Implementation of a (Big) Data Management Backbone

Enric Reverter
Pim Schoolkate

April 2022

1 Introduction

This report explains the architecture of pipeline from the trusted zone to the formatted zone and the creation of KPIs that are moved to the exploitation zone in Apache Spark and visualized in Tableau. Next to this, the assumptions that where made to build this pipeline are stated. Then, a machine learning model is implemented on top of a data stream to facilitate real time prediction.

2 Architecture Formatted Zone

The architecture can be divided into five sections, one for each of the data sources and one for the joining of them. These sources consist of a lookup table with reconciled neighbourhood IDs and names, scraped data from Idealista, and open data on both the average income in each neighbourhood in Barcelona, and the leisure facilities in each neighbourhood. For each of these, the pipeline and assumptions are described.

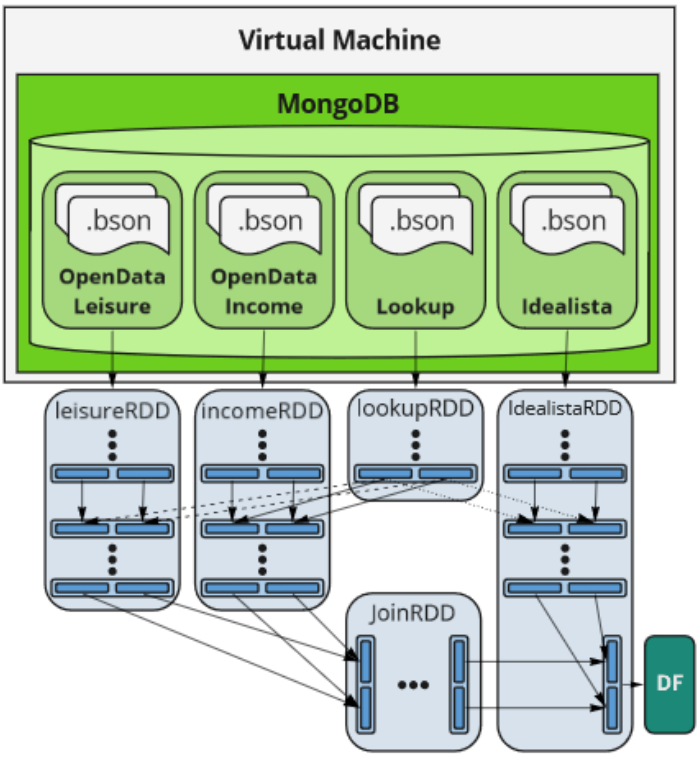

Figure 2.1: The formatted zone pipeline overview. Only joins are shown.

2.1 Lookup data

The lookup data is fed into the RDD with as key the non-reconciled name of the neighbourhood, and as values the reconciled name and ID of the neighbourhood. In total 127 instances exist, of which 16 have duplicate keys and are removed by reducing the RDD by key. The main assumption here is that the reconciled values are the same for all duplicate keys. However, if they would not be the same, this would not be an issue because, when joining, it would result in duplicate rows with "different" (but conceptually the same) neighbourhoods. This is undesirable as this increases the size of the data, slowing down the pipeline, influence statistics when creating the KPIs, or to later avoid these, introduce extra steps in removing the duplicates in the pipeline.

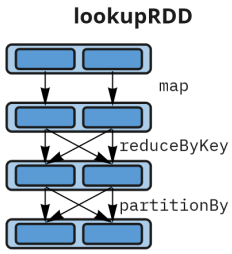

Figure 2.2: The lookup RDD

Next, the RDD is partitioned into two partitions by means of a custom function that is based on the key. The reason for this is that, by default, `pyspark` first partitions the data without considering the key. That is, two different RDDs with some shared keys might allocate its values in different partitions unless specified (and then keep them in there). Further applying `repartition` and `coalesce` results in the same issue (this can be checked using the `glom` method). So, in order to keep narrow dependencies when joining, `partitionBy` is used to consider the key at partitioning. It is worth to mention that unlike `coalesce`, this partition method shuffles the data, but is applied even if there is only one join during the pipeline since it is assumed that it will never harm the performance as the shuffle required for the join costs the same. Now, if one RDD was to be applied in more than one join, co-partitioning as mentioned is more efficient since it will not need to be shuffled again.

Lastly, the RDD is cached as it will be called in total 3 times, once for each join. This way this RDD does not have to be executed each time it gets joined with the other data.

2.2 Open data: Income

The income open data is loaded into an RDD by selecting the name of the neighbourhood as a key, and as value a tuple containing the year and the average income of this year. First, all the instances with the neighbourhood "No Consta" are filtered out, as they are not denoting a neighbourhood, but rather seem to be null values. It is assumed that they are not needed in any analysis and that they might appear in the lookup table. Although they would also be filtered out by the join, these instances do need to be send through the network and thus it is assumed that the cost of filtering out before the join is less than sending these null instances through the network. Two partitions are created after the map and before the join as mentioned in 2.1.

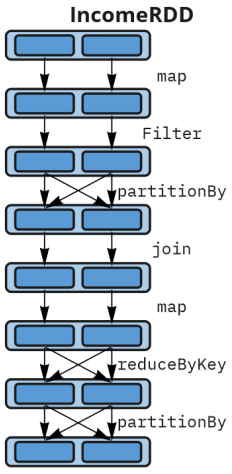

Figure 2.3: The income RDD

After, the income data is immediately joined with the lookup data. This was done so that, instead of the use the non-reconciled names in the open data, the reconciled names can be used in a reduce by key. The main assumption here is that it might be possible that inconsistencies in the neighbourhood names in the open data exist. These can be solved by accounting for this in the lookup data, adding an extra entry for this error. Next to this, not all neighbourhoods in the open data might be in the lookup data, and removing these early from the data by joining helps reduce the data load through the pipeline. Lastly, by having the neighbourhood IDs, it is easy to apply a hash function and equally partition data, guaranteeing narrow dependencies on some of the bigger joins.

Next, the pairs are changed to having the neighbourhood ID and name in a tuple as the key, and as value a dictionary with as key the year and as value the average income.

Lastly, a reduce by key on the neighbourhoods was done, merging each of the dictionaries resulting in a row with a dictionary containing all of the years as keys and the corresponding average income as values for each neighbourhood. Note how data is partitioned again so the future join between income and leisure data becomes narrow.

2.3 Open data: Leisure

The leisure open data enters the RDD pipeline with as key the neighbourhood name, and as value a tuple containing the type of leisure and the count of this type which is currently 1.

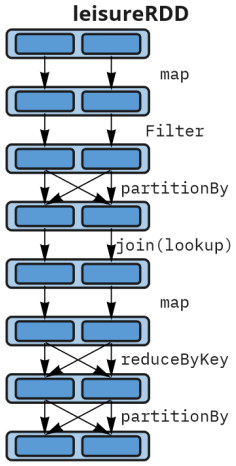

Figure 2.4: The leisure RDD

Next, all instances which do not have a neighbourhood name, but instead an empty string, are filtered out. Main reason for this that these cannot be joined with the lookup table, and as explained above (the income section) this filtering is assumed to cost less than the network load generated by the extra values. Two partitions are created after the map and before the join as mentioned in 2.1.

After filtering out empty strings, the leisure RDD is joined with the lookup table. The same arguments and assumptions described above (the income section) apply here and are thus not iterated.

The keys are now mapped to be the reconciled neighbourhood ID and name, and the values are changed into dictionaries containing as a key the leisure type and as a value the count. Then a reduce by key operation is performed using a custom made `merge_leisure_dict_count` function which adds up the counts of all leisure types of both dictionaries in one single dictionary. Although this function loops over all unique keys in both dictionaries, it is faster than the alternative. This would consist of two reduce by key functions, the first one counting the amount of occurrences of a each leisure type in each leisure type, and then another combining all the resulting counts in a dictionary as done with the income RDD. It is assumed that with scaling up the data, the cost of this looping over the dictionaries is cheaper than the extra shuffling and network load. Note how data is partitioned again so the future join between income and leisure data becomes narrow.

The leisure RDD is partitioned for a second time to prepare it for the join with the income data, see 2.1 for elucidation.

2.4 Joining open data

The two open data RDDs described above are joined using a full outer join on the reconciled neighbourhood ID and name. No other type of join was possible, as could be the case that for some neighbourhood data on leisure or income is missing. Note that this full outer join is a narrow dependency as the amount of partitions have been set to two, just like the amount of partitions for both the inputs of the full outer join.

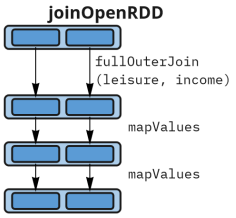

Figure 2.5: The joinOpen RDD

Because of this possibility, any possible null values that result from the full outer join are replaced with empty dictionaries to maintain the homogeneity of the data structure. This is done with `mapValues` function in order to preserve the partition information, which allow for the next join to be also a narrow dependency.

Lastly, it should be noted that in the current implementation a cache is present after joining the data. This is only there because `pyspark`, and especially the `pyworkers` are quite unstable and will crash easily. Without caching the pipeline wouldn't always work. Clearly, in a real application, this cache should not be there.

2.5 Idealista

The chosen key for the Idealista data is the property code, which is a unique identifier for each listing in the data. As values, the Idealista RDD starts of with the neighbourhood name, the scraping date, followed by the complete row. Thus some duplicate data exists in this pipeline, assuming that all of the Idealista data is needed later.

First, the possible duplicate listings (listings with the same property code) are removed, using as a reconciliation criteria the scrap date. The most recent listing is kept. Then, the property code key are swapped for the neighbourhood name and the duplicate scrap date is removed from the data.

Before the join with the lookup data, another partition is made as described in 2.1 to force a narrow

dependency on the join. After the join with the lookup data, the keys are mapped to be the reconciled neighbourhood name and ID on which another partition is applied in order to achieve a narrow dependency join with the joined open data.

When all data is joined, the key-value pair contains many nested tuples in its value. Therefore, a `flatten` function was created that unpacks the nested tuples into one single tuple.

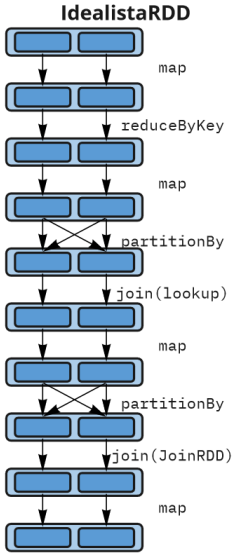

Figure 2.6: The idealista RDD

2.6 Extra assumptions on the Formatted zone

The formatted zone is not saved physically, but rather implemented as pipeline. It is assumed that any exploitation zone can be build on top of the formatted zone pipeline, as shown in the exploitation zone. Within the pipeline, the data from the formatted zone is stored as a `spark DataFrame` as an intermediate step in memory. As RDDs are lazily executed, not saving the data in an intermediate step would mean that for each KPI the whole pipeline needs to be run again. Also, using a data frame allows to store the data on disk as a checkpoint if needed.

3 Exploitation zone

The exploitation zone consists of three KPIs that are calculated using the data from the formatted zone. These are then saved as `.csv` files and visualized in Tableau.

3.1 KPI1: information score

The first KPI calculates the information score of the listing, reflecting how much information is accessible for people looking at the listing. The Idealista data contains seven features that relate to this KPI, namely `hasVideo` which is true or false depending if the listing contains a video tour of the property, `has360` which is true or false depending if a 360 photo is shown on the listing, `hasPlan` which is true or false depending if a plan of the property is included, `has3DTour` which is true or false depending on the listing containing a 3D tour, `showAddress` which is true or false depending if the listing shows the address, and `numPhotos`, the amount of photos.

The KPI is calculated as the sum of each of these features, which are divided by a weight that can either be the total count listings that have this feature set as true for boolen features, or the average for the numeric feature.

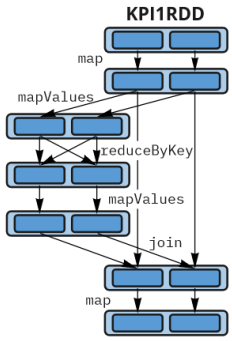

Figure 3.1: The KPI1 RDD

The pipeline for this KPI is as follows. First, for each listing the property code, the neighbourhood name, the price and a dictionary containing the features and the count (in the beginning just 1) are the values, and the key is a string named 'key'. This first RDD is coalesced to 1 in order to facilitate a narrow join later and cached so that it does not need to be called twice.

Then, the weights are calculated in a separate RDD by using a reduce by key and a custom `calc_totals` function to find all totals and next the amount of photos are passed through a function called `calc_averages` to get the average photos.

This separate RDD is then joined with the main KPI1 RDD so that each listing now also contains a dictionary with the weights. This join is a narrow dependency as all keys in this join are a named string with the value 'key' and both inputs only have 1 partition (the first RDD because of `coalesce` and the second because there is only one row). Lastly, the computation is described above is done to calculate the information score for each listing.

The KPI is collected and saved in a `.csv` file named `KPI1.csv`.

3.2 KPI2: amenities per price

The second KPI denotes the average price of the listings in a neighbourhood divided by the amount of amenities/leisure facilities in that neighbourhood. This thus reports the average price one pays for one amenity in the neighbourhood and can give a good overview of how vibrant the area is versus how much is payed for the property.

Because obviously renting and selling prices should not be confused with each other, the KPI differentiates between selling and renting price. However, currently in the data only listings that are being sold are present. Therefore, this differentiation was kept in the case that the data in the future would scale out to renting too.

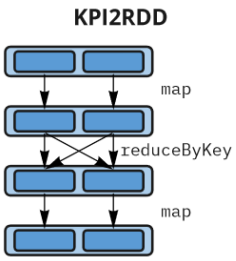

Figure 3.2: The KPI2 RDD

The RDD pipeline starts by mapping the observation to a key conformed by the name of the neighbourhood and the operation (either selling or renting), and the values containing the price, the number of different amenities computed through an auxiliary function, and a 1 to keep track of the count. Next, the observations are reduced by key, where the prices are summed as the count is increased in order to then compute the average. Finally, the observations are mapped so the key becomes the neighbourhood name and the values describe the posted operation and the average price divided by the number of amenities in such neighbourhood.

The KPI is collected and saved in a `.csv` file named `KPI2.csv`.

3.3 KPI3: listings posted per neighbourhood and date

The third KPI counts the number of listings that are posted for a given neighbourhood and month. That is, it allows to observe which neighbourhood have the largest rotation of listings and during which seasons.

To do so, the values are mapped to a key containing the district name and scrapped month (differentiated by year), and a value of 1. Then, observations are reduced by key and its values summed. Lastly, the RDD is collected and stored to be depicted in Tableau as in the previous KPIs.

4 ML and Streaming

Once the above mentioned data is joined, a regressor is trained to predict the size of a listing based on its neighborhood and price. Other variables are not taken into account because the stream to which it is applied only provides such two variables.

To do so, the data is split between train (70%) and test (30%) sets. Then, the data is cleaned through a pipeline that first indexes and encodes the "NeighbourhoodID" into dummy variables and assembles it in a vector altogether with the price. That is, the resulting features consist of a set of columns that indicate the neighbourhood to which the listing is located and its price. Next, a generalized linear model (GLM) is chosen as the model to make the predictions and everything is assembled into a pipeline method. In order to validate the model, 5-fold cross validation is used over the train set at the same time that the parameters of the GLM are tuned applying a grid. Results show the best value for `regParam` is 0.5 with a *R2* of 0.75. Making predictions over the test set with such model yields a *R2* of 0.69, which indicates that the model does generalize well.

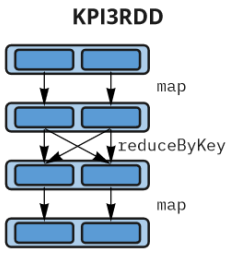

Figure 4.1: The KPI3 RDD

Finally, the model is stored and the streaming instance is initialized using *kafka*. The model is then loaded and used on top of it, which yields the results stored in the provided `.csv` file.