IT UNIVERSITY OF COPENHAGEN

Project 3: Analyzing Data using Spark ML

KSBIDMT1KU - Big Data Management (Technical)

Teacher: Iman Elghandour

Submitted by

Sophia Luise Berta Nagler

17895

sona@itu.dk

Sophia Anna Katharina Auer

17899

soau@itu.dk

Phase 1 Finding Top Categories in the Input Dataset

def findTopKCategory(productsMetadataDF : DataFrame, k:Int) : Array[String] =

In order to find the top `K` categories of the dataset, a DataFrame is created. The `explode()` method is used for the column categories in order to create a row for each category in the according categories column. It is used twice since categories is a nested array. As testing we used `.show()` to check if the `explode()` method performed correctly. As a next step the data is grouped by (`groupedBy()`) the column categories in order to use `count()`. As the goal of this function is to find out the top `K` categories, the data is ordered by the count in descending order by using `orderBy`. An alternative to `orderBy()` would have been `sort(desc, "count")`. As only the categories are needed, `select()` is used. Since the DataFrame only needs to include the `K` top categories, the `limit(K)` is applied before collecting the data and mapping the according columns to a string. Finally, we needed to get each row into an `Array[String]` whereby we used first the `collect()` method followed by mapping each row to a string as otherwise we would have an `Array[Row]`. Because of using the iterative operations `count()` and `collect()` we persist the `productMetadataDF` before applying the query on it.

In a first test run we used the dataset `meta_Patio_Lawn_and_Garden` and set `k` to 5. The following top Categories were generated. In the final version that is run on cluster we use only 3 top categories (`numCategoriesAsFeatures` = k = 3).

Top Categories run locally:
Patio, Lawn & Garden
Gardening & Lawn Care
Plants, Seeds & Bulbs
Lawn Mowers & Outdoor Power Tools
Replacement Parts & Accessories

Phase 2 Preparing Input Dataset

def findAvgProductRating(reviewsDF : DataFrame) : DataFrame =

In order to find the average reviews rating the DataFrame `reviewsDF` is created. The data is grouped by the column `asin` (representing the product ID). The combination of `avg("overall")` facilitates the aggregation of the average for the rating (column: overall) of the grouped product IDs. Additionally the resulting column `avg("overall")` is renamed to averageRating. This part we changed due to the feedback we got. We've used `agg(avg("overall"))` before as we understood we have to always use `agg` in order to apply an aggregation. Based on research, we figured out that `agg()` has to be only used when using it directly on DataFrames since `agg()` is a DataFrame method supporting to aggregate functions but if you `groupBy` a DataFrame it becomes a RelationalGroupedDataset which has aggregations like `avg()` as methods.

Before applying the query on the `reviewsDF`, we persist it because of the aggregation `avg()`.

Following is our code used for the testing within the main function and the according output based on the dataset `reviews_Patio_Lawn_and_Garden`:

```
amazonReviewsDF
.groupBy("asin")
.avg("overall")).as("averageRating")
.show(10)
```

```
+----------+------------------+
|      asin|      avg(overall)|
+----------+------------------+
|8805002585|               5.0|
|B00002N67Q| 4.486111111111111|
|B00002N8O2|             4.375|
|B00002NCGY| 3.857142857142857|
|B00004R9TL|               4.5|
|B00004S1VJ|3.6333333333333333|
|B00004TR7P|               5.0|
|B00004VWJF|               5.0|
|B00004Y86B|               5.0|
|B0000AX9MN|               3.0|
+----------+------------------+
only showing top 10 rows
```

def findProductFeatures(productsRatingsDF : DataFrame, productsMetadataDF : DataFrame) : DataFrame =

In order to successfully cluster the data in the next step, the data has to undergo another preparation step. A DataFrame is needed with the columns representing the features we're interested in and will be used for clustering the products. First the `productsRatingDF`, created in the previous step, is joined with the `productsMetadataDF` based on the Product ID (`asin`). Then the needed columns (representing the features the clustering is performed on) are selected from the joined DataFrames, namely `asin`, `categories`, `price`, `averageRating`.

Here we got the feedback that we should simplify our code so that we changed the code from
`.join(productsRatingsDF, productsMetadataDF("asin")===productsRatingsDF("asin"))`
`.select(productsMetadataDF("asin"), productsMetadataDF("categories"), productsMetadataDF("price"), productsRatingsDF("averageRating"))` to the current code:

```
def findProductFeatures(productsRatingsDF : DataFrame, productsMetadataDF : DataFrame) : DataFrame = {
  productsRatingsDF
    .join(productsMetadataDF,  usingColumn = "asin")
    .select( cols = $"asin", $"categories", $"price", $"averageRating")
}
```

We realized that if we have a column that is called the same we can simply join based on that column name (as seen above).

Following is the applied function and output based on
`meta_Patio_Lawn_and_Garden`
`reviews_Patio_Lawn_and_Garden`

We tested it within the main function:

```
amazonMetadataDF
        .join(amazonMetadataDF, "asin")
        .select($"asin", $"categories", $"price", $"averageRating")
```

```
+----------+--------------------+-----+------------------+
|      asin|          categories|price|     averageRating|
+----------+--------------------+-----+------------------+
|B00002N67Q|[[Patio, Lawn & G...|16.88| 4.486111111111111|
|B00002N8O2|[[Patio, Lawn & G...|67.99|             4.375|
|B00004R9TL|[[Patio, Lawn & G...|18.72|               4.5|
|B00004S1VJ|[[Patio, Lawn & G...| 4.99|3.6333333333333333|
|B00004Y86B|[[Patio, Lawn & G...| null|               5.0|
|B0000AX9MN|[[Patio, Lawn & G...| null|               3.0|
|B0000CGESB|[[Patio, Lawn & G...|89.99| 4.333333333333333|
|B0001CJ90Y|[[Patio, Lawn & G...| null|               1.5|
|B0001FT39I|[[Patio, Lawn & G...| null|               4.0|
|B0001URNEK|[[Patio, Lawn & G...|27.89|               5.0|
+----------+--------------------+-----+------------------+
```

def prepareDataForClustering(productData : DataFrame, topCategories : Array[String], spreadVal : Double) : DataFrame =

We first written an `udf` where we create out of the three features `topCategories`, `price` and `averageRating` one `Array[Double]` that can be then applied to create a new column named “features” to the `productData` DataFrame. Within the udf we first map every row depending on whether the product is within the `topCategories` (= 1.0) or not (= 0.0). Afterwards, `price` and `averageRating` multiplied by `spreadVal` are appended.

Before applying the udf on the `productData` we persist the `productData` DataFrame as `udf` involves an iterative activity in which it has to iteratively go through each row.

When we tested it we had the following output:

```
+--------------------------------------------------+
|features                                          |
+--------------------------------------------------+
|[1.0, 1.0, 0.0, 0.0, 0.0, 16.88, 4.486111111111111]|
|[1.0, 1.0, 0.0, 0.0, 0.0, 67.99, 4.375]           |
|[1.0, 0.0, 0.0, 1.0, 1.0, 18.72, 4.5]             |
|[1.0, 1.0, 0.0, 0.0, 0.0, 4.99, 3.6333333333333333]|
|null                                              |
|null                                              |
|[1.0, 0.0, 0.0, 0.0, 0.0, 89.99, 4.333333333333333]|
|null                                              |
|null                                              |
|[1.0, 0.0, 0.0, 0.0, 0.0, 27.89, 5.0]             |
+--------------------------------------------------+
```

As you can see from the output, there are a few rows with value `NULL`. This is based on the inconsistencies in the data used for the analysis. In this case the data suggests that if the price is not available the whole row of the dataset will be `NULL` after the join of the two DataFrames.

We counted the entries within the files and realized that indeed there are inconsistencies:

reviews_Patio_Lawn_and_Garden
asin: 1.008.719
overall: 1.015.722

meta_Patio_Lawn_and_Garden
asin: 111.267
price: 81.203

After all, we decided to proactively drop all rows with the value `NULL`. This decision is based on the fact that if the `NULL` values would be replaced with zero, it would distort the clusters. Hence, we used `na.drop` at the end of our query:

```
productData
  .withColumn( colName = "features", createFeatureVector($"categories",$"price", $"averageRating"))
  .select( col = "features")
  .na.drop
```

Phase 3: Discovering Interesting Cluster in the Dataset

def clusterUsingKmeans(data: DataFrame, k: Int, seed: Long):KMeansModel

A new variable `kmeans` is created. The method `KMeans()` is assigned to this variable. As the method KMeans needs the input of how many clusters the data should be divided into, the K is defined with `.setK(k)`. In order to be able to compare the output a seed should be defined. This is based on the functionality of the algorithm which initializes a random data point as cluster center in a first step. The `setSeed(0)` enables the randomization to have the same initialization. We used just a specific seed instead of having an argument for it as we decided not to test this specific characteristic besides Iman mentioning it would be fine like this. As a last step the application of the `KMeans` on the input data is initialized with `kmeans.fit(data)`.

def printKmeansCenters(model : KMeansModel, categories : Array[String],outFilePath: String,spreadVal : Double, dataset: DataFrame):Unit = {

We overall decided to not take the initial print function but instead create a DataFrame that consists of the cluster center prediction, the count of prediction and the List of features.

In the course of this, we first created a DataFrame based on the transformation of the model that consists of the cluster center prediction and features. In this we only took the prediction and grouped the predictions by prediction and counted them, adding this as another column (`val groupedPredictions`).

```
val predictions = model.transform(dataset)
//using persist before applying count()
predictions.persist()
val groupedPredictions = predictions.groupBy( col1 = "prediction").count().orderBy( sortCol = "prediction")
```

After, we've created a udf that takes each `clusterCenter` that matches the `index` (= prediction) and modifies it respectively towards `topCategories`, price rounded and rating divided by `spreadval`. We then converted all features to a List in order to append it to the List of Categories and eventually convert it all to a String to be able to be shown in a column of `groupedPredictions` DF.

After all, we have as output the one seen in the next section "Running on Cluster".

We encountered several problems for this function as it was quite hard to get all the features back to a List or Array without the error of "Schema of type Any is not supported". We eventually realized we need everything to one type and ideally a String. After all, the `toList` and then `toString` methods worked.

```
val predictions = model.transform(dataset)
val groupedPredictions = predictions.groupBy( col1 = "prediction").count().orderBy( sortCol = "prediction")

val clusterCentersInfo = udf { index: Int =>
  //model.clusterCenters(index)
  val centers = model.clusterCenters
  val v = centers(index).toArray.splitAt(categories.length)
  val c = v._2.splitAt(1)
  val completeList = v._1.zip(categories).filter(_._1 != 0).map(p => p._2).toList ::: c._1.map(a => a.round).toList ::: c._2.map(a =>(a/spreadVal).round).toList
  completeList.toString()
}
```

Eventually, we applied the udf on the groupedPredictions DF as a new column called "ClusterInfo", creating only one partition with `coalesce(1)` and write it all on a csv.

Running on Cluster

We ran the clustering on two different datasets (`Patio, Lawn, & Garden` and `Beauty`) with and without persist as well as with two different instance sizes (6 and 9 instances with m5.2xlarge). Overall, we could see a reasonable behavior where with more instances we have a faster execution, larger datasets take longer as well as using persist makes the execution faster than without persist. We get the fastest execution for the Patio, Lawn & Garden dataset taking 29 sec.

As mentioned already within the description of each function we used persist every time before a dataset is used for iterative operations such as `count()`. This has quite an effect on the execution time. See below a summary of our comparison:

	With Persist					Without Persist	
	Features	**k**	**Instance Type**	**Number of Instances**	**runtime**	**Instance Type**	**runtime**
Patio, Lawn, & Garden							
993,490 reviews, 109,094 products	top 3	2	m5.2xlarge	6	**30 sec**	m5.2xlarge	**32 sec**
Beauty							
2,023,070 reviews, 259,204 products	top 3	2	m5.2xlarge	6	**32 sec**	m5.2xlarge	**34 sec**
Patio, Lawn, & Garden							
993,490 reviews, 109,094 products	top 3	2	m5.2xlarge	9	**29 sec**	m5.2xlarge	**30 sec**
Beauty							
2,023,070 reviews, 259,204 products	top 3	2	m5.2xlarge	9	**31 sec**	m5.2xlarge	**33 sec**

We generally had some issues in building a `run.sh` to get it run properly but after trying out different ways it worked fine.That was the reason why we didn't manage to run the k-means clustering on the clusters when we handed in for feedback. Anyhow, this is our final run.sh:

```
#/bin/bash
spark-submit \
  --master yarn \
  --deploy-mode cluster \
  --class dk.itu.BIDMT.F19.P3.AmazonProductsClustering \
  --files application.conf \
  --conf spark.driver.extraJavaOptions=-Dconfig.file=application.conf \
  --conf spark.executor.extraJavaOptions=-Dconfig.file=application.conf \
  BIDMT_F19_P3-assembly-0.1.jar
```

Also we decided to run the `printingKmeansCentersCluster` in the same way as the local version of it so that we create a DataFrame that includes the clusterCenter prediction, the count of prediction and the List of features (`ClusterInfo`):

```
[hadoop@ip-172-31-6-58 ~]$ hdfs dfs -cat out/clusterCenters.csv/*
prediction,count,ClusterInfo
0,46043,"List(Patio, Lawn & Garden, Gardening & Lawn Care, Plants, Seeds & Bulbs, 62, 5)"
1,16182,"List(Patio, Lawn & Garden, Gardening & Lawn Care, Plants, Seeds & Bulbs, 60, 2)"
[hadoop@ip-172-31-6-58 ~]$
```

Overall, the hardest part of this project has been to get the printing function working so that we have one DataFrame including the List of Features.

Visualization

We visualized the k-means clustering locally with just one top category, price and review. Overall, it wasn't clear to us how to visualize more than one top category in one feature. Therefore, we decided to solely visualize it with the second top category of Patio, Lawn & Garden which is Gardening & Lawn Care as the top category is always the used input dataset which doesn't make much sense to visualize. We visualized it with five cluster centers and took the numeric values of the category feature.

We used JupyterNotebook with Python using pandas, numpy and matplotlib:

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

In [2]:
```
df_read = pd.read_csv("BDM-Cluster-Centers.csv")

df = pd.DataFrame(df_read, columns=["Cluster",'Gardening & Lawn Care', 'Price','Rating'])
print (df)
```

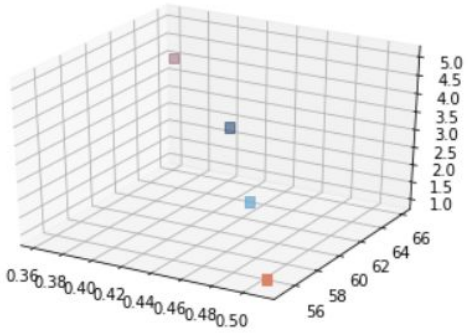
```
   Cluster  Gardening & Lawn Care  Price  Rating
0        0               0.361991     66       4
1        1               0.509283     55       1
2        2               0.377104     60       5
3        3               0.467432     59       2
4        4               0.423471     63       3
```

In [4]:
```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
x = np.array(df['Gardening & Lawn Care'])
y = np.array(df['Price'])
z = np.array(df['Rating'])

ax.scatter(x,y,z, marker="s", c=df["Cluster"], s=40, cmap="RdBu")

plt.show()
```

Based on this, we got a 3D visualization of the k-means clustering:



Summary of Changes we've made after the Feedback

Overall, we streamlined the code based on the feedback we got, see `findProductFeatures()` doing the join in a smarter way and `findProductRating()` using `avg()` without `agg()`.

Additionally, we played around with adding persist as we didn't manage timewise to do so until the submission for feedback. In summary, we added persist to

- `findTopCategory()` function for `productMetadataDF` due to the use of `count()` and collect(),
- `findAvgProductRating()` function for `reviewsDF` because of `avg()`,
- `prepareDataForClustering()` function for `productData` because of applying a udf on each row which is an iterative activity
- `printKmeansCenters()` and `printKmeansCentersCluster()` functions for predictions DF because of `count()` and `groupedPredictions` DF because of the applied udf.

We got the feedback that we do the computation twice by calling the `prepareDataForClustering` one more time when passing the result to the printing function. We would not need that if we would call the printing function within the `clusterAmazonData` function. We don't really understand how to call the printing function within the `clusterAmazonData` function as the the `clusterAmazonData` function is supposed to have as output `KmeansModel` but with the printing function that would lead to "Unit" output. Overall, we know that it's not optimal to call the `prepareDataForClustering` function twice but we couldn't figure out a different way to do so since we need the output of that function within the printing function.