Big Data Management Delivery II: Descriptive and Predictive Analysis

Louis Van Langendonck
Àlex Martorell Locascio

August 5, 2022

1 Introduction

The second part of the Big Data Management course project consists of implementing a formatted zone and an exploitation zone, both mostly relying on Apache Spark [1]. The formatted zone consists of integrating various source data in to a single reconciled one. From here on, the exploitation zone is used to perform both descriptive and predictive analysis. Finally, an artificial stream is ingested on which a previously trained predictive model is applied. All code is written in python and can be found in two notebooks: `formatted+exploitation.ipynb` and `streaming.ipynb`.

2 Formatted Zone

2.1 Data Integration and reconciliation

2.1.1 Data ingestion

The persistent zone out of which the all data is read, consists of four MongoDB [2] Collections: An Idealista collection, an OpenData Income collection, one lookup collection (containing both original lookup `.csv`s and as final data set, another OpenData collection about leisure activities per neighborhood. A row of this "Leisure" data set consists of a neighborhood, with the same naming as in the income data set, and a description of an instance of leisure infrastructure in that neighborhood (fe. el Raval, MACBA, Museum, Plaça dels Àngels, 1, ...)

These data sets are read directly from MongoDB using a Spark-Mongo connector into a dataframe (by default). Each of them is converted into RDD format after which different Spark transformation are applied. An overview of the pipeline can be found in Figure 2. First, some general assumptions are made:

- Assume that the project takes place in an actual big data setting: with bigger data sets and using multiple machines. In practicality this is not the case, as everything runs locally, and automatically only one chunk is assigned per data set.

- To emulate the big data setting some size distinctions are made for the data sets. Given a fixed undefined chunk size, it is decided to artificially assume amounts of chunks assigned to each data source: the smallest data set, lookup, is assumed to be assigned the one partition (one arrow on the abstract overview). The Income data set is initially bigger, such that two partitions are assumed (thus 2 arrows). The leisure is even bigger such that 3 partitions are assumed and finally the idealista data, the biggest data set, is considered to initially contain 4 partitions. This reasoning is purely hypothetical, in order to later reason about the benefits of parallelism, caching, coalescing, ...

2.1.2 Lookup Data

- Initially, only a mapping is applied on the lookup data. This `map` selects the following: as key the neighborhood (as used in either the Idealista or OpenData data sets) and as value the reconciled neighborhood name.

- The choice is made to not use the generated codes as identifier of neighborhoods but instead reconciling the neighborhood names across the data sets and joining on these. Although numerical join keys are also an attractive option, this approach allows for keeping the rows in the idealista that have neighborhoods not in the lookup (which is the case), by using the naming used for the neighborhood (instead of needing a code to identify a neighborhood). However, because it will be used for a predictive model at the end, the codes are appended at the end of every row.

- It is chosen not to remove duplicates, which would reduce the amount of rows from 127 to 111. The little decrease in volume is considered not to be worth the relatively costly `reduceByKey` operation. Note however that as the whole data set is considered to be in one chunk, the reduceByKey would in this case not really be considered a wide dependency.

- This RDD is cached as it is small (so easily kept in memory throughout the whole process without occupying much space) and will be reused for multiple joins.

2.1.3 Leisure Data

- It is chosen to format the leisure such that the RDD would have per row: a neighborhood as key and as value a dictionary with every leisure type and its amount (fe. 'Sarrià', {"Casals d'avis": 2, 'Àrees de jocs infantils': 15, 'Biblioteques': 4, ...}).

- To do this, first the combination of (neighborhood, leisure type) is set as key and as value 1, after which these are reduced with a simple sum to count each of these combinations. Next, only the neighborhood is set as key and filtered on empty keys. Next, on a `GroupByKey` is applied to put all leisure counts next to each other in a dictionary.

- Now, the data is coalesced to 2 partitions. The argument for this is twofold: 1) The data now holds the same amount of data and is grouped by the same key as the resulting income data (next section). This results in a copartition, facilitating the join. As coalesce minimize the amount of shuffling and joins are in general the most expensive shuffling operations possible, the coalesce is considered the more feasible shuffle operation to execute. 2) The `reduceByKey` and `groupByKey` reduce the amount of data by a lot (more than half) thus leaving partitions rather empty and unbalanced.

2.1.4 Income Data

- Similar to the leisure data, the goal is to have as RDD per row: a neighborhood as key and as value a dictionary with for every year the income index (fe. 'Sant Antoni', {2007: '103.8', 2008: '102.9', 2009: '99.5', ... }).

- To do this the neighborhood names are set as key, after a which a `GroupByKey` is applied putting all income indices of all years in one dictionary.

- Now, the income data is joined with the leisure data set first. This is because these share the exact same amount of rows with the exact same keys (one per neighborhood), and as the leisure data is repartitioned into the same amount as income, the two are copartitioned. This makes the join a narrow dependency.

- On the resulting joined RDD, keeping the neighborhoods as key, the lookup table is now joined. This join doesn't generate None-values as all keys are also present in the lookup. A `map` is then applied to rename the neighborhood key to a reconciled one.

2.1.5 Idealista data

- First, the schema of the data set is saved to be reused later (no data included, just meta-data)
- Then, the property code is set as key because it uniquely identifies a property thus allowing for duplicate removal using a `reduceByKey`. This halves the amount of rows from roughly 20000 to 10000.
- Next, the neighborhood is set as key and all rows without neighborhood names filtered out, as they are considered worthless without the information integral to coupling it to the other data. Moreover, the none-values make joining the data difficult. This reduces the rows from about 10000 to about 6700.
- First, the RDD is joined with lookup using a `leftOuterJoin`, because neighborhoods do not uniquely identify a row and moreover, not every neighborhood in idealista is in the lookup table. The former are replaced by the reconciled neighborhood name. For the latter, the the neighborhoodId is set to 'unknown' and the original idealista naming for the neighborhood kept.
- Subsequently, the income-leisure RDD is joined on the idealista RDD, again using a `leftOuterJoin` for the same reasons as before. Subsequently, A `map` sets the property-Code again as key, replaces none values by 'unknown' for both income and leisure. The mapping also flattens the data set such that nesting is removed to facilitate transforming the RDD back in to a Spark Data frame.
- Finally, all data is forced back in to a data frame using a schema consisting of the original one from the idealista data, with at the beginning an additional column for the property code and at the end a column each for neighborhoodId, leisure and income data.
- The final RDD is cached because it will be re-used for different applications (KPI's and model building). It is probably written to disk as it might be to big to fit in memory, but given that the previous transformations are costly because of all the joins, the writing is worth not having to repeat the whole process for every application in the exploitation zone. Note however, that this is probably the weakest link in the whole pipeline of the project, because in an actual big data setting, this data frame might be too big to efficiently write in the user's computer. An actual implementation would thus likely check for row count and file size and depending on this, decide to store the data frame distributively or not.

3 Exploitation Zone

3.1 Descriptive analysis

Three KPI's are calculated, again using Spark RDD operations, thus avoiding distributively storing the formatted data frame. Instead, the calculated KPI data will be stored as local `.csv`-files as these are small in size. The overall structure is summarized in Figure 3. All KPI's are visualized using Tableau and shown within this document. However, in order to access all visualizations interactively, a `KPIs_Dashboard.twbx` is included in the project delivery files.

3.1.1 KPI I: Average listings price/area per amount of theatres, parcs or libraries in the neighborhood.

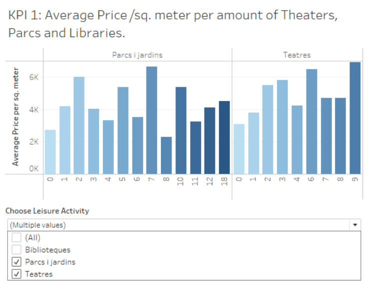
This is calculated by using a `map` to extract for each of the leisure instances mentioned, the amount of them present in each neighborhood, together with the priceByArea (price/m^2) of the listing. Afterwords the average is calculated for each same amount of leisure instances using a `reduceByKey` and `mapValues`. This process is repeated for each of the leisre instances and the result assembled using a `union`. The final RDD is then saved as `kpi1.csv`. Tableau is used to visualize the KPI of which the result is displayed in Figure 1a

3.1.2 KPI II: Monthly evolution (based on scrape date) of the average listings price/area.

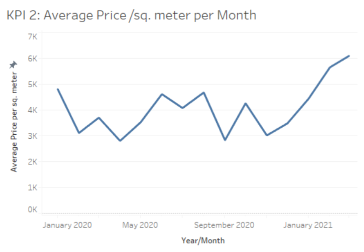
This KPI is calculated by using a `map` to extract the month and year of the scrape date of a listing. The average price/m^2 is then calculated analogous to KPI I and the result saved in `kpi2.csv`. A tableau visualization is displayed in Figure 1b. Note that although the scrape date doesn't always represent the listing date, it does represent correctly the state of the general market of that day.

3.1.3 KPI III: Count of every property type per neighborhood.

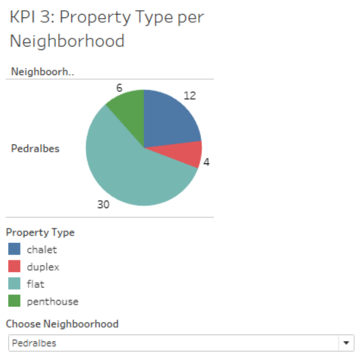
This is calculated by taking as key (neighborhood, propertyType) and as value 1 after which a `reduceByKey` counts the presence of each key. The result is saved in `kpi3.csv` and the tableau visualization can be found in Figure 1c.



(a) KPI no. 1



(b) KPI no. 2



(c) KPI no.3

Figure 1: Visual representation of the three KPIs

3.2 Predictive analysis

For the predictive analysis, a pipeline is build, to ultimately predict the `propertyType` based on a selection of features of mixed type. An overview of the process can be found in Figure 3. First, the cached formatted data is splitted in to train and test using a 80/20 random split. Because multi-level categorical features can not directly be fed to a model (such as the target propertyType or some explanatory variables like 'neighborhood'), encoding is necessary. Therefore, the first two steps of the pipeline consist of: first, using `StringIndexer` to appoint an index to each string of the multilevel category and next, one hot encoding these indices using `OneHotEndocder`. The next step in the pipeline is to assemble all explanatory variables of interest in to one vector using the `VectorAssembler`. Finally, this vector is used as predictor to train a Random Forests model for predicting the propertyType.

These steps are assembled in a pipeline and fitted on the train data. Afterwords, the fitted pipeline is applied on the test set for evaluation. The `MulticlassClassificationEvaluator` is used to evaluate the models performance and yields results of a weighted recall of about 88 %, accuracy of about 88 % too and and f1-score of about 83 %. The pipeline is now saved as `pipeline.model`.

4 Streaming

Because the artificial stream doesn't contain the features used in the predictive model described in Section 3.2, a model predicting just the price from the neighborhoodId is build similarly to the previous predictive model and saved as `pipeline2.model`.

To ingest the stream, a spark connection is made to the producing kafka server. Now, for further processing, Spark Structured Streaming is used instead spark streaming because for the latter, support is depricated for the current version of pyspark. Structured streaming is based on data frame operations and therefore expects queries. The transformations on the stream are thus defined as follows:

1. The 'value' of the stream is casted as a string using "CAST(value AS STRING)".
2. The value column is splitted on ',' and the resulting columns named according to its content.
3. The saved pipeline model is loaded in and a applied on the stream values using `.transform`. The time, neighborhoodId, price and prediction columns are selected.

Now the stream ingestion and query processing is started using `predict.writestream.start()`. As output it is chosen to just display results within the notebook. This choice is warranted assuming that the user of the stream simply wants a program (here notebook) that prints the predictions of the current stream and not per se save it somewhere. However, if necessary, the outputs could easily be written into a .csv. The stream can again be stopped using 'spark.streams.active[0].stop()'. An output example is shown in the overview of Figure 3.

5 Conclusion

The goal was to implement a formatted zone and an exploitation zone using a variety of tools, but mostly relying on spark. Although eventually functional, integrating different sources, software and techniques introduce some difficulties. The pipeline is designed for a big data setting in which it would most likely succeed because of the distributed processing achieved by Spark Jobs. However, the final data frame on which different RDD transformations for exploitation are applied, is currently saved in to memory using `.collect()`. In an actual implementation, this might have to be avoided as the size of the data frame might be too big, and instead be saved distributively in a `.parquet`-file.

References

[1] Apache Software Foundation, *Apache spark.*

[2] MongoDB, *Mongodb.*

Appendix

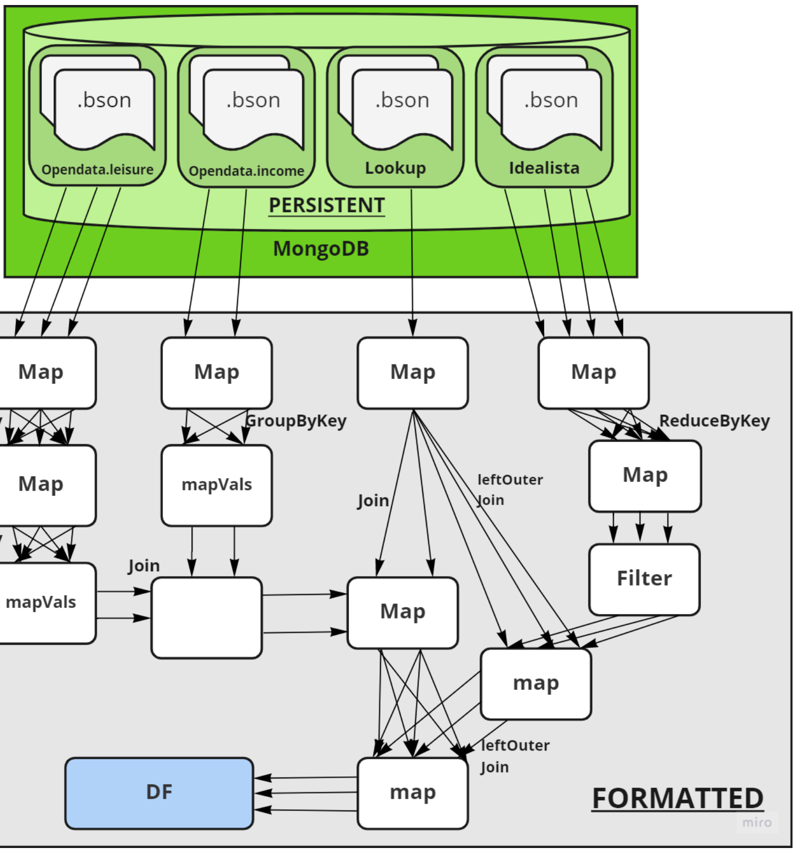
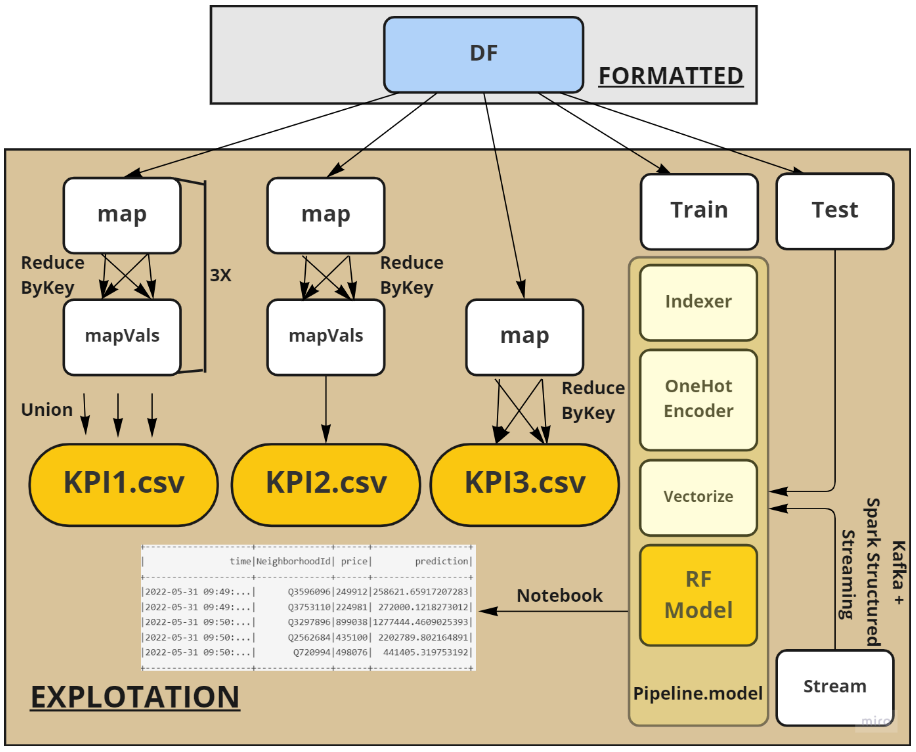


Figure 2: Abstract overview of formatted zone.



```
+-------------------+--------------+------+------------------+
|               time|NeighborhoodId| price|        prediction|
+-------------------+--------------+------+------------------+
|2022-05-31 09:49:...|      Q3596096|249912|258621.65917207283|
|2022-05-31 09:49:...|      Q3753110|224981| 272000.1218273012|
|2022-05-31 09:50:...|      Q3297896|899038|1277444.4609025393|
|2022-05-31 09:50:...|      Q2562684|435100| 2202789.802164891|
|2022-05-31 09:50:...|       Q720994|498076|  441405.319753192|
+-------------------+--------------+------+------------------+
```

Figure 3: Abstract overview of exploitation zone.