Saamia Shafqat 3rd December 2023

Analysis of Electricity Demand, Consumption, and Electric Vehicle Trends Across Regions from 2016 to 2021

Processing Big Data for Analytics Applications
Fall 2023

Cleaning Data:

Reading csv file and storing in a dataframe

```
scala> def readCSV(path: String): DataFrame = {
     |   // Create a SparkSession and read CSV file into a DataFrame
     |   val df = spark.read
     |     .format("csv")
     |     .option("header", true)
     |     .option("inferSchema", true)
     |     .load(path)
     |
     |   // Return the DataFrame
     |   df
     | }
readCSV: (path: String)org.apache.spark.sql.DataFrame

scala>

scala> // Read the CSV file

scala> val filePath = "hw8/EV_and_RegActivity.csv"
filePath: String = hw8/EV_and_RegActivity.csv

scala> var eV_data = readCSV(filePath)
```

Trimming unnecessary columns from the dataset, retaining only vital ones for subsequent analysis. The original dataset encompassed 35 columns, but now it comprises only 9 essential columns. Additionally, eliminating any instances of null values, converting the Year to a date type format, and transforming the odometer reading to a double type format to enhance precision.

```
scala> // Select relevant columns

scala> eV_data = eV_data.select(
     |   "Model Year", "Make", "Model", "Electric Range", "Odometer Reading",
     |   "Odometer Code", "New or Used Vehicle", "Transaction Year", "State of Residence"
     | )
eV_data: org.apache.spark.sql.DataFrame = [Model Year: int, Make: string ... 7 more fields]

scala>

scala> // Drop rows with null values in any column

scala> eV_data = eV_data.na.drop()
eV_data: org.apache.spark.sql.DataFrame = [Model Year: int, Make: string ... 7 more fields]

scala>

scala> // Additional cleaning steps

scala> eV_data = eV_data.withColumn("Odometer Reading", col("Odometer Reading").cast(DoubleType))
eV_data: org.apache.spark.sql.DataFrame = [Model Year: int, Make: string ... 7 more fields]
```

Associating states with regions for the purpose of merging the three datasets subsequently. Introducing a new column that includes the region information.

```
scala> // State to region mapping

scala> val stateToRegion = Map(
     |   "WA" -> "NW", "CA" -> "CAL", "UT" -> "SW", "VA" -> "SE", "OR" -> "NW", "TX" -> "TEX", "MI" -> "MIDW",
     |   "TN" -> "TEN", "DE" -> "MIDA", "MN" -> "MIDW", "HI" -> "SW", "ID" -> "NW", "AL" -> "SE", "IN" -> "MIDW",
     |   "WY" -> "NW", "GA" -> "SE", "FL" -> "FLA", "KY" -> "SE", "DC" -> "MIDA", "MD" -> "MIDA", "AZ" -> "SW",
     |   "NE" -> "MIDW", "LA" -> "SE", "PA" -> "NE", "NC" -> "CAR", "NY" -> "NY", "SD" -> "MIDW", "AK" -> "NW",
     |   "NM" -> "SW", "MO" -> "MIDW", "CO" -> "SW", "IL" -> "MIDW", "NV" -> "SW", "SC" -> "CAR", "MS" -> "SE",
     |   "MT" -> "NW", "ND" -> "MIDW", "OH" -> "MIDW", "MA" -> "NE", "CT" -> "NE", "NJ" -> "NE", "WI" -> "MIDW",
     |   "IA" -> "MIDW", "KS" -> "CENT", "OK" -> "SW", "AR" -> "SE", "NH" -> "NE", "ME" -> "NE"
     | )
stateToRegion: scala.collection.immutable.Map[String,String] = Map(MA -> NE, IN -> MIDW, ID -> NW, NM -> SW, OR -> NW, IA -> MIDW, IL -> MIDW, TN -> TEN, MO -> MIDW, ME -> NE, AZ -
> SW, AK -> NW, WA -> NW, SD -> MIDW, KY -> SE, NJ -> NE, TX -> TEX, MI -> MIDW, MD -> MIDA, NV -> SW, NE -> MIDW, MN -> MIDW, KS -> CENT, OK -> SW, CT -> NE, OH -> MIDW, AR -> SE,
 FL -> FLA, WI -> MIDW, CO -> SW, MT -> NW, DC -> MIDA, ND -> MIDW, PA -> NE, GA -> SE, NH -> NE, HI -> SW, WY -> NW, LA -> SE, CA -> CAL, UT -> SW, AL -> SE, VA -> SE, NC -> CAR,
NY -> NY, SC -> CAR, MS -> SE, DE -> MIDA)

scala>

scala> // User-defined function (UDF) for mapping state to region

scala> val mapStateToRegion = udf((state: String) => stateToRegion.getOrElse(state, "Unknown"))
mapStateToRegion: org.apache.spark.sql.expressions.UserDefinedFunction = SparkUserDefinedFunction($Lambda$4156/77763873@7ef1a064,StringType,List(Some(class[value[0]: string])),Some
(class[value[0]: string]),None,true,true)

scala>

scala> // Add 'Region' column to DataFrame

scala> val cleaned_data = eV_data.withColumn("Region", mapStateToRegion($"State of Residence"))
cleaned_data: org.apache.spark.sql.DataFrame = [Model Year: int, Make: string ... 8 more fields]
```

For our collaborative analysis, I isolated data values from 2016 to 2021 and organized them by consolidating all electric vehicle transactions within each region for each year. Additionally, I computed the figures for both new and old electric vehicles purchased in each region per year and determined the respective percentages.

```
scala> // For joint analysis

scala> val filteredDF = cleaned_data.filter(col("Transaction Year").between(2016, 2021))
filteredDF: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [Model Year: int, Make: string ... 8 more fields]

scala>

scala> // Filter out rows with "Unknown" region

scala> val intermediateDF = filteredDF.filter(col("Region") =!= "Unknown")
intermediateDF: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [Model Year: int, Make: string ... 8 more fields]

scala>

scala> // Count occurrences of each region for each year

scala> val regionCounts = intermediateDF.groupBy("Region", "Transaction Year").agg(count("Region").alias("EV_adoption_count"),sum(when(col("New or Used Vehicle") === "New", 1).othe
rwise(0)).alias("new_count"),sum(when(col("New or Used Vehicle") === "Used", 1).otherwise(0)).alias("used_count"))
regionCounts: org.apache.spark.sql.DataFrame = [Region: string, Transaction Year: int ... 3 more fields]

scala>

scala> // Calculate percentages for new and used cars

scala> val resultDF = regionCounts.withColumn("new_percentage", col("new_count") / col("EV_adoption_count") * 100).withColumn("used_percentage", col("used_count") / col("EV_adoptio
n_count") * 100).orderBy("Region", "Transaction Year")
resultDF: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [Region: string, Transaction Year: int ... 5 more fields]

scala>

scala> resultDF.show()
+------+----------------+----------------+---------+---------+-----------------+-----------------+
|Region|Transaction Year|EV_adoption_count|new_count|used_count|    new_percentage|   used_percentage|
+------+----------------+----------------+---------+---------+-----------------+-----------------+
|   CAL|            2016|              15|        6|        9|             40.0|             60.0|
|   CAL|            2017|              35|        9|       26| 25.71428571428571| 74.28571428571429|
|   CAL|            2018|              63|       24|       39|38.095238095238095|61.904761904761905|
|   CAL|            2019|              88|       27|       61|30.681818181818183| 69.31818181818183|
|   CAL|            2020|              99|       21|       78| 21.21212121212121| 78.78787878787878|
|   CAL|            2021|             104|       24|       80|23.076923076923077| 76.92307692307693|
|   CAR|            2016|               5|        0|        5|              0.0|            100.0|
|   CAR|            2017|               2|        0|        2|              0.0|            100.0|
|   CAR|            2018|               8|        5|        3|             62.5|             37.5|
|   CAR|            2019|              12|        3|        9|             25.0|             75.0|
|   CAR|            2020|              10|        1|        9|             10.0|             90.0|
|   CAR|            2021|              24|        7|       17|29.166666666666668| 70.83333333333334|
```

Profiling Data:

Dataframe Schema:

```
scala> println("DataFrame Schema:")
DataFrame Schema:

scala> profiled_data.printSchema()
root
 |-- Model Year: integer (nullable = true)
 |-- Make: string (nullable = true)
 |-- Model: string (nullable = true)
 |-- Electric Range: integer (nullable = true)
 |-- Odometer Reading: double (nullable = true)
 |-- Odometer Code: string (nullable = true)
 |-- New or Used Vehicle: string (nullable = true)
 |-- Transaction Year: integer (nullable = true)
 |-- State of Residence: string (nullable = true)
 |-- Region: string (nullable = true)
```

Shape of the dataset:

```
scala>

scala> println("\nNumber of Rows and Columns in the DataFrame:")

Number of Rows and Columns in the DataFrame:

scala> println(s"Rows: ${profiled_data.count()}, Columns: ${profiled_data.columns.length}")
Rows: 832177, Columns: 10
```

Count of Null Values:

```
scala> println("\nCount of null values in each column:")

Count of null values in each column:

scala> profiled_data.select(profiled_data.columns.map(c => sum(col(c).isNull.cast("int")).alias(c)): _*).show()
+----------+----+-----+--------------+----------------+-------------+-------------------+----------------+------------------+------+
|Model Year|Make|Model|Electric Range|Odometer Reading|Odometer Code|New or Used Vehicle|Transaction Year|State of Residence|Region|
+----------+----+-----+--------------+----------------+-------------+-------------------+----------------+------------------+------+
|         0|   0|    0|             0|               0|            0|                  0|               0|                 0|     0|
+----------+----+-----+--------------+----------------+-------------+-------------------+----------------+------------------+------+
```

Upon scrutinizing the 'Odometer Reading' column, it was observed that 68% of the readings are zero. This substantial occurrence of zero readings could significantly influence the overall mean. Consequently, during the profiling process, a decision was made to exclude both zero readings and readings of new cars. The aim was to investigate purchase

decisions related to the odometer readings of old cars, particularly when individuals are making choices regarding electric vehicle (EV) purchases. This choice was motivated by the recognition that, unlike traditional cars where odometer readings are crucial indicators, EVs don't experience wear and tear in components like brakes. The analysis sought to discern whether analogous decision patterns exist in the context of EV purchases.

```
scala> // Analyze how many 'Odometer Reading' values are equal to 0

scala> val zeroOdometerCount = profiled_data.filter(col("Odometer Reading") === 0).count()
zeroOdometerCount: Long = 567800

scala> println(s"\nNumber of rows with 'Odometer Reading' equal to 0: $zeroOdometerCount")

Number of rows with 'Odometer Reading' equal to 0: 567800

scala>

scala> // Calculate total 'Odometer Reading' count

scala> val totalOdometerCount = profiled_data.filter(col("Odometer Reading").isNotNull).count()
totalOdometerCount: Long = 832177

scala> println(s"\nTotal 'Odometer Reading' count: $totalOdometerCount")

Total 'Odometer Reading' count: 832177

scala>

scala> // Calculate percentage of 'Odometer Reading' values equal to 0

scala> val percentageZeroOdometer = (zeroOdometerCount.toDouble / totalOdometerCount) * 100
percentageZeroOdometer: Double = 68.23067688724875

scala> println(f"\nPercentage of 'Odometer Reading' equal to 0: $percentageZeroOdometer%.2f%%")

Percentage of 'Odometer Reading' equal to 0: 68.23%
```

Analysis:

Function for calculating statistics for a numerical column ‘calculateStatistics( )’ was used to find the statistics for odometer reading column and the mean is 2889 which is less because the 0 vales are not excluded when the mean is calculated.

```
scala> def calculateStatistics(inputDF: DataFrame, columnName: String): String = {
     |   val transformedDF = inputDF
     |     .withColumn(columnName, col(columnName).cast(DoubleType))
     |     .na.drop(Seq(columnName))
     |
     |   val meanValue = transformedDF.select(mean(col(columnName))).as[Double].first()
     |   val medianValue = transformedDF.stat.approxQuantile(columnName, Array(0.5), 0.01)(0)
     |   val modeValue = transformedDF.groupBy(col(columnName))
     |     .agg(count(col(columnName)).alias("count"))
     |     .sort(desc("count"))
     |     .limit(1)
     |     .select(col(columnName))
     |     .collect()
     |     .map(_.get(0).toString)
     |     .mkString(", ")
     |   val stddevValue = transformedDF.select(stddev(col(columnName))).as[Double].first()
     |
     |   s"Column: $columnName, Mean: $meanValue, Median: $medianValue, Mode: $modeValue, Standard Deviation: $stddevValue"
     | }
calculateStatistics: (inputDF: org.apache.spark.sql.DataFrame, columnName: String)String

scala>

scala> calculateStatistics(EV_data, "Odometer Reading")
res0: String = Column: Odometer Reading, Mean: 2889.358351648748, Median: 0.0, Mode: 0.0, Standard Deviation: 11876.815331088157
```

The mean value after filtering the 0s column and eliminating the ‘New’ cars rises to 27173 from 2889. The binary column is made which checks if the reading of the mean value is greater than or less than the mean value.

```
scala> val meanValue: Double = EV_data.filter(col("Odometer Reading").isNotNull && col("Odometer Reading") =!= 0 && col("New or Used Vehicle") === "Used").agg(mean(col("Odometer Re
ading").cast("double")).alias("mean_odometer")).head().getAs[Double](0)
meanValue: Double = 27173.837339798312

scala>

scala> //checking of odometer reading is greater than mean value

scala> var dFwithBinary: DataFrame = EV_data.withColumn("binaryColumn", when(col("Odometer Reading").cast("double") > meanValue, 1).otherwise(0))
dFwithBinary: org.apache.spark.sql.DataFrame = [Model Year: int, Make: string ... 9 more fields]

scala>

scala> dFwithBinary.show()
+----------+-----+-------+--------------+----------------+--------------------+--------------------+---------------+-----------------+------+------------+
|Model Year| Make|  Model|Electric Range|Odometer Reading|       Odometer Code|New or Used Vehicle|Transaction Year|State of Residence|Region|binaryColumn|
+----------+-----+-------+--------------+----------------+--------------------+--------------------+---------------+-----------------+------+------------+
|      2013| TESLA|Model S|           208|         15500.0|      Actual Mileage|                Used|           2016|               WA|    NW|           0|
|      2014| TESLA|Model S|           208|         16331.0|      Actual Mileage|                Used|           2015|               WA|    NW|           0|
|      2013| TESLA|Model S|           208|         26629.0|      Actual Mileage|                Used|           2016|               WA|    NW|           0|
|      2013| TESLA|Model S|           208|             0.0|Odometer reading ...|                Used|           2020|               WA|    NW|           0|
|      2014| TESLA|Model S|           208|         20032.0|      Actual Mileage|                Used|           2016|               WA|    NW|           0|
|      2011|NISSAN|   Leaf|            73|             5.0|      Actual Mileage|                 New|           2011|               WA|    NW|           0|
|      2011|NISSAN|   Leaf|            73|             5.0|      Actual Mileage|                 New|           2011|               WA|    NW|           0|
|      2014| TESLA|Model S|           208|          4673.0|      Actual Mileage|                Used|           2015|               WA|    NW|           0|
|      2013| TESLA|Model S|           208|             0.0|Odometer reading ...|                Used|           2018|               WA|    NW|           0|
|      2014| TESLA|Model S|           208|         30510.0|      Actual Mileage|                Used|           2016|               WA|    NW|           1|
|      2013| TESLA|Model S|           208|         31292.0|      Actual Mileage|                Used|           2016|               WA|    NW|           1|
|      2016| TESLA|Model S|           210|          4159.0|      Actual Mileage|                Used|           2016|               WA|    NW|           0|
|      2013| TESLA|Model S|           208|         26629.0|      Actual Mileage|                Used|           2016|               WA|    NW|           0|
|      2014| TESLA|Model S|           208|             0.0|Odometer reading ...|                Used|           2019|               WA|    NW|           0|
|      2014| TESLA|Model S|           208|         20032.0|      Actual Mileage|                Used|           2016|               WA|    NW|           0|
|      2013| TESLA|Model S|           208|         22791.0|      Actual Mileage|                Used|           2016|               WA|    NW|           0|
|      2014| TESLA|Model S|           208|         23868.0|      Actual Mileage|                Used|           2016|               WA|    NW|           0|
|      2013| TESLA|Model S|           208|         22791.0|      Actual Mileage|                Used|           2016|               WA|    NW|           0|
|      2012| TESLA|Model S|           265|             0.0|Odometer reading ...|                Used|           2017|               WA|    NW|           0|
|      2013| TESLA|Model S|           208|             0.0|Odometer reading ...|                Used|           2017|               WA|    NW|           0|
+----------+-----+-------+--------------+----------------+--------------------+--------------------+---------------+-----------------+------+------------+
only showing top 20 rows
```

The high value of 93% is found which means most purchases of EV cars are higher than the mean value of Odometer Reading.

```
scala> val totalRows = dFWithBinary.filter(col("New or Used Vehicle") === "Used").count()
totalRows: Long = 522175

scala> val percentageDF: DataFrame = countsDF.withColumn("percentage", (col("count") / totalRows * 100).cast("double"))
percentageDF: org.apache.spark.sql.DataFrame = [binaryColumn: int, count: bigint ... 1 more field]

scala>

scala> println(s"The percentage of cars purchased with high odometer range is percentage of binary column 1")
The percentage of cars purchased with high odometer range is percentage of binary column 1

scala> println(s"The percentage of cars purchased with low odometer range is percentage of binary column 0")
The percentage of cars purchased with low odometer range is percentage of binary column 0

scala>

scala> percentageDF.show()
+------------+------+-----------------+
|binaryColumn| count|       percentage|
+------------+------+-----------------+
|           1| 35096|6.721118399004165|
|           0|487079|93.27888160099583|
+------------+------+-----------------+
```