkingData['Issue Date'], 'MM/dd/yyyy')))
nyParkingData.createOrReplaceTempView("nyParkingDataView")
```

Question 1: When are tickets most likely to be issued?

- **Query:**
  SELECT `Violation Time`, COUNT(*) AS Ticket_Frequency FROM nyParkingDataView GROUP BY `Violation Time` ORDER BY Ticket_Frequency DESC
- **Approach:**
  I have used Violation Time to find out when were the most tickets issued.
- **Answer:**
  At 8:36 AM most tickets were issued, with exact count of 35300.

→ **Screenshot:**

Question 1: When are tickets most likely to be issued?

```
[28]: sparkObj.sql("SELECT `Violation Time`, COUNT(*) AS Ticket_Frequency FROM nyParkingDataView GROUP BY `Violation Time` ORDER BY Ticket_Frequency DESC").show()
```

```
+--------------+----------------+
|Violation Time|Ticket_Frequency|
+--------------+----------------+
|         0836A|           35300|
|         0838A|           33070|
|         0839A|           33052|
|         0840A|           32776|
|         0906A|           32194|
|         0837A|           31797|
|         0841A|           31670|
|         0842A|           31067|
|         0843A|           30451|
|         0908A|           30307|
|         0844A|           29968|
|         0845A|           29823|
|         0910A|           29692|
|         0909A|           29676|
|         0907A|           29566|
|         1140A|           29395|
|         0806A|           28904|
|         0846A|           28830|
|         1142A|           28670|
|         1141A|           28614|
+--------------+----------------+
only showing top 20 rows
```

Answer 1:

- I have sorted this according to the frequency in descending order and fetched violation time. Hence, at 0836A i.e 8:36 AM, we have maximum violators that is 35300

Question 2: What are the most common years and types of cars to be ticketed?

→ **Query:**
SELECT `Vehicle Body Type` as `Type of Car`,`Issue Year`, COUNT(*) AS `Violation_Count` FROM nyParkingDataView WHERE (`Vehicle Year` > 0) GROUP BY `Vehicle Body Type`, `Issue Year` ORDER BY `Violation_Count` DESC

→ **Approach:**
There was no column with issue year so I created a Issue Year column with the help of Issue date column and I have used Issue year column to find out common years to be ticketed and Vehicle Body Type column to find out type of cars to be ticketed.

→ **Answer:**
Most Common Year is 2023 And Type of Car is SUBN which is ticketed.

→ **Screenshot:**

Issue Year column creation:

```
]: nyParkingData = nyParkingData.withColumn('Issue Year', psf.year(psf.to_date(nyParkingData['Issue Date'], 'MM/dd/yyyy')))
```

Question 2: What are the most common years and types of cars to be ticketed?

```
sparkObj.sql("SELECT `Vehicle Body Type` as `Type of Car`,`Issue Year`, COUNT(*) AS `Violation_Count` FROM nyParkingDataView WHERE (`Vehicle Year` > 0) GROUP BY `Vehicle Body Type`, `Issue Year`
ORDER BY `Violation_Count` DESC").show(10)
```

```
+-----------+----------+---------------+
|Type of Car|Issue Year|Violation_Count|
+-----------+----------+---------------+
|       SUBN|      2023|        2248534|
|       SUBN|      2022|        1493740|
|       4DSD|      2023|        1237369|
|       4DSD|      2022|         887320|
|        VAN|      2023|         712154|
|        VAN|      2022|         489204|
|       PICK|      2023|         159446|
|       DELV|      2023|         136793|
|       PICK|      2022|         106445|
|       DELV|      2022|          98201|
+-----------+----------+---------------+
only showing top 10 rows
```

Answer 2:

- Most Common Year is 2023
- And Type of Car is SUBN

Question 3: Where are tickets most commonly issued?

- **Query:**
  SELECT `Violation Location`, COUNT(*) AS No_of_tickets FROM nyParkingDataView GROUP BY `Violation Location` ORDER BY count(*) DESC
- **Approach:**
  I have used Violation Location to find out the location where the tickets are most commonly issued.
- **Answer:**
  Location with most tickets issued is 19.
- **Screenshot:**

Question 3: Where are tickets most commonly issued

```
[30]: sparkObj.sql("SELECT `Violation Location`, COUNT(*) AS No_of_tickets  FROM nyParkingDataView GROUP BY `Violation Location` ORDER BY count(*) DESC").show(15)
```

```
+------------------+-------------+
|Violation Location|No_of_tickets|
+------------------+-------------+
|                19|       543760|
|                13|       444043|
|                 6|       422414|
|               114|       414982|
|                14|       367244|
|                18|       319307|
|                 9|       296111|
|                 1|       287019|
|               109|       270077|
|               115|       247344|
|                20|       231725|
|               108|       228404|
|                70|       217372|
|                84|       212563|
|                10|       206185|
+------------------+-------------+
only showing top 15 rows
```

Answer 3:

- Location with most tickets issued: 19

Question 4: Which color of the vehicle is most likely to get a ticket?

- **Query:**
  SELECT `Vehicle Color`, COUNT(*) as Ticket_Count FROM nyParkingDataView GROUP BY `Vehicle Color` ORDER BY COUNT(*) DESC
- **Approach:**
  I have used Vehicle colour column to find out most likely which coloured vehicle is getting the tickets.
- **Answer:**
  Vehicle Colour is WH and ticket count for this Vehicle Color: 2227522

- **Screenshot:**

Question 4: Which color of the vehicle is most likely to get a ticket ?

```
[31]: sparkObj.sql("SELECT `Vehicle Color`, COUNT(*) as Ticket_Count FROM nyParkingDataView GROUP BY `Vehicle Color` ORDER BY COUNT(*) DESC").show(15)
```

```
+-------------+------------+
|Vehicle Color|Ticket_Count|
+-------------+------------+
|           WH|     2227522|
|           GY|     2034985|
|           BK|     1734455|
|        WHITE|     1251084|
|        BLACK|      783294|
|           BL|      709865|
|         GREY|      584562|
|           RD|      401003|
|         BLUE|      281506|
|        SILVE|      273305|
|        BROWN|      269821|
|          RED|      200192|
|           GR|      144667|
|           TN|       79292|
|        OTHER|       73179|
+-------------+------------+
only showing top 15 rows
```

Answer 4:

- Vehicle Color: WH
- Ticket Count for this Vehicle Color: 2227522

Question 5: Based on a K-Means algorithm, please try to answer the following question. Given a Black vehicle parking illegally at 34510, 10030, 34050 (street codes). What is the probability that it will get a ticket?

- **Approach:**

1. Below are the possible black colours for vehicles and street codes for illegal parking:
   Input for the K-means:

```
[13]: blackColor=['BLK.', 'Black', 'BC', 'BLAC', 'BK/', 'BLK', 'BK.', 'BCK', 'BK', 'B LAC']
streetCode=[34510, 10030, 34050]
```

2. I have only kept the columns which we need:

```
nyParkingData = nyParkingData.select(nyParkingData['Street Code1'].cast('float'), nyParkingData['Street Code2'].cast('float'), nyParkingData['Street Code3'].cast('float'), nyParkingData['Vehicle Color'])
```

3. I have added a new column named "Street_codes," which is a vectorized representation of the columns "Street Code1," "Street Code2," and "Street Code3" using the Spark VectorAssembler.

```
[8]: def vectorizeStreetCodes(inputData):
#Vectorize street codes using Spark VectorAssembler.
    return VectorAssembler(inputCols=["Street Code1", "Street Code2", "Street Code3"], outputCol="Street_codes").transform(inputData)
```

4. This function initializes and fits a K-Means clustering model with k=4 clusters on a Spark DataFrame inputData that includes a "Street_codes" column, and it returns the transformed DataFrame along with the cluster centers as a NumPy array.

```
def initializeKmeans(inputData):
#Initialize and fit K-Means clustering.
    kmeans = KMeans(k=4, featuresCol="Street_codes")
    kmeansFitData = kmeans.fit(inputData.select('Street_codes'))
    clusterCenters = np.array(kmeansFitData.clusterCenters()).astype(float)
    return kmeansFitData.transform(inputData).cache(),clusterCenters
```

5. This function calculates the probability of having black-colored vehicles in each cluster based on a Spark DataFrame inputData containing a 'prediction' column and a 'Vehicle Color' column, using the specified black colours in the blackColor list.

```
: def calculateBlackProbability(inputData, blackColor):

    blackProb = inputData.groupBy('prediction').agg(
        psf.sum(psf.when(psf.col('Vehicle Color').isin(blackColor), 1)).alias('Count'),
        psf.count('Vehicle Color').alias('Total_Cars')
    ).orderBy('prediction')

    return blackProb.select(
        'prediction',
        'Count',
        'Total_Cars',
        (psf.col('Count') / psf.col('Total_Cars')).alias('Probability')
    )
```

6. This function finds the cluster with the closest center to a specified set of streets by calculating the Euclidean distance between each cluster center and the streets, then returning the index of the closest cluster.

```
def findClosestCluster(streetsData, clusterCenters):
# Find the cluster with the closest center to specified streets.

    closestDistance = float("inf")
    clusterCentreID = 0

    for index in range(len(clusterCenters)):
        newDist = np.sum(np.power((np.array(streetsData) - clusterCenters[index]), 2))
        if newDist < closestDistance:
            closestDistance = newDist
            clusterCentreID = index

    return clusterCentreID
```

7. Printing the cluster ID and Probability of that specific cluster using below function printClusterProb and calculatePrintProbability is the main function to call all the functions.

```
[11]:
def printClusterProb(clusterCentreID, prob):
# Print the probability for the specified cluster
    print('Cluster ID for given Street Code (34510, 10030, 34050):', clusterCentreID)
    print("-----------------------------------------------------------------------")
    print("Probability for that Specific Cluster:")
    prob.filter(psf.col('prediction') == clusterCentreID).show()
```

```
[12]:
def calculatePrintProbability(inputData, blackColor, streetCode):
# Main function to calculate and print probability.
    inputData = vectorizeStreetCodes(inputData)
    kmeansFitData, clusterCenters= initializeKmeans(inputData)

    prob = calculateBlackProbability(kmeansFitData, blackColor)

    clusterCentreID = findClosestCluster(streetCode, clusterCenters)

    # Print the probability for the specified cluster
    printClusterProb(clusterCentreID, prob)
```

8. Below is the function call and out of the Kmeans.

```
calculatePrintProbability(nyParkingData,blackColor,streetCode )
```

```
Cluster ID for given Street Code (34510, 10030, 34050): 0
---------------------------------------------------------------------------
Probability for that Specific Cluster:
+----------+------+---------+-------------------+
|prediction| Count|Total_Cars|        Probability|
+----------+------+---------+-------------------+
|         0|856969|   5863372|0.14615634143629297|
+----------+------+---------+-------------------+
```

→ Answer: For K value 4 I have built the model and probability is **0.14615634143629297.**

Output:

```
Cluster ID for given Street Code (34510, 10030, 34050): 0
-----------------------------------------------------------------------
Probability for that Specific Cluster:
+----------+------+----------+-------------------+
|prediction| Count|Total_Cars|        Probability|
+----------+------+----------+-------------------+
|         0|856969|   5863372|0.14615634143629297|
+----------+------+----------+-------------------+
```

Part 2: NBA Shot Logs:

Data File: shot_logs.csv

Spark Session:

3]:
```
sparkObj = pyspark.sql.SparkSession.builder.appName("Assignment2_NBA").getOrCreate()
sparkObj.conf.set("spark.sql.repl.eagerEval.enabled", True)
```

4]:
```
sparkObj
```

4]: **SparkSession - in-memory**

SparkContext

Spark UI

Version	`v3.5.0`
Master	`local[*]`
AppName	`Assignment2_NBA`

Loading the Data:

```
[5]: nbaData = sparkObj.read.format("csv").option("header", "true",).option("inferSchema","true").load('shot_logs.csv')
```

```
[6]: nbaData.show(n=2,vertical=True)
```

```
-RECORD 0------------------------------------------
 GAME_ID                    | 21400899
 MATCHUP                    | MAR 04, 2015 - CH...
 LOCATION                   | A
 W                          | W
 FINAL_MARGIN               | 24
 SHOT_NUMBER                | 1
 PERIOD                     | 1
 GAME_CLOCK                 | 2023-11-18 01:09:00
 SHOT_CLOCK                 | 10.8
 DRIBBLES                   | 2
 TOUCH_TIME                 | 1.9
 SHOT_DIST                  | 7.7
 PTS_TYPE                   | 2
 SHOT_RESULT                | made
 CLOSEST_DEFENDER           | Anderson, Alan
 CLOSEST_DEFENDER_PLAYER_ID | 101187
 CLOSE_DEF_DIST             | 1.3
 FGM                        | 1
 PTS                        | 2
 player_name                | brian roberts
 player_id                  | 203148
-RECORD 1------------------------------------------
 GAME_ID                    | 21400899
 MATCHUP                    | MAR 04, 2015 - CH...
 LOCATION                   | A
 W                          | W
 FINAL_MARGIN               | 24
 SHOT_NUMBER                | 2
 PERIOD                     | 1
 GAME_CLOCK                 | 2023-11-18 00:14:00
 SHOT_CLOCK                 | 3.4
 DRIBBLES                   | 0
 TOUCH_TIME                 | 0.8
 SHOT_DIST                  | 28.2
 PTS_TYPE                   | 3
 SHOT_RESULT                | missed
 CLOSEST_DEFENDER           | Bogdanovic, Bojan
 CLOSEST_DEFENDER_PLAYER_ID | 202711
 CLOSE_DEF_DIST             | 6.1
 FGM                        | 0
 PTS                        | 0
 player_name                | brian roberts
```

Dataset Schema:

```
[7]: print("Total Rows: " , nbaData.count())
     print("Total Columns: " , len(nbaData.columns))
```

```
Total Rows:  128069
Total Columns:  21
```

```
[8]: nbaData.printSchema()
```

```
root
 |-- GAME_ID: integer (nullable = true)
 |-- MATCHUP: string (nullable = true)
 |-- LOCATION: string (nullable = true)
 |-- W: string (nullable = true)
 |-- FINAL_MARGIN: integer (nullable = true)
 |-- SHOT_NUMBER: integer (nullable = true)
 |-- PERIOD: integer (nullable = true)
 |-- GAME_CLOCK: timestamp (nullable = true)
 |-- SHOT_CLOCK: double (nullable = true)
 |-- DRIBBLES: integer (nullable = true)
 |-- TOUCH_TIME: double (nullable = true)
 |-- SHOT_DIST: double (nullable = true)
 |-- PTS_TYPE: integer (nullable = true)
 |-- SHOT_RESULT: string (nullable = true)
 |-- CLOSEST_DEFENDER: string (nullable = true)
 |-- CLOSEST_DEFENDER_PLAYER_ID: integer (nullable = true)
 |-- CLOSE_DEF_DIST: double (nullable = true)
 |-- FGM: integer (nullable = true)
 |-- PTS: integer (nullable = true)
 |-- player_name: string (nullable = true)
 |-- player_id: integer (nullable = true)
```

Question 1:

For each pair of the players (A, B), we define the fear sore of A when facing B is the hit rate, such that B is closet defender when A is shooting. Based on the fear sore, for each player, please find out who is his "most unwanted defender".

Approach:

1. In below, two conditions, madeCond and missedCond, are defined to represent whether a basketball shot was made or missed. The NBA dataset is then grouped by player and closest defender, and a new DataFrame, unwantedDf, is created to aggregate the counts of made and missed shots for each player and defender pair, providing insights into their scoring performance.

```
[9]: madeCond= psf.when(psf.col("SHOT_RESULT") == "made", 1).otherwise(0)
missedCond = psf.when(psf.col("SHOT_RESULT") == "missed", 1).otherwise(0)

unwantedDf = nbaData.groupBy(
    psf.col("player_id").alias("Player ID"),
    psf.col("CLOSEST_DEFENDER_PLAYER_ID").alias("Defender ID")
).agg(
    psf.sum(madeCond).alias("Scored"),
    psf.sum(missedCond).alias("Not Scored")
)


unwantedDf.show(10)
```

```
+---------+-----------+------+----------+
|Player ID|Defender ID|Scored|Not Scored|
+---------+-----------+------+----------+
|   203148|     101179|     0|         1|
|   202687|     201980|     1|         0|
|     2744|       1717|     0|         2|
|   203469|     202329|     1|         1|
|   201945|     202322|     0|         3|
|   202689|     202699|     6|         8|
|   202689|     203924|     1|         0|
|   203077|       2730|     1|         0|
|   203077|     201584|     2|         0|
|   202362|     201188|     2|         0|
+---------+-----------+------+----------+
only showing top 10 rows
```

2. In below a new column "HitRate" is added to the DataFrame unwantedDf, representing the ratio of made shots to the total shots (made plus missed), providing a measure of scoring efficiency for each player and defender pair.

```
[10]: unwantedDf = unwantedDf.withColumn(
    "HitRate",
    psf.col("Scored") / (psf.col("Scored") + psf.col("Not Scored"))
)
unwantedDf.show(10)
```

```
+---------+-----------+------+----------+------------------+
|Player ID|Defender ID|Scored|Not Scored|           HitRate|
+---------+-----------+------+----------+------------------+
|   203148|     101179|     0|         1|               0.0|
|   202687|     201980|     1|         0|               1.0|
|     2744|       1717|     0|         2|               0.0|
|   203469|     202329|     1|         1|               0.5|
|   201945|     202322|     0|         3|               0.0|
|   202689|     202699|     6|         8|0.42857142857142855|
|   202689|     203924|     1|         0|               1.0|
|   203077|       2730|     1|         0|               1.0|
|   203077|     201584|     2|         0|               1.0|
|   202362|     201188|     2|         0|               1.0|
+---------+-----------+------+----------+------------------+
only showing top 10 rows
```

3. In first step it filters out rows in the DataFrame unwantedDf where the "HitRate" column is not null, removes duplicate rows based on "Player ID" and "HitRate," calculates the minimum "HitRate" for each player, and then joins this information back to the original DataFrame. After joining with the NBA dataset (nbaData), it creates a new DataFrame unwantedDf containing information about the most unwanted defender for each player. Finally, it drops duplicate records based on "Player ID" and "Defender ID" and selects and displays the columns "Player Name" and "Most Unwanted Defender" for the first 10 rows.

```
[11]: unwantedDf = unwantedDf.filter(psf.col("HitRate").isNotNull())
```

```
[12]: unwantedDf = unwantedDf.dropDuplicates(subset=["Player ID", "HitRate"])
```

```
[13]: finalDf = unwantedDf.groupBy("Player ID").agg(psf.min("HitRate").alias("HitRate"))
```

```
[14]: unwantedDf = unwantedDf.join(finalDf, ["Player ID", "HitRate"]).select("Player ID", "Defender ID")


unwantedDf = unwantedDf.join(
    nbaData,
    (nbaData["player_id"] == unwantedDf["Player ID"]) & (nbaData["CLOSEST_DEFENDER_PLAYER_ID"] == unwantedDf["Defender ID"])
).withColumn("Player Name", col("player_name")).withColumn("Most Unwanted Defender", col("CLOSEST_DEFENDER"))

unwantedDf = unwantedDf.dropDuplicates(["Player ID", "Defender ID"])

unwantedDf.select("Player Name", "Most Unwanted Defender").show(10)
```

```
+-------------+----------------------+
|  Player Name|Most Unwanted Defender|
+-------------+----------------------+
| kevin garnett|          Exum, Dante|
|   kobe bryant|      Anderson, Kyle|
|    tim duncan|      Roberts, Brian|
|  vince carter|     Crawford, Jamal|
|dirk nowtizski|          Hickson, JJ|
|    paul pierce|       Waiters, Dion|
|   andre miller|     Splitter, Tiago|
|   shawn marion|   Tolliver, Anthony|
|    jason terry|         Lopez, Brook|
|  manu ginobili|    Bennett, Anthony|
+-------------+----------------------+
only showing top 10 rows
```

4. Final Output:

- From below final output we can see top 10 players with their most unwanted defenders.
- Let’s say if Kevin Garnett is the shooter, the most the unwanted defender is the Exum, Dante.

```
+-------------+----------------------+
|  Player Name|Most Unwanted Defender|
+-------------+----------------------+
| kevin garnett|          Exum, Dante|
|   kobe bryant|      Anderson, Kyle|
|    tim duncan|      Roberts, Brian|
|  vince carter|     Crawford, Jamal|
|dirk nowtizski|          Hickson, JJ|
|    paul pierce|       Waiters, Dion|
|   andre miller|     Splitter, Tiago|
|   shawn marion|   Tolliver, Anthony|
|    jason terry|         Lopez, Brook|
|  manu ginobili|    Bennett, Anthony|
+-------------+----------------------+
only showing top 10 rows
```

Question 2: For each player, we define the comfortable zone of shooting is a matrix of, {SHOT DIST, CLOSE DEF DIST, SHOT CLOCK} Please develop a Spark-based algorithm to classify each player's records into 4 comfortable zones. Considering the hit rate, which zone is the best for James Harden, Chris Paul, Stephen Curry, and Lebron James.

Spark Sesson Creation and Loading the data:

```
sparkObj = pyspark.sql.SparkSession.builder.appName("Assignment2_NBA_2").getOrCreate()
sparkObj.conf.set("spark.sql.repl.eagerEval.enabled", True)
```

```
nbaData = sparkObj.read.format("csv").option("header", "true",).option("inferSchema","true").load('shot_logs.csv').select("player_name","SHOT_DIST","CLOSE_DEF_DIST", "SHOT_CLOCK", "SHOT_RESULT").na.drop()
```

Approach:

1. It selects specific columns ("player_name," "SHOT_DIST," "CLOSE_DEF_DIST," "SHOT_CLOCK," and "SHOT_RESULT") and drops rows with missing values. The "SHOT_RESULT" column is then transformed into a binary float column (1 for 'made' shots, 0 for 'missed' shots). The list comfortableZoneMat is created, specifying columns to be used as features for later analysis, such as shot distance, defender distance, and shot clock time.

```
nbaData = sparkObj.read.format("csv").option("header", "true",).option("inferSchema","true").load('shot_logs.csv').select("player_name","SHOT_DIST","CLOSE_DEF_DIST", "SHOT_CLOCK", "SHOT_RESULT").na.drop()
```

```
nbaShotsData = nbaData.withColumn('SHOT_RESULT', psf.when(psf.col('SHOT_RESULT') == 'made', 1).otherwise(0).cast('float'))
comfortableZoneMat = ["SHOT_DIST", "CLOSE_DEF_DIST", "SHOT_CLOCK"]
```

2. Below loop iterates over the columns specified in comfortableZoneMat for the PySpark DataFrame nbaShotsData and casts each column to a float type.

```
for feature in comfortableZoneMat:
    nbaShotsData = nbaShotsData.withColumn(feature, psf.col(feature).cast("float"))
```

3. It uses VectorAssembler to combine the specified features (comfortableZoneMat) into a single vector column named "shooting_zone" for each player in the nbaShotsData DataFrame. It initializes a K-Means clustering model with k=4 clusters and fits the model to the transformed data, creating kmeansFitData. It filters the player data for specific players ('james harden', 'chris paul', 'stephen curry', 'lebron james'). It uses the fitted K-Means model to predict the cluster assignments for the selected players, resulting in a DataFrame pred containing player names, predicted cluster labels, and shot results.

```
vecAssembler = VectorAssembler(inputCols=comfortableZoneMat, outputCol="shooting_zone")
nbaShotsData = vecAssembler.transform(nbaShotsData).select('player_name', 'shooting_zone', 'SHOT_RESULT')


kmeans = KMeans(k=4, featuresCol="shooting_zone")
kmeansFitData = kmeans.fit(nbaShotsData)

playersData = nbaShotsData.filter(nbaShotsData['player_name'].isin(['james harden', 'chris paul', 'stephen curry', 'lebron james']))

pred = kmeansFitData.transform(playersData).select('player_name', 'prediction', 'SHOT_RESULT')
```

4. Below code is calculating the average shot result for each player and cluster prediction, ordering the results. It then identifies the maximum average shot result for each player,

performing a join to pinpoint the rows where the average is the highest. The final DataFrame, named bestZone, retains all columns from the original DataFrame, and the results are displayed.

```
]: pred.createOrReplaceTempView("player_zones")
```

```
]:

avgShotResultQuery = """SELECT player_name, prediction, AVG(SHOT_RESULT) AS avgShotResult FROM player_zones GROUP BY player_name, prediction ORDER BY player_name, prediction """
res = sparkObj.sql(avgShotResultQuery)

maxAvgShot = res.groupBy("player_name").agg(psf.max("avgShotResult").alias("maxAvgShotResult"))

bestZone = res.alias("df1").join(maxAvgShot.alias("df2"), (psf.col("df1.player_name") == psf.col("df2.player_name")) & (psf.col("df1.avgShotResult") == psf.col("df2.maxAvgShotResult")))

bestZone = bestZone.select("df1.*")
bestZone.show()
```

5. Final Output:

- Zone-1 corresponds to a prediction value of 0, Zone-2 to 1, and Zone-3 to 2 and Zone 4 to 3 in the 'prediction' column.
- To determine each player's comfort zone, we grouped the data by player and zone, calculating the average score for each group.
- As seen in the table Lebron, James, Stephen has best Zone 1 and Chris has best Zone 4.

```
+-------------+----------+------------------+
|  player_name|prediction|     avgShotResult|
+-------------+----------+------------------+
| lebron james|         3|0.6613545816733067|
|   chris paul|         0|0.5563380281690141|
| james harden|         3|0.5604395604395604|
|stephen curry|         3|0.6350710900473934|
+-------------+----------+------------------+
```