Spark assignment

- Author: David Kozak
- Email: dkozak94@gmail.com

The goal of this assignment was to implement 12 data analysis tasks using Apache Spark. The structure of the report is as follows. First, the dataset is introduced. Afterwards descriptions and solutions for each of the tasks are given.

Dataset

As in the first mongo assignment, I decided to use the dataset employee reviews, which contains reviews from employees of 6 IT companies - Google, Facebook, Amazon, Apple and Netflix. The reviews are stored in a csv file. The structure of one review can be determined by checking the header line.

```
head -n 1 employee_reviews.csv
```

Which results in:

```
id,company,location,dates,job-title,summary,pros,cons,advice-to-mgmt,overall-ratings,
work-balance-stars,culture-values-stars,carrer-opportunities-stars,comp-benefit-stars
,senior-mangemnet-stars,helpful-count,link
```

The size is 67529 reviews.

```
tail -n 1
```

```
67529,microsoft,none," Dec 14, 2010",Former Employee - Sen ....
```

Run the code

The whole repository is a gradle module and the source code is written in Kotlin. It can be compiled into jar using the following command.

```
gralde jar
```

This jar can then be submitted to locally installed apache spark instance.

```
${SPARK_EXEC} --class io.dkozak.estg.spark.assignment.MainKt \
--master "local[4]" ${JAR_FILE} ${INPUT_FILE} ${OUTPUT_DIR}
```

- SPARK_EXEC is the location of the spark-submit file in the bin folder inside the spark directory
- JAR_FILE is the jar build the previous step
- INPUT_FILE is a path to the input csv file
- OUTPUT_DIR is a directory into which the output should be written

To allow for easier execution, execute.sh shell script is prepared. This script takes following arguments.

- location where ApacheSpark is installed
- [optional] a single task that should be run (default is all tasks)

Please note that for this script to work correctly, the input csv file has to be stored at path ./employee_reviews.csv.

The output can be found in the ./output directory, each task results are stored in a subfolder identified by the number of the task.

Tasks

In this section description and solutions for individual tasks will be given

1) Lookup collection

The goal of this task was to replace one enumeration column in the dataset with a numeric value. The mapping between the numeric value and the original enumeration value should be saved in a separate lookup collection.

For this task I decided to replace the company column in the original dataset.

First I created the lookup collection, saved it and loaded it into map.

```
private fun prepareCompanyDataset(
    dataset: Dataset<Row>,
    outputDir: String
): Dataset<Row> {
    val companyDataset = dataset.select("company")
        .distinct()
        .coalesce(1) dataset.col("company")
    var whenColumn: Column? = null
    for ((company, id) in companies) {
        whenColumn = if (whenColumn == null) {
            `when`(companyColumn.equalTo(company), id)
        } else {
            whenColumn.`when`(companyColumn.equalTo(company), id)
        }
    companyDataset
        .withColumn("id", monotonically_increasing_id())
        .writeCsv("$outputDir/lookup")
    return companyDataset
}

val companies = prepareCompanyDataset(dataset, outputDir)
    .collectAsList()
    .map { it.getString(0) }
    .mapIndexed { i, company ->
        company to i
    }.toMap()
```

Then I used this map to generate a query that replaces the column values in the original dataset with numerical values.

```
val companyColumn = dataset.col("company")
var whenColumn: Column? = null
for ((company, id) in companies) {
    whenColumn = if (whenColumn == null) {
        `when`(companyColumn.equalTo(company), id)
    } else {
        whenColumn.`when`(companyColumn.equalTo(company), id)
    }
}

dataset.withColumn("company", whenColumn)
    .writeCsv("$outputDir/reviews")
```

2) Oversampling

Since the amount of reviews per company is different, it might be important for some types of analysis to transform the dataset in a way to make sure that all the categories are represented equally. Therefore, for the oversampling task, I decided to even out the amount of reviews per company. I achieved this by inserting some reviews multiple times.

First I calculated how many reviews per company are in the dataset.

```
val reviewsPerCompany = dataset.groupBy("company")
    .count()
    .orderBy(desc("count"))
    .collectAsList()
    .map { it.getString(0) to it.getLong(1) }

val (maxCompanyName, max) = reviewsPerCompany[0]
```

Then I used this to compute how many reviews need to be added.

```
val reviewsToAdd = reviewsPerCompany
        .filter { (name, _) -> name != maxCompanyName }
        .map { (name, count) -> Triple(name, count, max - count) }
```

In the end I used this information to generate a new bigger dataset in which all companies are represented equally.

```
var result = dataset.where(dataset.col("company").equalTo(maxCompanyName))
for ((name, count, toAdd) in reviewsToAdd) {
    logger.task("Oversampling on $name") {
        val reviewsForCompany = dataset.where(dataset.col("company").equalTo(name))

        var buffer = reviewsForCompany
        val fullIterations = toAdd / count
        val reminder = toAdd % count
        val percentage = reminder.toDouble() / count
        logger.log("Need $fullIterations full iterations and another $percentage in the last one")
        for (i in 0 until fullIterations) {
            buffer = buffer.union(reviewsForCompany)
        }
        if (reminder > 0) {
            val subset = reviewsForCompany.sample(percentage)
            buffer = buffer.union(subset)
        }

        logger.log("Final review count for $name is ${buffer.count()}")
        result = result.union(buffer)
    }
}
```

3) Undersampling

As in the case of mongodb, undersampling was easier to implement, because there is a sample operator that takes a randomly selected subset from the dataset. Again, I used undersampling to ensure that the amount of tasks per company is equal.

The initial procedures were almost the same as the previous task, except that I wanted to find out the company with lowest amount of reviews, so that I could sample the reviews from other companies accordingly.

```
var result = dataset.where(dataset.col("company").equalTo(minCompanyName))
for ((companyName, percentage) in percentagePerCompany) {
    val sampled = dataset.where(dataset.col("company").equalTo(companyName))
        .sample(percentage)
    result = result.union(sampled)
}
```

4) Discretizing

For this task I decided to discretize the overall rating column. Originally it contained values from the interval <0.0,5.0> and I transformed it into values from {1,2,3,4,5} by rounding the floating point numbers up to their nearest integer.

```
val col = dataset.col("overall-ratings")
val columnRule = `when`(col.leq(1), 1)
    .`when`(col.leq(2), 2)
    .`when`(col.leq(3), 3)
    .`when`(col.leq(14), 4)
    .otherwise(5)

val discretized = dataset.withColumn("overall-ratings", columnRule)
```

5) Probabilistic analysis

For this task I decided to calculate the probabilities of a review belonging to a company. For that I counted how many reviews are in the whole dataset and then for each company I divided the amount of it's review by the amount of all reviews.

```
val totalReviews = dataset.count().toDouble()
val probabilities = dataset.groupBy("company")
    .count()
    .collectAsList()
    .map { it.getString(0) to it.getLong(1) }
    .map { (name, count) -> name to count / totalReviews }
```

6) Tf-idf

As in the mongodb task, I decided to calculate the tf-idf for the word 'work' in the summary column.

First, I had to count how many documents contain that word.

```
val containsWord = object : Function1<Row, Int>, Serializable {
        override fun apply(it: Row): Int {
            return if ((it.getString(summaryIndex) ?: "").contains(" $word ")) 1 else 0
        }
    }

val count = dataset.map(containsWord, Encoders.INT())
    .agg(sum("value")).collectAsList()[0].getLong(0).toDouble()
```

From that, I could compute idf.

```
val idf = Math.log(dataset.count() / count)
```

Afterwards, I computed tf-idf for each document.

```
val idfCalc = object : Function1<Row, Double>, Serializable {
    override fun apply(it: Row): Double {
        val words = (it.getString(summaryIndex) ?: "").split(" ")
        val workCount = words.count { it == word }.toDouble()
        val tf = workCount / words.size
        return tf * idf
    }
}

val tf_idf = dataset.map(idfCalc, Encoders.DOUBLE())
```

7) Index

Since the column containing longest strings is the summary, I decided to create inverted index for this one.

First, I transformed documents into tuples, where the first element is a word and the second element is a documentId of a document, where this word can be found.

```
val getText = object : FlatMapFunction<Row, Tuple2<String, String>>, Serializable {

    override fun call(it: Row): MutableIterator<Tuple2<String, String>> {
        val id = it.getString(idIndex)
        val text = (it.getString(summaryIndex) ?: "")
            .split(" ")
            .filter { it.isNotEmpty() }
            .map { Tuple2(it, id) }
        return object : MutableIterator<Tuple2<String, String>> {

            var i = 0

            override fun hasNext(): Boolean = i < text.size

            override fun next(): Tuple2<String, String> = text[i++]
        }
    }
}
```

Then I grouped them by the first column, that is the word, merging all the document ids together.

```
val result = dataset.flatMap(getText, Encoders.tuple(Encoders.STRING(), Encoders.STRING()))
    .groupBy("_1")
    .agg(functions.concat_ws(" ", collect_list("_2")))
    .toDF("word", "documents")
```

8) k-fold cross validation

For this task I decided to split the dataset into 5 disjoint datasets that can be used for example to test that a prediction model is not overfitted.

For that I iterated over the original dataset and sliced it.

```
val foldSize = (dataset.count() / k).toInt()

for (i in 0 until k) {
    val start = i * foldSize
```

```
    val rows = dataset
        .where(dataset.col("id").`$greater$eq`(start))
        .limit(foldSize)

    rows.show()
    rows.writeCsv("$outputDir/fold_$i")
}
```

9) Normalization

I decided to normalize the values in overall-ratings, which originally were form the interval <0.0,5.0>.

First I calculated new values in a temporal dataset.

```
val normalizationFunction = object : Function1<Row, Double>, Serializable {
    override fun apply(it: Row): Double {
        val rating = it.getString(colIndex).toDoubleOrNull()
            ?: throw IllegalArgumentException("Could not parse ${it.getString(colIndex)}")
        return rating / 5
    }
}

val resultCol = dataset.map(normalizationFunction, Encoders.DOUBLE())
    .withColumn("id", functions.monotonically_increasing_id())
```

Then I merged it back to the original one.

```
val joined = dataset
        .drop("overall-ratings")
        .join(resultCol, "id")
```

10) Remove noise

I decided to remove all reviews that were older than 1.1.2017, because they are outdated and therefore they are less valuable for people deciding in which company to go now.

```
val filtered = dataset.filter {
    val dateString = it.getString(dateColIndex).trim()
    try {
        val dateFormat = DateTimeFormatter.ofPattern("MMM d, yyyy")
        val date = LocalDate.parse(dateString, dateFormat)
        date >= limit
    } catch (ex: DateTimeParseException) {
        false
    }
}
```

11) Fill missing values

Again I encountered the problem that I could not find missing values. However, to fulfil the task, I decided to create a code that would insert average rating into reviews where overall-rating would be missing.

```
val avgRating = dataset.agg(avg("overall-ratings"))
    .collectAsList()[0]
    .getDouble(0)

val addAverage = `when`(dataset.col("overall-ratings").isNull, avgRating)

dataset.withColumn("overall-ratings", addAverage)
```

12) Pivot table

For this tasks, I decided to compute the average value per company for the following columns. * overall-ratings * work-balance-stars * culture-values-stars * carrer-opportunities-stars

```
val pivot = dataset
    .groupBy("company")
    .agg(
```

```
        avg("overall-ratings"),
        avg("work-balance-stars"),
        avg("culture-values-stars"),
        avg("carrer-opportunities-stars")
    )
```