FirstJob

July 7, 2025

1 Big Data project A.Y. 2024-2025 - First Job

1.1 Members

- Giovanni Antonioni
- Luca Rubboli - 0001083742

[1]:

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession.builder
  .appName("First job")
  .getOrCreate()

val sc = spark.sparkContext
```

```
Intitializing Scala interpreter …

Spark Web UI available at http://10.201.101.147:4042
SparkContext available as 'sc' (version = 3.5.1, master = local[*], app id =␣
 ↪local-1751894105603)
SparkSession available as 'spark'
```

[1]:

```
import org.apache.spark.sql.SparkSession
spark: org.apache.spark.sql.SparkSession =
org.apache.spark.sql.SparkSession@3cea046f
sc: org.apache.spark.SparkContext = org.apache.spark.SparkContext@50be5645
```

1.1.1 Define useful parameters

- Dataset location
- Directories
- Iterator (defined like this to overcome different names for same columns in dataset)

[2]:

```
val decimals: Int = 4
val minimumYearDataset = 2024
val projectDir: String = "/Users/luca/Desktop/Luca/Università/Magistrale/Corsi/
 ↪BigData/Drivers"
val datasetDir = "dataset"
val outputDir = "output/firstJobOutput"
val yellowDatasetDir = s"$datasetDir/yellow_cab"
val greenDatasetDir = s"$datasetDir/green_cab"
val fhvDatasetDir = s"$datasetDir/fhv_cab"
val fhvhvDatasetDir = s"$datasetDir/fhvhv_cab"
val datasetDirMap: Map[String, String] = Map("yellow" -> yellowDatasetDir,␣
 ↪"green" -> greenDatasetDir, "fhv" -> fhvDatasetDir, "fhvhv" ->␣
 ↪fhvhvDatasetDir)
val datasetIterator: Map[String, (String, String)] = Map(
  "yellow" -> ("tpep_dropoff_datetime", "tpep_pickup_datetime"),
  "green" -> ("lpep_dropoff_datetime", "lpep_pickup_datetime"),
  // ("fhv", "tpep_dropoff_datetime", "tpep_pickup_datetime"),
  // ("fhvhv", "tpep_dropoff_datetime", "tpep_pickup_datetime"),
)
```

[2]:

```
decimals: Int = 4
minimumYearDataset: Int = 2024
projectDir: String =
/Users/luca/Desktop/Luca/Università/Magistrale/Corsi/BigData/Drivers
datasetDir: String = dataset
outputDir: String = output/firstJobOutput
yellowDatasetDir: String = dataset/yellow_cab
greenDatasetDir: String = dataset/green_cab
fhvDatasetDir: String = dataset/fhv_cab
fhvhvDatasetDir: String = dataset/fhvhv_cab
datasetDirMap: Map[String,String] = Map(yellow -> dataset/yellow_cab, green ->
dataset/green_cab, fhv -> dataset/fhv_cab, fhvhv -> dataset/fhvhv_cab)
datasetIterator: Map[String,(String, String)] = Map(yellow ->
(tpep_dropoff_datetime,tpep_pickup_datetime), green ->
(lpep_dropoff_datetime,lpep_pickup_datetime))
```

1.2 Define Columns for analysis

- Columns names
- Time zones for overprice
- Columns used in classification for average price calculation
- Columns which values are used in analysis

[3]:

```
val colDurationMinutes: String = "duration_minutes"
val colDurationMinutesBinLabel: String = "duration_minutes_bin_label"
val colYear: String = "year"
val colWeekdaySurcharge: String = "weekday_surcharge"
val colAggregateFee: String = "fees"
val colAggregateFeeBin: String = "agg_fee_bin_label"
val colDistanceBin: String = "distance_bin_label"
val colFareAmount: String = "fare_amount"
```

```scala
val colPricePerDistance: String = "cost_per_distance"
val colPricePerTime: String = "cost_per_time"
val colAvgPricePerDistance: String = "avg_cost_per_distance"
val colAvgPricePerTime: String = "avg_cost_per_time"
val colPricePerDistanceDiff: String = "cost_per_distance_diff"
val colPricePerDistanceDiffPcg: String = "cost_per_distance_diff_pcg"
val colPricePerTimeDiff: String = "cost_per_time_diff"
val colPricePerTimeDiffPcg: String = "cost_per_time_diff_pcg"
val colPricePerDistanceDiffPcgLabel: String = colPricePerDistanceDiffPcg + "_label"
val colPricePerTimeDiffPcgLabel: String = colPricePerTimeDiffPcg + "_label"

val timeZoneOver: String = "overnight"
val timeZones = Map(timeZoneOver -> (20, 6), "regular" -> (6, 20))
val weekDaySurcharge: Double = 2.5

val colDurationOvernightPcg: String = s"${timeZoneOver}_duration_pcg"

val colToUse: Set[String] = Set(
  "tpep_pickup_datetime",
  "tpep_dropoff_datetime",
  "lpep_pickup_datetime",
  "lpep_dropoff_datetime",
  "passenger_count",
  "trip_distance",
  "ratecodeid",
  "store_and_fwd_flag",
  "payment_type",
  "fare_amount",
  "extra",
  "mta_tax",
  "tip_amount",
  "tolls_amount",
  "improvement_surcharge",
  "total_amount",
  "congestion_surcharge",
  "airport_fee")

val colFees: Set[String] = Set(
  "extra",
  "mta_tax",
  "improvement_surcharge",
  "congestion_surcharge",
  "airport_fee")

val colsForClassification: Seq[String] = Seq(
  "passenger_count",
  "store_and_fwd_flag",
  "payment_type",
  colAggregateFeeBin,
  colDurationMinutesBinLabel,
  colDistanceBin,
  colYear,
  s"${colDurationOvernightPcg}_label",
  colPricePerDistanceDiffPcgLabel,
  colPricePerTimeDiffPcgLabel
)

val colsForValuesAnalysis: Seq[String] = Seq(
  "passenger_count",
  "store_and_fwd_flag",
  "payment_type",
  colAggregateFeeBin,
  colDurationMinutesBinLabel,
  colDistanceBin,
  colYear,
  s"${colDurationOvernightPcg}_label",
)
```

[3]:

```
colDurationMinutes: String = duration_minutes
colDurationMinutesBinLabel: String = duration_minutes_bin_label
colYear: String = year
colWeekdaySurcharge: String = weekday_surcharge
colAggregateFee: String = fees
colAggregateFeeBin: String = agg_fee_bin_label
colDistanceBin: String = distance_bin_label
colFareAmount: String = fare_amount
colPricePerDistance: String = cost_per_distance
colPricePerTime: String = cost_per_time
colAvgPricePerDistance: String = avg_cost_per_distance
colAvgPricePerTime: String = avg_cost_per_time
colPricePerDistanceDiff: String = cost_per_distance_diff
colPricePerDistanceDiffPcg: String = cost_per_distance_diff_pcg
colPricePerTimeDiff: String = cost_per_time_diff
colPricePerTimeDiffPcg: String = cost_per_time_diff_pcg
colPricePerDistanceDiffPcgLabel: String = …
```

1.2.1 Define preprocess rules

[4]:

```
import java.time.LocalDateTime

val featureFilters: Map[String, Any => Boolean] = Map(
  "passenger_count" -> {
    case i: Int => i > 0
    case f: Float => val i = f.toInt; i > 0
    case d: Double => val i = d.toInt; i > 0
    case _ => false
  },
  "trip_distance" -> {
    case i: Int => i > 0
    case i: Float => i > 0
    case i: Double => i > 0
    case _ => false
  },
  "ratecodeid" -> {
    case i: Int => (i >= 1 && i <= 6) || i == 99
    case f: Float => val i = f.toInt; (i >= 1 && i <= 6) || i == 99
    case d: Double => val i = d.toInt; (i >= 1 && i <= 6) || i == 99
    case _ => false
  },
  "store_and_fwd_flag" -> {
    case i: String => i == "Y" || i == "N"
    case _ => false
  },
  "payment_type" -> {
    case i: Int => i >= 1 && i <= 6
    case f: Float => val i = f.toInt; i >= 1 && i <= 6
    case d: Double => val i = d.toInt; i >= 1 && i <= 6
    case _ => false
  },
  "fare_amount" -> {
    case i: Int => i > 0
    case i: Float => i > 0
    case i: Double => i > 0
    case _ => false
  },
  "tolls_amount" -> {
    case i: Int => i >= 0 && i < 200
    case i: Float => i >= 0 && i < 200
    case i: Double => i >= 0 && i < 200
    case _ => false
  }
)

val taxFilter: Any => Boolean = {
  case tax: Int => tax >= 0 && tax < 20
  case tax: Float => tax >= 0 && tax < 20
  case tax: Double => tax >= 0 && tax < 20
  case _ => false
}
```

```
val dateFilter: (Any, Int) => Boolean = {
  case (date: LocalDateTime, minimumYearDataset: Int) => val year: Int = date.
↪getYear; year >= minimumYearDataset && year <= LocalDateTime.now().getYear
  case _ => false
}
```

[4]:

```
import java.time.LocalDateTime
featureFilters: Map[String,Any => Boolean] = Map(trip_distance ->
$Lambda$2388/0x0000000801090040@5cfef98e, tolls_amount ->
$Lambda$2393/0x0000000801093840@3d8f4a9a, payment_type ->
$Lambda$2391/0x0000000801092040@3a18b3fc, fare_amount ->
$Lambda$2392/0x0000000801093040@106ab276, passenger_count ->
$Lambda$2387/0x000000080108f040@3bf30b7c, store_and_fwd_flag ->
$Lambda$2390/0x0000000801091840@7c6f5d02, ratecodeid ->
$Lambda$2389/0x0000000801090840@3979620)
taxFilter: Any => Boolean = $Lambda$2394/0x0000000801094840@462b7d26
dateFilter: (Any, Int) => Boolean = $Lambda$2395/0x0000000801095040@65a02674
```

1.2.2 Utils functions for rdd

[5]:

```
import java.time.temporal.ChronoUnit
import java.time.{DayOfWeek, LocalDate}
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.Row
import scala.math.BigDecimal.RoundingMode

def getDatasetPath(localPath: String): String = {
  "file://" + projectDir + "/" + localPath
}

def binColByStepValue(rdd: RDD[Row], indexOfColToDiscrete: Int, stepValue: Int␣
↪= 5): RDD[Row] = {
  rdd
    .map { row =>
      val value: Double = row.get(indexOfColToDiscrete) match {
        case i: Int => i.toDouble
        case d: Double => d
        case l: Long => l.toDouble
        case s: String => try { s.toDouble } catch { case _: Throwable =>␣
↪Double.NaN}
        case _ => Double.NaN
      }

      val rawBin = (value / stepValue).toInt * stepValue
      val binBase = if (value < 0 && value % stepValue == 0) rawBin + stepValue␣
 ↪else rawBin
      val label = if (value < 0) { s"[${(binBase - stepValue).toInt}|${binBase.
 ↪toInt})" } else { s"[${binBase.toInt}|${(binBase + stepValue).toInt})" }

      Row.fromSeq(row.toSeq :+ label)
    }
}

val castForFilter: Any => Any = {
  case s: String => if (s.matches("""^-?\d+\.\d+$""")) s.toDouble else if (s.
 ↪matches("""^-?\d+$""")) s.toInt else s.trim
  case d: Double => d
  case i: Int => i
  case l: Long => l.toDouble
  case f: Float => f.toDouble
  case b: Boolean => b
  case null => null
  case other => other.toString.trim
}

val preciseBucketUDF: (Map[String, (Int, Int)], LocalDateTime, LocalDateTime,␣
 ↪Int) => Map[String, Double] = { (timeZones: Map[String, (Int, Int)], start:␣
 ↪LocalDateTime, end: LocalDateTime, decimals: Int) =>

  val overlap: (LocalDateTime, LocalDateTime, LocalDateTime, LocalDateTime,␣
 ↪Int) => Double = { (start1: LocalDateTime, end1: LocalDateTime, start2:␣
 ↪LocalDateTime, end2: LocalDateTime, decimals: Int) =>
    val overlapStart = if (start1.isAfter(start2)) start1 else start2
    val overlapEnd = if (end1.isBefore(end2)) end1 else end2
    if (overlapEnd.isAfter(overlapStart)) BigDecimal(ChronoUnit.MILLIS.
 ↪between(overlapStart, overlapEnd) / 60000.0).setScale(decimals, RoundingMode.
 ↪HALF_UP).toDouble else 0.0
  }

  var result = timeZones.keys.map(_ -> 0.0).toMap

  if (!(start == null || end == null)) {

    if (!end.isBefore(start)) {

      var current = start.toLocalDate.atStartOfDay

      while (!current.isAfter(end)) {
        val nextDay = current.plusDays(1)
        timeZones
          .foreach {
            case (label, (startHour, endHour)) if startHour > endHour => {
              val bucketStartBeforeMidnight = current.withHour(startHour).
 ↪withMinute(0).withSecond(0).withNano(0)
              val bucketEndBeforeMidnight = current.withHour(23).withMinute(59).
 ↪withSecond(59)
              val bucketStartAfterMidnight = current.withHour(0).withMinute(0).
 ↪withSecond(0).withNano(0)
              val bucketEndAfterMidnight = current.withHour(endHour).
 ↪withMinute(0).withSecond(0).withNano(0)

              val minutesBeforeMidnight = overlap(start, end,␣
 ↪bucketStartBeforeMidnight, bucketEndBeforeMidnight, decimals)
              val minutesAfterMidnight = overlap(start, end,␣
 ↪bucketStartAfterMidnight, bucketEndAfterMidnight, decimals)

              result = result.updated(label, result(label) +␣
 ↪minutesBeforeMidnight + minutesAfterMidnight)
            }
            case (label, (startHour, endHour)) => {
              val bucketStart = current.withHour(startHour).withMinute(0).
 ↪withSecond(0).withNano(0)
              val bucketEnd = if (endHour == 24) current.plusDays(1).
 ↪withHour(0).withMinute(0).withSecond(0).withNano(0) else current.
 ↪withHour(endHour).withMinute(0).withSecond(0).withNano(0)

              val minutes = overlap(start, end, bucketStart, bucketEnd,␣
 ↪decimals)

              result = result.updated(label, result(label) + minutes)
            }
          }

        current = nextDay
      }
    }
  }
  result
}

val isUSHolidayOrWeekend: LocalDate => Boolean = { date =>
  val month = date.getMonthValue
  val day = date.getDayOfMonth
  val dayOfWeek = date.getDayOfWeek
```

```scala
  val isIndependenceDay = month == 7 && day == 4
  val isChristmas = month == 12 && day == 25
  val isNewYear = month == 1 && day == 1
  val isLaborDay = month == 9 && dayOfWeek == DayOfWeek.MONDAY && day <= 7

  val isThanksgiving = month == 11 && dayOfWeek == DayOfWeek.THURSDAY && day >= 22 && day <= 28 && ((day - 1) / 7 + 1 == 4)

  isIndependenceDay || isChristmas || isNewYear || isLaborDay || isThanksgiving || dayOfWeek == DayOfWeek.SATURDAY || dayOfWeek == DayOfWeek.SUNDAY
}

val selectColumns: (RDD[Row], Seq[String], Set[String]) => RDD[Row] = { (rdd, headers, columnsToKeep) =>
  val keepIndexes = headers.zipWithIndex.collect {
    case (col, idx) if columnsToKeep.contains(col) => idx
  }

  rdd
    .map { row =>
      val selectedValues = keepIndexes.map(row.get)
      Row.fromSeq(selectedValues)
    }
}

val removeColumns: (RDD[Row], Seq[String], Set[String]) => RDD[Row] = { (rdd, headers, columnsToRemove) =>
  val lowerHeaders = headers.map(_.toLowerCase)
  val removeSet = columnsToRemove.map(_.toLowerCase)

  val keepIndexes = lowerHeaders.zipWithIndex.collect {
    case (col, idx) if !removeSet.contains(col) => idx
  }

  rdd
    .map { row =>
      val selectedValues = keepIndexes.map(row.get)
      Row.fromSeq(selectedValues)
    }
}
```

[5]:

```
import java.time.temporal.ChronoUnit
import java.time.{DayOfWeek, LocalDate}
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.Row
import scala.math.BigDecimal.RoundingMode
getDatasetPath: (localPath: String)String
```

```
binColByStepValue: (rdd: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row],
indexOfColToDiscrete: Int, stepValue:
Int)org.apache.spark.rdd.RDD[org.apache.spark.sql.Row]
castForFilter: Any => Any = $Lambda$2545/0x00000080110d040@3566623f
preciseBucketUDF: (Map[String,(Int, Int)], java.time.LocalDateTime,
java.time.LocalDateTime, Int) => Map[String,Double] =
$Lambda$2546/0x00000080110d840@76c0348c
isUSHolidayOrWeekend: java.time.LocalDate => Boolean =
$Lambda$2547/0x00000080110e040@5a1dbf9e
selectColumns: (org.apache.spark.rdd.RDD[org.apache.spark.sql.Row], Seq[St…
```

2 Actual job

1) Select dataset [yellow or green]

[6]:

```scala
val name: String = "green"
val (dropoff, pickup) = datasetIterator(name)
```

[6]:

```
name: String = green
dropoff: String = lpep_dropoff_datetime
pickup: String = lpep_pickup_datetime
```

2) Load dataset

[7]:

```scala
val startTime = System.currentTimeMillis()

val dataset = spark.read.parquet(getDatasetPath(datasetDirMap(name)))
var headers: Seq[String] = dataset.columns.map(_.toLowerCase)
val indexesToUse: Seq[Int] = headers.zipWithIndex.collect {
  case (h, i) if colToUse.contains(h.toLowerCase) => i
}
headers = headers
  .filter(head => colToUse.contains(head.toLowerCase))
```

[7]:

```
startTime: Long = 1751894127218
dataset: org.apache.spark.sql.DataFrame = [VendorID: int, lpep_pickup_datetime:
timestamp_ntz … 18 more fields]
headers: Seq[String] = ArraySeq(lpep_pickup_datetime, lpep_dropoff_datetime,
store_and_fwd_flag, ratecodeid, passenger_count, trip_distance, fare_amount,
extra, mta_tax, tip_amount, tolls_amount, improvement_surcharge, total_amount,
payment_type, congestion_surcharge)
indexesToUse: Seq[Int] = ArraySeq(1, 2, 3, 4, 7, 8, 9, 10, 11, 12, 13, 15, 16,
17, 19)
headers: Seq[String] = ArraySeq(lpep_pickup_datetime, lpep_dropoff_datetime,
store_and_fwd_flag, ratecodeid, passenger_count, trip_distance, fare_amount,
extra, mta_tax, tip_amount, tolls_amount, improvement_surcharge, total_amount,
payment_type, congestion_surcharge)
```

3) Filter taxes and features based on filter conditions previously defined

[8]:

```
import org.apache.spark.sql.DataFrame
import java.time.format.DateTimeFormatter

def transformRDD(dataset: DataFrame, idxs: Seq[Int], castFunc: Any => Any):␣
 ↪RDD[Row] = {
  dataset.rdd
    .map(row => Row.fromSeq(idxs.map(row.get).map(castFunc)))
}

val rdd = transformRDD(dataset, indexesToUse, castForFilter)

def applyFilters(rdd: RDD[Row], headers: Seq[String], colOfFees: Set[String],␣
 ↪taxFilter: Any => Boolean, featFilter: Map[String, Any => Boolean],␣
 ↪dateFilter: (Any, Int) => Boolean, dropoff: String, pickup: String,␣
 ↪minimumYearDataset: Int): RDD[Row] = {
  rdd
    .filter { row =>
      val formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd'T'HH:mm[:ss]")
      headers.zip(row.toSeq)
        .forall {
          case (header: String, value) => {
            val taxFilterCondition = if (colOfFees.contains(header.
 ↪toLowerCase)) taxFilter(value) else true

            featFilter.get(header.toLowerCase) match {
              case Some(filterFunc) => taxFilterCondition && filterFunc(value)
              case None => if (header.equals(pickup) || header.equals(dropoff))␣
 ↪{
                dateFilter(LocalDateTime.parse(row.getAs[String](headers.
 ↪indexOf(header)).trim, formatter), minimumYearDataset) && taxFilterCondition
              } else taxFilterCondition
            }
          }
        }
    }
}

val rddFiltered = applyFilters(rdd, headers, colFees, taxFilter,␣
 ↪featureFilters, dateFilter, dropoff, pickup, minimumYearDataset)
```

[8]:

```
import org.apache.spark.sql.DataFrame
import java.time.format.DateTimeFormatter
transformRDD: (dataset: org.apache.spark.sql.DataFrame, idxs: Seq[Int],
castFunc: Any => Any)org.apache.spark.rdd.RDD[org.apache.spark.sql.Row]
rdd: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] = MapPartitionsRDD[8] at
map at <console>:46
applyFilters: (rdd: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row], headers:
Seq[String], colOfFees: Set[String], taxFilter: Any => Boolean, featFilter:
Map[String,Any => Boolean], dateFilter: (Any, Int) => Boolean, dropoff: String,
pickup: String, minimumYearDataset:
Int)org.apache.spark.rdd.RDD[org.apache.spark.sql.Row]
rddFiltered: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] =
MapPartitionsRDD[9] at filter at <console>:53
```

4) Add duration and timezones

[9]:

```
import java.time.Duration

def addDuration(rdd: RDD[Row], headers: Seq[String], pickup: String, dropoff:␣
 ↪String, decimals: Int): RDD[Row] = {
  rdd
    .map { row =>
      val formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd'T'HH:mm[:ss]")

      val pickupStr = row.getAs[String](headers.indexOf(pickup)).trim
      val dropoffStr = row.getAs[String](headers.indexOf(dropoff)).trim

      val pickupTS = LocalDateTime.parse(pickupStr, formatter)
      val dropoffTS = LocalDateTime.parse(dropoffStr, formatter)
      val durationMillis = Duration.between(pickupTS, dropoffTS).toMillis
      val durationMinutes = BigDecimal(durationMillis / 60000.0).
 ↪setScale(decimals, RoundingMode.HALF_UP).toDouble

      val pickupYear = pickupTS.getYear

      Row.fromSeq(row.toSeq ++ Seq(durationMinutes, pickupYear))
    }
    .filter {
      row => row.getAs[Double](row.toSeq.length - 2) > 0.0
    }
}

val rddDuration = addDuration(rddFiltered, headers, pickup, dropoff, decimals)
headers = headers ++ Seq(colDurationMinutes, colYear)
```

```
val rddDurationBin = binColByStepValue(rddDuration, headers.indexOf(colDurationMinutes), 5)
headers = headers :+ colDurationMinutesBinLabel

def addTimeZones(rdd: RDD[Row], headers: Seq[String], timezones: Map[String, (Int, Int)], weekDaySurcharge: Double, colDuration: String, pickup: String, dropoff: String, decimals: Int, preciseBucketUDF: (Map[String, (Int, Int)], LocalDateTime, LocalDateTime, Int) => Map[String, Double], isUSHolidayOrWeekendTZ: LocalDate => Boolean): RDD[Row] = {
  rdd
    .map { row =>
      val formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd'T'HH:mm[:ss]")

      val timeZonesDuration: Map[String, Double] = preciseBucketUDF(timezones, LocalDateTime.parse(row.getAs[String](headers.indexOf(pickup)).trim, formatter), LocalDateTime.parse(row.getAs[String](headers.indexOf(dropoff)).trim, formatter), decimals)

      val weekday_surcharge: Double = if (isUSHolidayOrWeekendTZ(LocalDateTime.parse(row.getAs[String](headers.indexOf(pickup)).trim, formatter).toLocalDate)) 0 else weekDaySurcharge
      val colsToAdd: Seq[Double] = timezones.keys.toSeq
        .flatMap { tz =>
          val duration = timeZonesDuration.getOrElse(tz, 0.0)
          val totalDuration = row.getAs[Double](headers.indexOf(colDuration))
          Seq(duration, BigDecimal(duration * 100 / totalDuration).setScale(decimals, RoundingMode.HALF_UP).toDouble)
        }
      Row.fromSeq((row.toSeq ++ colsToAdd) :+ weekday_surcharge)
    }
}

val rddTimeZones = addTimeZones(rddDurationBin, headers, timeZones, weekDaySurcharge, colDurationMinutes, pickup, dropoff, decimals, preciseBucketUDF, isUSHolidayOrWeekend)

val headersToAdd: Seq[String] = timeZones.keys.toSeq.flatMap { tz =>
  Seq(tz + "_duration", tz + "_duration_pcg")
} :+ colWeekdaySurcharge

headers = headers ++ headersToAdd
```

[9]:
```
import java.time.Duration
addDuration: (rdd: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row], headers:
Seq[String], pickup: String, dropoff: String, decimals:
Int)org.apache.spark.rdd.RDD[org.apache.spark.sql.Row]
rddDuration: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] =
MapPartitionsRDD[11] at filter at <console>:67
headers: Seq[String] = ArraySeq(lpep_pickup_datetime, lpep_dropoff_datetime,
store_and_fwd_flag, ratecodeid, passenger_count, trip_distance, fare_amount,
extra, mta_tax, tip_amount, tolls_amount, improvement_surcharge, total_amount,
payment_type, congestion_surcharge, duration_minutes, year,
duration_minutes_bin_label, overnight_duration, overnight_duration_pcg,
regular_duration, regular_duration_pcg, weekday_surcharge)
rddDurationBin: org.apache.spark.rdd.RDD[or…
```

[10]:

```
val colToRemoveTimeZones = Set(pickup, dropoff, "overnight_duration",␣
 ↪"regular_duration", "regular_duration_pcg", "ratecodeid", "tip_amount",␣
 ↪"tolls_amount", "total_amount")

val rddTimeZonesOpt = removeColumns(rddTimeZones, headers, colToRemoveTimeZones)
var headersTimeZonesOpt = headers.filterNot(col => colToRemoveTimeZones.
 ↪contains(col.toLowerCase))
```

[10]:

```
colToRemoveTimeZones: scala.collection.immutable.Set[String] =
Set(lpep_dropoff_datetime, regular_duration, overnight_duration,
regular_duration_pcg, tolls_amount, tip_amount, lpep_pickup_datetime,
total_amount, ratecodeid)
rddTimeZonesOpt: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] =
MapPartitionsRDD[14] at map at <console>:153
headersTimeZonesOpt: Seq[String] = ArraySeq(store_and_fwd_flag, passenger_count,
trip_distance, fare_amount, extra, mta_tax, improvement_surcharge, payment_type,
congestion_surcharge, duration_minutes, year, duration_minutes_bin_label,
overnight_duration_pcg, weekday_surcharge)
```

5) Add Aggregate fees and bins

[11]:

```
def addAggregateFees(rdd: RDD[Row], headers: Seq[String], colOfFees:␣
 ↪Set[String]): RDD[Row] = {
  rdd
    .map { row =>
      val fees = colOfFees
        .filter(col => headers.contains(col.toLowerCase))
        .map(col => row.getAs[Double](headers.indexOf(col.toLowerCase))).sum

      Row.fromSeq(row.toSeq :+ fees)
    }
}

val rddAggFees = addAggregateFees(rddTimeZonesOpt, headersTimeZonesOpt, colFees)
headersTimeZonesOpt = headersTimeZonesOpt :+ colAggregateFee
```

```
val rddAggFeesBin = binColByStepValue(rddAggFees, headersTimeZonesOpt.
 ↪indexOf(colAggregateFee), 2)
headersTimeZonesOpt = headersTimeZonesOpt :+ colAggregateFeeBin
```

```
[11]: addAggregateFees: (rdd: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row],
headers: Seq[String], colOfFees:
Set[String])org.apache.spark.rdd.RDD[org.apache.spark.sql.Row]
rddAggFees: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] =
MapPartitionsRDD[15] at map at <console>:43
headersTimeZonesOpt: Seq[String] = ArraySeq(store_and_fwd_flag, passenger_count,
trip_distance, fare_amount, extra, mta_tax, improvement_surcharge, payment_type,
congestion_surcharge, duration_minutes, year, duration_minutes_bin_label,
overnight_duration_pcg, weekday_surcharge, fees, agg_fee_bin_label)
rddAggFeesBin: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] =
MapPartitionsRDD[16] at map at <console>:40
headersTimeZonesOpt: Seq[String] = ArraySeq(store_and_fwd_flag, passenger_count,
trip_distance, fare…
```

```
[12]: val colToRemoveAggFees = colFees ++ Set(colAggregateFee)

val rddAggFeesOpt = removeColumns(rddAggFeesBin, headersTimeZonesOpt,␣
 ↪colToRemoveAggFees)
var headersAggFeesOpt = headersTimeZonesOpt.filterNot(col => colToRemoveAggFees.
 ↪contains(col.toLowerCase))
```

```
[12]: colToRemoveAggFees: scala.collection.immutable.Set[String] =
Set(improvement_surcharge, fees, extra, airport_fee, congestion_surcharge,
mta_tax)
rddAggFeesOpt: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] =
MapPartitionsRDD[17] at map at <console>:153
headersAggFeesOpt: Seq[String] = ArraySeq(store_and_fwd_flag, passenger_count,
trip_distance, fare_amount, payment_type, duration_minutes, year,
duration_minutes_bin_label, overnight_duration_pcg, weekday_surcharge,
agg_fee_bin_label)
```

6) Add price per mile and minute

```
[13]: def addPricePerDistanceAndTime(rdd: RDD[Row], headers: Seq[String],␣
 ↪colFareAmount: String, colDuration: String, colDistance: String): RDD[Row] =␣
 ↪{
  rdd
    .map { row =>
      val pricePerTime = Math.round(row.getAs[Double](headers.
 ↪indexOf(colFareAmount)) / row.getAs[Double](headers.indexOf(colDuration)) *␣
 ↪100) / 100.0
```

```
      val pricePerDistance = Math.round(row.getAs[Double](headers.
  ↪indexOf(colFareAmount)) / row.getAs[Double](headers.indexOf(colDistance)) *␣
  ↪100) / 100.0

      Row.fromSeq(row.toSeq ++ Seq(pricePerTime, pricePerDistance))
    }
}

val rddPriced = addPricePerDistanceAndTime(rddAggFeesOpt, headersAggFeesOpt,␣
  ↪colFareAmount, colDurationMinutes, "trip_distance")
headersAggFeesOpt = headersAggFeesOpt ++ Seq(colPricePerTime,␣
  ↪colPricePerDistance)
```

[13]:
```
addPricePerDistanceAndTime: (rdd:
org.apache.spark.rdd.RDD[org.apache.spark.sql.Row], headers: Seq[String],
colFareAmount: String, colDuration: String, colDistance:
String)org.apache.spark.rdd.RDD[org.apache.spark.sql.Row]
rddPriced: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] =
MapPartitionsRDD[18] at map at <console>:43
headersAggFeesOpt: Seq[String] = ArraySeq(store_and_fwd_flag, passenger_count,
trip_distance, fare_amount, payment_type, duration_minutes, year,
duration_minutes_bin_label, overnight_duration_pcg, weekday_surcharge,
agg_fee_bin_label, cost_per_time, cost_per_distance)
```

[14]:
```
val colToRemovePricePerDistanceAndTime = Set(colFareAmount, colDurationMinutes)

val rddPricePerDistanceAndTimeOpt = removeColumns(rddPriced, headersAggFeesOpt,␣
  ↪colToRemovePricePerDistanceAndTime)
var headersPricePerDistanceAndTimeOpt = headersAggFeesOpt.filterNot(col =>␣
  ↪colToRemovePricePerDistanceAndTime.contains(col.toLowerCase))
```

[14]:
```
colToRemovePricePerDistanceAndTime: scala.collection.immutable.Set[String] =
Set(fare_amount, duration_minutes)
rddPricePerDistanceAndTimeOpt:
org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] = MapPartitionsRDD[19] at map
at <console>:153
headersPricePerDistanceAndTimeOpt: Seq[String] = ArraySeq(store_and_fwd_flag,
passenger_count, trip_distance, payment_type, year, duration_minutes_bin_label,
overnight_duration_pcg, weekday_surcharge, agg_fee_bin_label, cost_per_time,
cost_per_distance)
```

7) Add distance bin and duration in overnight time zone

[15]:
```
val rddDistBin = binColByStepValue(rddPricePerDistanceAndTimeOpt,␣
  ↪headersPricePerDistanceAndTimeOpt.indexOf("trip_distance"), 5)
```

```
headersPricePerDistanceAndTimeOpt = headersPricePerDistanceAndTimeOpt :+␣
 ↪colDistanceBin

val rddOvernightBin = binColByStepValue(rddDistBin,␣
 ↪headersPricePerDistanceAndTimeOpt.indexOf(colDurationOvernightPcg), 5)
headersPricePerDistanceAndTimeOpt = headersPricePerDistanceAndTimeOpt :+␣
 ↪(colDurationOvernightPcg + "_label")
```

[15]:

```
rddDistBin: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] =
MapPartitionsRDD[20] at map at <console>:40
headersPricePerDistanceAndTimeOpt: Seq[String] = ArraySeq(store_and_fwd_flag,
passenger_count, trip_distance, payment_type, year, duration_minutes_bin_label,
overnight_duration_pcg, weekday_surcharge, agg_fee_bin_label, cost_per_time,
cost_per_distance, distance_bin_label, overnight_duration_pcg_label)
rddOvernightBin: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] =
MapPartitionsRDD[21] at map at <console>:40
headersPricePerDistanceAndTimeOpt: Seq[String] = ArraySeq(store_and_fwd_flag,
passenger_count, trip_distance, payment_type, year, duration_minutes_bin_label,
overnight_duration_pcg, weekday_surcharge, agg_fee_bin_label, cost_per_time,
cost_per_distance, distance_bin_lab…
```

[16]:

```
val colToRemoveOvernightBin = Set("trip_distance", colDurationOvernightPcg,␣
 ↪colWeekdaySurcharge)

val rddOvernightBinOpt = removeColumns(rddOvernightBin,␣
 ↪headersPricePerDistanceAndTimeOpt, colToRemoveOvernightBin)
var headersOvernightBinOpt = headersPricePerDistanceAndTimeOpt.filterNot(col =>␣
 ↪colToRemoveOvernightBin.contains(col.toLowerCase))
```

[16]:

```
colToRemoveOvernightBin: scala.collection.immutable.Set[String] =
Set(trip_distance, overnight_duration_pcg, weekday_surcharge)
rddOvernightBinOpt: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] =
MapPartitionsRDD[22] at map at <console>:153
headersOvernightBinOpt: Seq[String] = ArraySeq(store_and_fwd_flag,
passenger_count, payment_type, year, duration_minutes_bin_label,
agg_fee_bin_label, cost_per_time, cost_per_distance, distance_bin_label,
overnight_duration_pcg_label)
```

8) Add key for average calculation based on columns for classification

[17]:

```
import org.apache.spark.storage.StorageLevel

val actualHeader = headersOvernightBinOpt
def addKey(rdd: RDD[Row], colsClassification: Seq[String], headers:␣
 ↪Seq[String]): RDD[(String, Row)] = {
  rdd
    .map { row =>
      val key = colsClassification.filter(col => headers.contains(col.
↪toLowerCase))
      .map(col => row.get(headers.indexOf(col.toLowerCase)))
      .mkString("_")
      (key, row)
    }
}

val rddWithKey = addKey(rddOvernightBinOpt, colsForClassification,␣
↪actualHeader).persist(StorageLevel.MEMORY_AND_DISK_SER)
```

```
(1.0_N_1.0_[0|2)_[5|10)_[0|5)_2024.0_[100|105),[N,1.0,1.0,2024.0,[5|10),[0|2),1.
17,7.5,[0|5),[100|105)])
```

[17]:

```
import org.apache.spark.storage.StorageLevel
actualHeader: Seq[String] = ArraySeq(store_and_fwd_flag, passenger_count,
payment_type, year, duration_minutes_bin_label, agg_fee_bin_label,
cost_per_time, cost_per_distance, distance_bin_label,
overnight_duration_pcg_label)
addKey: (rdd: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row],
colsClassification: Seq[String], headers:
Seq[String])org.apache.spark.rdd.RDD[(String, org.apache.spark.sql.Row)]
rddWithKey: org.apache.spark.rdd.RDD[(String, org.apache.spark.sql.Row)] =
MapPartitionsRDD[23] at map at <console>:43
```

9) Calculate prices per distance and time

[18]:

```
def calculatePrices(rdd: RDD[(String, Row)], headers: Seq[String],␣
↪colPriceDistance: String, colPriceTime: String): RDD[(String, (Double,␣
↪Double, Long))] = {
  rdd
    .mapValues { row =>
      val costPerDistance = row.getAs[Double](headers.indexOf(colPriceDistance))
      val costPerTime = row.getAs[Double](headers.indexOf(colPriceTime))
      (costPerDistance, costPerTime, 1L)
    }
}

val rddForAvg = calculatePrices(rddWithKey, headersOvernightBinOpt,␣
↪colPricePerDistance, colPricePerTime)
```

[18]:

```
calculatePrices: (rdd: org.apache.spark.rdd.RDD[(String,
org.apache.spark.sql.Row)], headers: Seq[String], colPriceDistance: String,
colPriceTime: String)org.apache.spark.rdd.RDD[(String, (Double, Double, Long))]
rddForAvg: org.apache.spark.rdd.RDD[(String, (Double, Double, Long))] =
MapPartitionsRDD[24] at mapValues at <console>:42
```

10) Calculate average prices per distance and time

[19]:

```
def calculateAvgPrices(rdd: RDD[(String, (Double, Double, Long))], decimals:␣
 ↪Int): RDD[(String, (Double, Double))] = {
  rdd
    .reduceByKey {
      case ((d1, t1, c1), (d2, t2, c2)) => (d1 + d2, t1 + t2, c1 + c2)
    }
    .mapValues {
      case (sumDist, sumTime, count) =>
        val avgDist = BigDecimal(sumDist / count).setScale(decimals, BigDecimal.
 ↪RoundingMode.HALF_UP).toDouble
        val avgTime = BigDecimal(sumTime / count).setScale(decimals, BigDecimal.
 ↪RoundingMode.HALF_UP).toDouble
        (avgDist, avgTime)
    }
    .filter {
      case (_, (dist, time)) => dist > 0.0 && time > 0.0
    }
}

val rddWithAvgPrices = calculateAvgPrices(rddForAvg, decimals)
```

[19]:

```
calculateAvgPrices: (rdd: org.apache.spark.rdd.RDD[(String, (Double, Double,
Long))], decimals: Int)org.apache.spark.rdd.RDD[(String, (Double, Double))]
rddWithAvgPrices: org.apache.spark.rdd.RDD[(String, (Double, Double))] =
MapPartitionsRDD[27] at filter at <console>:48
```

11) Join average prices to previous rdd

[20]:

```
def applyJoin(rdd: RDD[(String, Row)], rddToJoin: RDD[(String, (Double,␣
 ↪Double))]): RDD[Row] = {
  rdd
    .join(rddToJoin)
    .map { case (_, (originalRow, (avgCostPerDistance, avgCostPerTime))) =>
      Row.fromSeq(originalRow.toSeq ++ Seq(avgCostPerDistance, avgCostPerTime))
    }
}

val rddJoined = applyJoin(rddWithKey, rddWithAvgPrices)

rddWithKey.unpersist()
```

```
headersOvernightBinOpt = headersOvernightBinOpt ++ Seq(colAvgPricePerDistance,␣
↪colAvgPricePerTime)
```

[20]:

```
applyJoin: (rdd: org.apache.spark.rdd.RDD[(String, org.apache.spark.sql.Row)],
rddToJoin: org.apache.spark.rdd.RDD[(String, (Double,
Double))])org.apache.spark.rdd.RDD[org.apache.spark.sql.Row]
rddJoined: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] =
MapPartitionsRDD[31] at map at <console>:43
headersOvernightBinOpt: Seq[String] = ArraySeq(store_and_fwd_flag,
passenger_count, payment_type, year, duration_minutes_bin_label,
agg_fee_bin_label, cost_per_time, cost_per_distance, distance_bin_label,
overnight_duration_pcg_label, avg_cost_per_distance, avg_cost_per_time)
```

[21]:

```
val colToRemoveJoin = Set("trip_distance", colDurationOvernightPcg)

val rddJoinOpt = removeColumns(rddJoined, headersOvernightBinOpt,␣
↪colToRemovePricePerDistanceAndTime)
var headersJoinOpt = headersOvernightBinOpt.filterNot(col => colToRemoveJoin.
↪contains(col.toLowerCase))
```

[21]:

```
colToRemoveJoin: scala.collection.immutable.Set[String] = Set(trip_distance,
overnight_duration_pcg)
rddJoinOpt: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] =
MapPartitionsRDD[32] at map at <console>:153
headersJoinOpt: Seq[String] = ArraySeq(store_and_fwd_flag, passenger_count,
payment_type, year, duration_minutes_bin_label, agg_fee_bin_label,
cost_per_time, cost_per_distance, distance_bin_label,
overnight_duration_pcg_label, avg_cost_per_distance, avg_cost_per_time)
```

12) Add price comparison w.r.t. average price and actual price difference

[22]:

```
def addPriceComparison(rdd: RDD[Row], headers: Seq[String], colPriceDistance:␣
↪String, colAvgPriceDistance: String, colPriceTime: String, colAvgPriceTime:␣
↪String, decimals: Int) = {
  rdd.map { row =>
    val priceColsToAdd: Seq[Double] = Seq((colPriceDistance,␣
↪colAvgPriceDistance), (colPriceTime, colAvgPriceTime))
      .flatMap { case (colPrice, colAvgPrice) =>
        val price = row.getAs[Double](headers.indexOf(colPrice))
        val priceAvg = row.getAs[Double](headers.indexOf(colAvgPrice))
        val priceDiff = BigDecimal(price - priceAvg).setScale(decimals,␣
↪BigDecimal.RoundingMode.HALF_UP).toDouble
        val priceDiffPcg = BigDecimal(priceDiff / priceAvg * 100).
↪setScale(decimals, BigDecimal.RoundingMode.HALF_UP).toDouble

```

```
        Seq(priceDiff, priceDiffPcg)
      }
    Row.fromSeq(row.toSeq ++ priceColsToAdd)
  }
}

val rddPriceComparison = addPriceComparison(rddJoinOpt, headersJoinOpt,␣
 ↪colPricePerDistance, colAvgPricePerDistance, colPricePerTime,␣
 ↪colAvgPricePerTime, decimals)
headersJoinOpt = headersJoinOpt ++ Seq(colPricePerDistanceDiff,␣
 ↪colPricePerDistanceDiffPcg, colPricePerTimeDiff, colPricePerTimeDiffPcg)
```

[22]:
```
addPriceComparison: (rdd: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row],
headers: Seq[String], colPriceDistance: String, colAvgPriceDistance: String,
colPriceTime: String, colAvgPriceTime: String, decimals:
Int)org.apache.spark.rdd.RDD[org.apache.spark.sql.Row]
rddPriceComparison: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] =
MapPartitionsRDD[33] at map at <console>:48
headersJoinOpt: Seq[String] = ArraySeq(store_and_fwd_flag, passenger_count,
payment_type, year, duration_minutes_bin_label, agg_fee_bin_label,
cost_per_time, cost_per_distance, distance_bin_label,
overnight_duration_pcg_label, avg_cost_per_distance, avg_cost_per_time,
cost_per_distance_diff, cost_per_distance_diff_pcg, cost_per_time_diff,
cost_per_time_diff_pcg)
```

13) Bin price difference per time and distance

[23]:
```
val rddPriceDistBin = binColByStepValue(rddPriceComparison,headersJoinOpt.
 ↪indexOf(colPricePerDistanceDiffPcg), 5)
val rddPriceDistTimeBin = binColByStepValue(rddPriceDistBin, headersJoinOpt.
 ↪indexOf(colPricePerTimeDiffPcg), 5)

headersJoinOpt = headersJoinOpt ++ Seq(colPricePerDistanceDiffPcgLabel,␣
 ↪colPricePerTimeDiffPcgLabel)
```

[23]:
```
rddPriceDistBin: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] =
MapPartitionsRDD[34] at map at <console>:40
rddPriceDistTimeBin: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] =
MapPartitionsRDD[35] at map at <console>:40
headersJoinOpt: Seq[String] = ArraySeq(store_and_fwd_flag, passenger_count,
payment_type, year, duration_minutes_bin_label, agg_fee_bin_label,
cost_per_time, cost_per_distance, distance_bin_label,
overnight_duration_pcg_label, avg_cost_per_distance, avg_cost_per_time,
cost_per_distance_diff, cost_per_distance_diff_pcg, cost_per_time_diff,
cost_per_time_diff_pcg, cost_per_distance_diff_pcg_label,
cost_per_time_diff_pcg_label)
```

[24]:
```
val colToRemovePriceDistTimeBin = Set(colPricePerDistanceDiff,␣
 ↪colPricePerDistanceDiffPcg, colPricePerTimeDiff, colPricePerTimeDiffPcg,␣
 ↪colPricePerTime, colPricePerDistance, colAvgPricePerDistance,␣
 ↪colAvgPricePerTime)

val rddPriceDistTimeBinOpt = removeColumns(rddPriceDistTimeBin, headersJoinOpt,␣
 ↪colToRemovePriceDistTimeBin)
var headersPriceDistTimeBinOpt = headersJoinOpt.filterNot(col =>␣
 ↪colToRemovePriceDistTimeBin.contains(col.toLowerCase))
```

[24]:
```
colToRemovePriceDistTimeBin: scala.collection.immutable.Set[String] =
Set(avg_cost_per_distance, cost_per_distance, cost_per_distance_diff_pcg,
avg_cost_per_time, cost_per_time_diff, cost_per_time_diff_pcg,
cost_per_distance_diff, cost_per_time)
rddPriceDistTimeBinOpt: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] =
MapPartitionsRDD[36] at map at <console>:153
headersPriceDistTimeBinOpt: Seq[String] = ArraySeq(store_and_fwd_flag,
passenger_count, payment_type, year, duration_minutes_bin_label,
agg_fee_bin_label, distance_bin_label, overnight_duration_pcg_label,
cost_per_distance_diff_pcg_label, cost_per_time_diff_pcg_label)
```

14) Reduce to analysis columns only

[25]:
```
val headersForAnalysis = headersPriceDistTimeBinOpt.zipWithIndex.filter(head =>␣
 ↪colsForClassification.contains(head._1.toLowerCase))

val headersForAnalysisIdxs = headersForAnalysis.map(_._2)
val headersForAnalysisCols = headersForAnalysis.map(_._1)

def reduceToAnalysis(rdd: RDD[Row], idxs: Seq[Int]): RDD[Row] = {
  rdd.map { row => Row.fromSeq(idxs.map(row.get)) }
}

val rddAnalysis = reduceToAnalysis(rddPriceDistTimeBinOpt,␣
 ↪headersForAnalysisIdxs)

val totalCount = rddAnalysis.count()
```

[25]:
```
headersForAnalysis: Seq[(String, Int)] = ArraySeq((store_and_fwd_flag,0),
(passenger_count,1), (payment_type,2), (year,3), (duration_minutes_bin_label,4),
(agg_fee_bin_label,5), (distance_bin_label,6), (overnight_duration_pcg_label,7),
(cost_per_distance_diff_pcg_label,8), (cost_per_time_diff_pcg_label,9))
headersForAnalysisIdxs: Seq[Int] = ArraySeq(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
headersForAnalysisCols: Seq[String] = ArraySeq(store_and_fwd_flag,
passenger_count, payment_type, year, duration_minutes_bin_label,
agg_fee_bin_label, distance_bin_label, overnight_duration_pcg_label,
cost_per_distance_diff_pcg_label, cost_per_time_diff_pcg_label)
reduceToAnalysis: (rdd: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row],
idxs: Seq[Int])org.apache.spark.rdd.RDD[org.apache.spark.sql.Row]
rddAnaly…
```

15) Group by feature value

[26]:
```
def groupByFeatures(rdd: RDD[Row], colForValuesAnalysis: Seq[String],␣
 ↪colPriceDistanceDiffPcgLabel: String, colPriceTimeDiffPcgLabel: String,␣
 ↪headersAnalysis: Seq[String], decimals: Int, totalCount: Long):␣
 ↪Seq[RDD[Row]] = {
  colForValuesAnalysis
    .map { colName =>
      val groupCols = Seq(colPriceDistanceDiffPcgLabel,␣
 ↪colPriceTimeDiffPcgLabel):+ colName
      val grouped = rdd
        .map { row =>
          val key = groupCols.map(col => row.get(headersAnalysis.indexOf(col.
 ↪toLowerCase)))
          (key, 1)
        }
        .reduceByKey(_ + _)
        .map {
          case (keySeq, count) => {
            val value = keySeq.last.toString
            val costDistLabel = keySeq(0).toString
            val costTimeLabel = keySeq(1).toString
            val pcg = BigDecimal(count.toDouble / totalCount * 100).
 ↪setScale(decimals, BigDecimal.RoundingMode.HALF_UP).toDouble
            Row.fromSeq(Seq(colName, value, count, pcg, costDistLabel,␣
 ↪costTimeLabel))
          }
        }
      grouped
    }
}

val rddFeatures = groupByFeatures(rddAnalysis, colsForValuesAnalysis,␣
 ↪colPricePerDistanceDiffPcgLabel, colPricePerTimeDiffPcgLabel,␣
 ↪headersForAnalysisCols, decimals, totalCount)
```

[26]:
```
groupByFeatures: (rdd: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row],
colForValuesAnalysis: Seq[String], colPriceDistanceDiffPcgLabel: String,
colPriceTimeDiffPcgLabel: String, headersAnalysis: Seq[String], decimals: Int,
totalCount: Long)Seq[org.apache.spark.rdd.RDD[org.apache.spark.sql.Row]]
rddFeatures: Seq[org.apache.spark.rdd.RDD[org.apache.spark.sql.Row]] =
List(MapPartitionsRDD[40] at map at <console>:52, MapPartitionsRDD[43] at map at
<console>:52, MapPartitionsRDD[46] at map at <console>:52, MapPartitionsRDD[49]
at map at <console>:52, MapPartitionsRDD[52] at map at <console>:52,
MapPartitionsRDD[55] at map at <console>:52, MapPartitionsRDD[58] at map at
<console>:52, MapPartitionsRDD[61] at map at <console>:52)
```

16) Reduce to single rdd and write output

[27]:

```scala
import org.apache.spark.sql.types._

val headersForSchema = Seq(
  StructField("feature", StringType),
  StructField("value", StringType),
  StructField("count", IntegerType),
  StructField("pcg", DoubleType),
  StructField("cost_distance_label", StringType),
  StructField("cost_time_label", StringType)
)

val schema = StructType(headersForSchema)

val dfForAnalysis = spark.createDataFrame(rddFeatures.reduce(_ union _).
  ↪coalesce(1), schema)

dfForAnalysis.show(1)
val endTime = System.currentTimeMillis()
val durationMs = endTime - startTime

println(s"Job $name-dataset non optimized executed in $durationMs ms")

dfForAnalysis.write.mode("overwrite").parquet(getDatasetPath(outputDir + f"/
  ↪$name"))
```

```
+---------------+-----+-----+------+-------------------+---------------+
|        feature|value|count|   pcg|cost_distance_label|cost_time_label|
+---------------+-----+-----+------+-------------------+---------------+
|passenger_count|  6.0|    1|1.0E-4|            [35|40)|      [-95|-90)|
+---------------+-----+-----+------+-------------------+---------------+
only showing top 1 row

Job green-dataset non optimized executed in 71139 ms
```

[27]:

```
import org.apache.spark.sql.types._
headersForSchema: Seq[org.apache.spark.sql.types.StructField] =
List(StructField(feature,StringType,true), StructField(value,StringType,true),
StructField(count,IntegerType,true), StructField(pcg,DoubleType,true),
StructField(cost_distance_label,StringType,true),
StructField(cost_time_label,StringType,true))
schema: org.apache.spark.sql.types.StructType = StructType(StructField(feature,S
tringType,true),StructField(value,StringType,true),StructField(count,IntegerType
,true),StructField(pcg,DoubleType,true),StructField(cost_distance_label,StringTy
pe,true),StructField(cost_time_label,StringType,true))
dfForAnalysis: org.apache.spark.sql.DataFrame = [feature: string, value: string
… 4 more fields]
endTime: Long = 1751894198357
durationMs: Long = 71139
```