BDM P2

Marc Catrisse, Kathryn Weissman

1 Formatted and Exploitation zone

In this final part of the project, we chose to continue the work Marc did together with Adrià Medina in P1 delivery. While keeping the Third dataset proposed by Kathryn, which contains information regarding the businesses of Barcelona, categorized by neighborhood. The landing zone is supported by HDFS and stores parquet files separated by datasource.

On the other hand, this part of the project is focused on the following layers: formatted and exploitation zones. For the formatted zone, we chose to use a **relational columnar database (MonetDB)**, providing higher performance than a standard database. In columnar databases, data is stored in columns instead of rows, accessing only to the required ones, hence, reduces the amount of data read for a given query.

A **Python** program is in charge of both uploading data from landing to formatted and from there to exploitation, in a script fashion. In Figure 1 we present a sketch of the structure of the whole pipeline. In posterior subsections we describe in depth all considered processes:

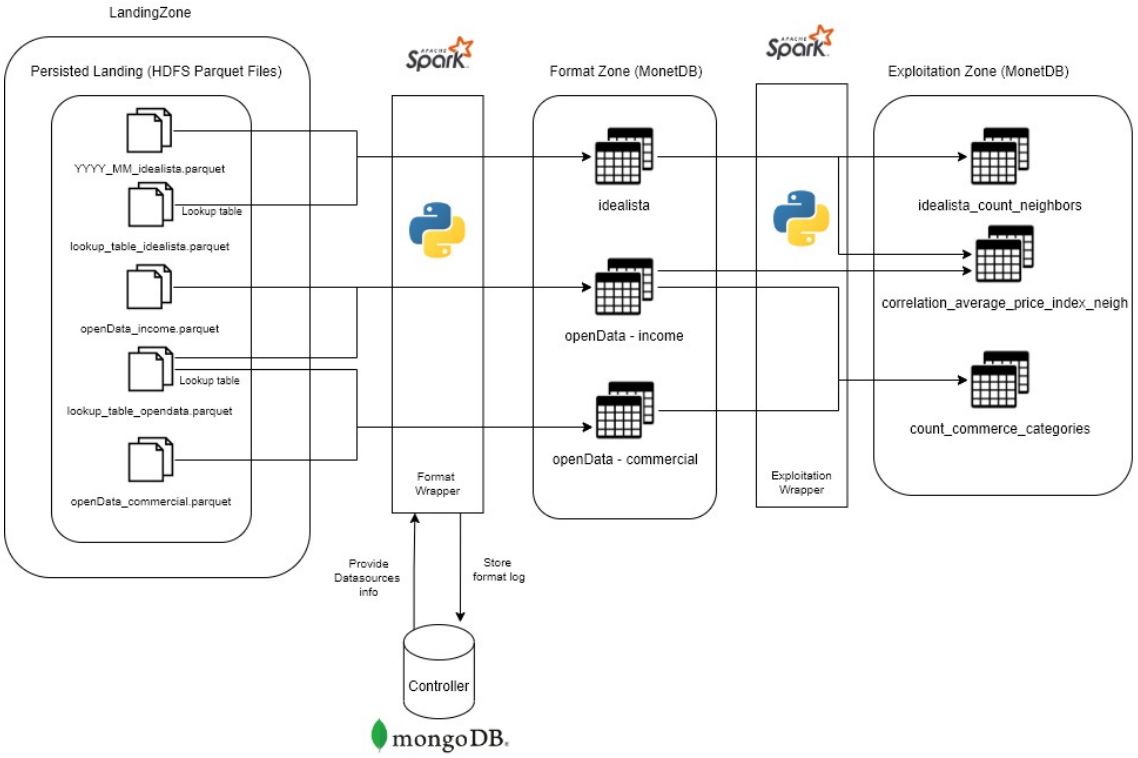

Figure 1: Format and Exploitation zone processing pipeline sketch

1.1 Format zone

When running the python code with the option `--format-process`, unprocessed parquet files are retrieved from the Landing Zone, based on the information stored inside the Mongodb datasources collection.

For each datasource, a Spark pipeline is created. It retrieves the files to process together with the corresponding lookup table (idealista or opendata). Then, a **Left-Join** between the proper data and the lookup table is performed, storing the neighborhood and district information on each row.

Next, we describe the simplified version of the processing Spark pipeline, we are ignoring some specific required steps used for fixing some formatting errors we encountered:

```
spark = SparkSession.builder.master("local[*]").appName("datasource").config('spark.
    driver.extraClassPath','./drivers/monetdb-jdbc-3.2.jre8.jar').getOrCreate()
#Read unprocessed files from HDFS of a given dataset
df = spark.read.parquet(*files_list).withColumn("_process_time", current_timestamp()
    ).withColumn("_input_file_name", input_file_name())
# Read corresponding lookup table
df_lookup_table = spark.read.parquet(lookup_idealista_file)
#Join
df = df.join(df_lookup_table, ['district', 'neighborhood'], 'leftouter')
#Write to MonetDB
df.write.format("jdbc").mode('append').options(url=jdbc_url, dbtable=datasource.
    dest_table).save()
```

Notice, we are also adding some metadata to each row, that help us to track the source file of each row:

- **_process_time**: Timestamp in which the row was processed
- **_input_file_name**: Landing zone hdfs path of the source file, origin of the row

1.2 Exploitation zone

The exploitation zone is intended to be the access point for data scientists to analyze the required KPI's of the data stored in the formatted zone. In our case, we decided to use the same MonetDB relational database used in the Formatted Zone, as the inherent relational schema facilitates retrieving data, together with the columnar management, making scanning fast and efficient. Notice, we are building a single SQL table for each KPI, populated by ad-hoc spark pipelines. Those pipelines are executed when running the python code with the option `--explo-process`, overwriting the whole exploitation zone SQL tables.

2 Descriptive Analysis - Results

In this section, we describe some KPI's we consider interesting to study about the provided datasets, using Tableau as a descriptive analysis tool.

2.0.1 Listing count per neighborhood

It shows the number of idealista listings per neighborhood, showing which one has the higher number of housing available. This data comes from the table "idealista_count_neighbors", populated by the spark code that can be found in Annex (listing 1)

Figure 2 provides an insight of the KPI described before. It shows an important imbalance between neighborhoods, probably due the method used for obtaining the idealista data, showing Dreta Eixample, Sant Gervasi and Sants as the top 3 neighborhoods with the most idealista listings, that seems to be related also to the expected density of residential areas.

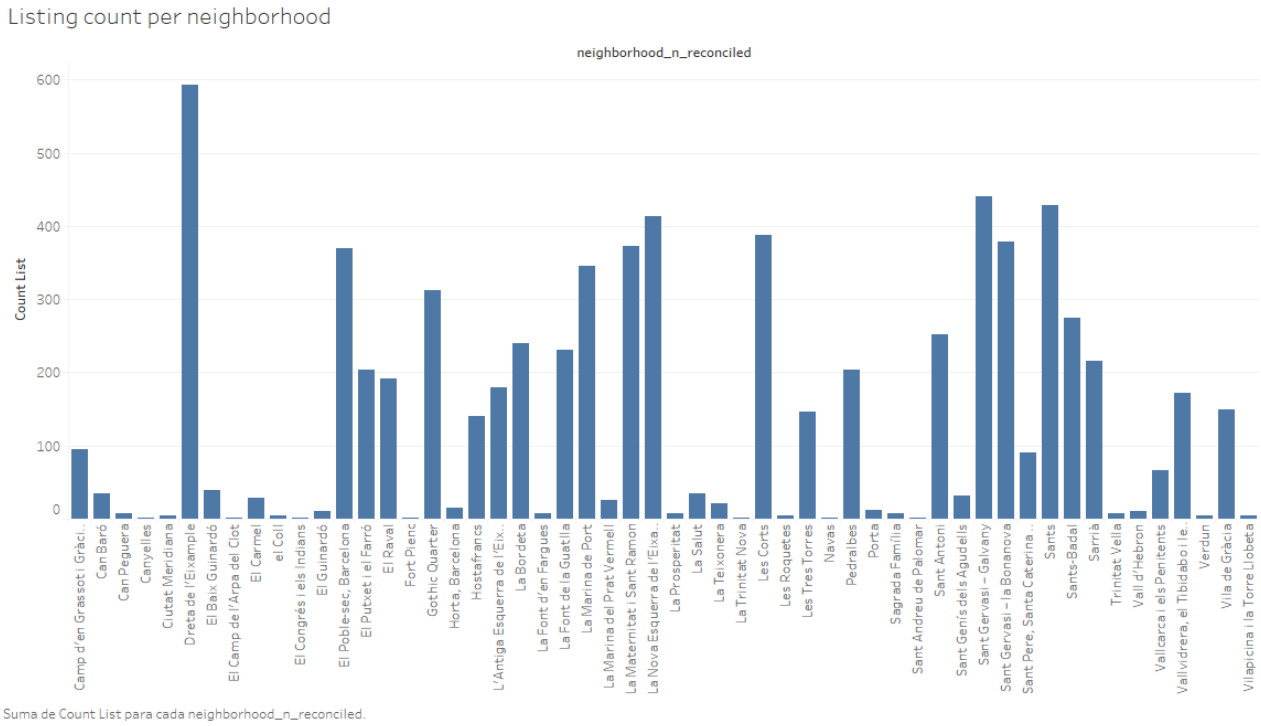

Figure 2: Number of listings per neighborhood - Tableau

2.0.2 Correlation between family income and listing price

This KPI shows the correlation between the average idealista pricing and the average income family index of the neighborhood. Figure 3 provides an insight of the KPI described before, showing a quite lineal correlation, with most of the listings below 800.000 EUR pricing and 120 index rfid.

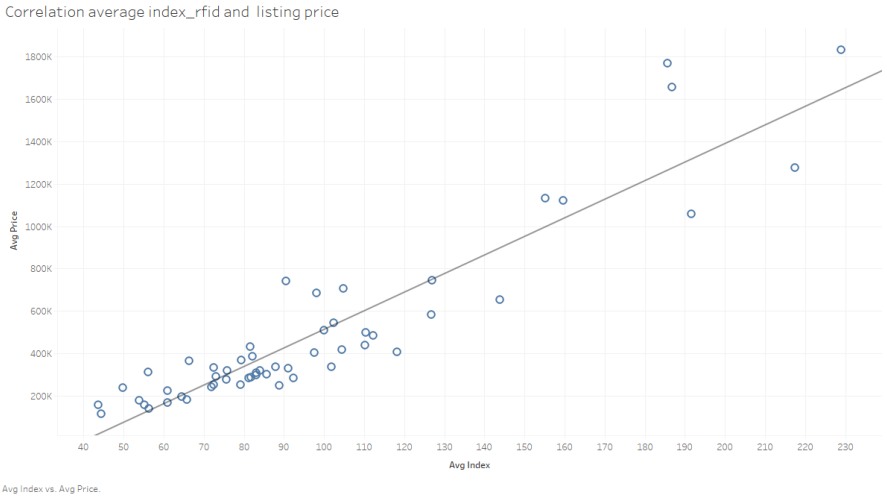

Figure 3: Correlation between family income and housing prices (average) - Tableau

The spark code used can be found in Annex (listing 2)

2.0.3 Commerce insights for each district

Finally, this last KPI provides the number of commercial businesses per district, classified by business category. Figure 4 provides an insight of the KPI described before. It shows clearly that "Eixample" holds

the majority of commerce with difference, with a quite interesting varierty of businesses, predominating the categories "Other" and "Restaurants, bars and hotels". Notice that there are some districts that nearly have commerce, this is related to the number of idealista listings shown in the first KPI. The spark code used can

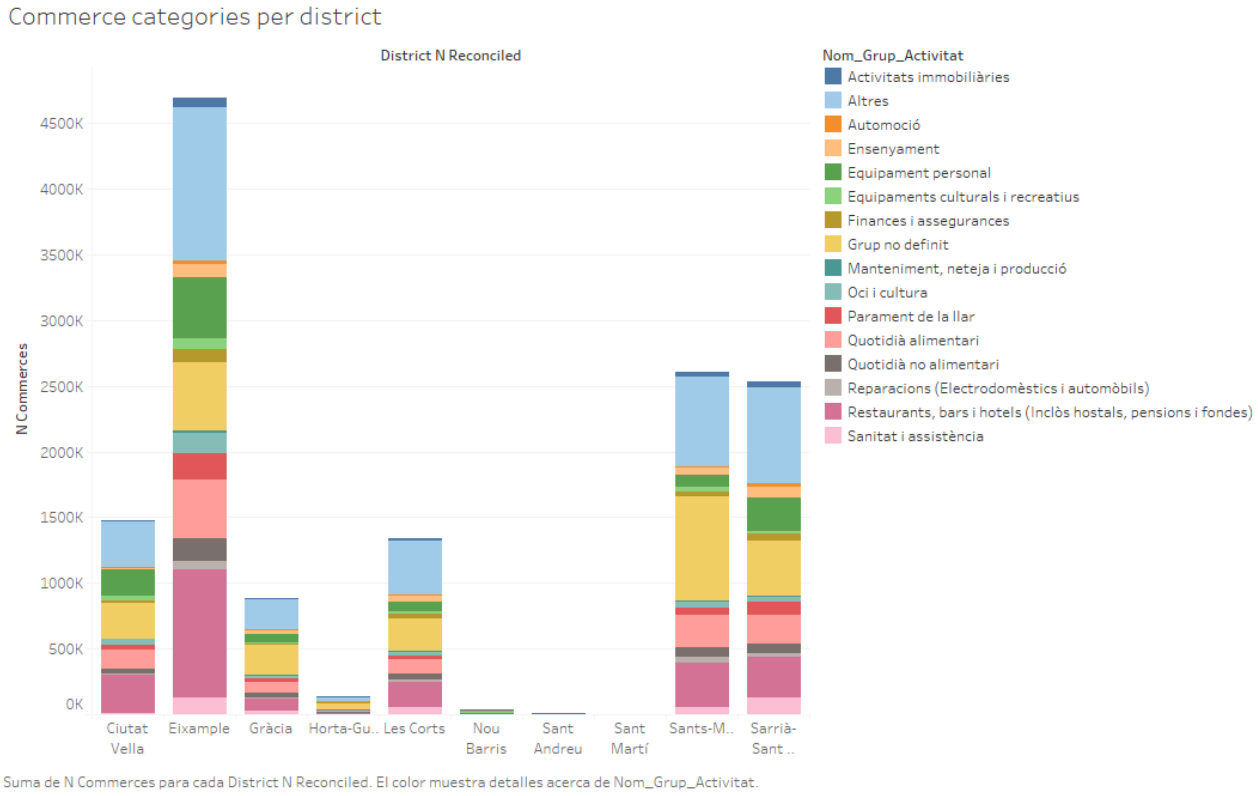

Figure 4: Commerce insights per district - Tableau

be found in Annex (listing 3)

3 Predictive Analysis - ML

We implemented a linear regression model with Spark MLlib using the idealista table from the formatted zone. Provided with a listing, the model predicts the price using only the neighborhood id. Neighborhood id was chosen as the only feature for the regression model so that it is compatible with the data provided in the Kafka stream. We randomly partitioned the data into train and test sets using a 70/30 split, and then created a pipeline that includes pre-processing steps of StringIndexer and OneHotEncoder for the categorical feature. The final step of the pipeline is the Linear Regression model. The validation metrics are shown in Table 1.

The code to develop the model is in the file "MachineLearning.ipynb" and the model is saved in the folder "Fitted-ML-Pipeline".

Metric	**Train Results**	**Test Results**
Root Mean Squared Error	626,955	607,213
R2 Score	0.37	0.36
Mean Absolute Error	325,513	316,830

Table 1: Training and Validation Metrics of Linear Regression Model

4 Apply Machine Learning to a Data Stream

After saving the ML model, it was loaded and applied to a data stream. We used Spark to ingest the data stream and make predictions in real time. The predictions and error were printed to the console while streaming the data. Below are some examples of the streaming predictions.

The code to ingest the data stream and apply the machine learning model is in the file "KafkaStreamML.ipynb".

```
-------------------------------------------
Batch: 1
-------------------------------------------
+--------------------+---------------+-------+------------------+-----------------+
|                date|neighborhood_id|  price|        prediction|            error|
+--------------------+---------------+-------+------------------+-----------------+
|2022-06-21 18:48:...|       Q1904302|3479966|1121793.0438250077|2358172.956174992|
+--------------------+---------------+-------+------------------+-----------------+
-------------------------------------------
Batch: 2
-------------------------------------------
+--------------------+---------------+------+-----------------+-----------------+
|                date|neighborhood_id| price|       prediction|            error|
+--------------------+---------------+------+-----------------+-----------------+
|2022-06-21 18:48:...|       Q1026658|560034|502431.0746205747|57602.92537942529|
+--------------------+---------------+------+-----------------+-----------------+
-------------------------------------------
Batch: 3
-------------------------------------------
+--------------------+---------------+------+-----------------+------------------+
|                date|neighborhood_id| price|       prediction|             error|
+--------------------+---------------+------+-----------------+------------------+
|2022-06-21 18:48:...|       Q3298510|245007|312176.4716073198|-67169.47160731978|
+--------------------+---------------+------+-----------------+------------------+
```

5 Annex

5.1 Exploitation zone - Spark code

```
df = spark.read.format('jdbc').options(url=jdbc_url, dbtable='idealista', **self.properties_db).load()
df = df.select('district_n_reconciled', 'district_id', 'neighborhood_n_reconciled', 'neighborhood_id')
# Convert to RDD
rdd = df.rdd.map(lambda entry: (entry.neighborhood_id, (entry.district_id, entry.district_n_reconciled, entry.neighborhood_n_reconciled, 1)))
rdd = rdd.filter(lambda entry: entry[0] is not None)
rdd = rdd.reduceByKey(lambda x, y: (x[0], x[1], x[2], x[3]+y[3]))
#Map to SQL row
df = rdd.map(lambda entry: Row(neighborhood_id=entry[0], district_id=entry[1][0], district_n_reconciled=entry[1][1],
                neighborhood_n_reconciled=entry[1][2], count_list=entry[1][3])).toDF()
df.write.format("jdbc").mode('overwrite').options(url=jdbc_url, dbtable='idealista_count_neighbors', **self.properties_db).save()
```

Listing 1: pySpark code - Listing count per neighborhood

```
spark = SparkSession.builder.master("local[*]").appName("top_listings_per_neighborhood").config('spark.driver.extraClassPath',
    './drivers/monetdb-jdbc-3.2.jre8.jar').getOrCreate()
df_idealista = spark.read.format('jdbc').options(url=jdbc_url, dbtable='idealista', **self.properties_db).load()
df_opendata_inc = spark.read.format('jdbc').options(url=jdbc_url, dbtable='opendatabcn_income', **self.properties_db).load()
df_opendata_inc = df_opendata_inc.select("neighborhood_id", "index_rfid")
df = df_idealista.join(df_opendata_inc, ['neighborhood_id'])
df = df.groupBy("neighborhood_id", "neighborhood_n_reconciled", "district_id", "district_n_reconciled")\
    .agg(avg("index_rfid").alias("avg_index"), avg("price").alias("avg_price"))
df = df.select('avg_index', 'avg_price', 'neighborhood_id' ,'neighborhood_n_reconciled', 'district_id', 'district_n_reconciled')
 df.write.format("jdbc").mode('overwrite').options(url=jdbc_url, dbtable='idealista_count_neighbors', **self.properties_db).save()
```

Listing 2: pySpark code - Correlation between family income and listing price

```
spark = SparkSession.builder.master("local[*]").appName("top_listings_per_neighborhood").config('spark.driver.extraClassPath',
                './drivers/monetdb-jdbc-3.2.jre8.jar').getOrCreate()
df_idealista = spark.read.format('jdbc').options(url=jdbc_url, dbtable='idealista', **self.properties_db).load()
df_opendata_commerce = spark.read.format('jdbc').options(url=jdbc_url, dbtable='opendatabcn_commercial', **self.properties_db).load()
df_opendata_commerce = df_opendata_commerce.select("neighborhood_id", "Codi_Grup_Activitat", "Nom_Grup_Activitat")
df = df_idealista.join(df_opendata_commerce, ['neighborhood_id'])
df = df.groupBy("neighborhood_id", "Codi_Grup_Activitat", "Nom_Grup_Activitat", "neighborhood_n_reconciled", "district_id", "district_n_reconciled")\
    .agg(count("*").alias("n_commerces"))
df = df.select('neighborhood_id', 'Codi_Grup_Activitat', 'Nom_Grup_Activitat', 'n_commerces', 'neighborhood_n_reconciled', 'district_id', 'district_n_reconciled')
```

```
df.write.format("jdbc").mode('overwrite').options(url=jdbc_url, dbtable='
    idealista_count_neighbors', **self.properties_db).save()
```

Listing 3: pySpark code - Commerce insights for each district